INTERDEPARTMENTAL PROGRAM OF POSTGRADUATE STUDIES (I.P.P.S.)
IN INFORMATION SYSTEMS

MSc Dissertation

M5 COMPETITION TIME SERIES FORECASTING BY IMPLEMENTING EDA, FEATURE ENGINEERING AND MODELLING WITH STATISTICAL AND MACHINE LEARNING METHODS

Written by

ELENI NTZARNTANIDI

Submitted as a prerequisite in fulfilment of the requirements for the acquisition of the postgraduate degree in Information Systems

February 2022

Abstract

The purpose of this dissertation was to analyse the daily sales for several items sold in different Walmart stores in different US cities within 5 years and proceed to forecast the sales for 27 days, based on the historic sales data.

Our methodology was to first conduct an Exploratory Data Analysis (EDA) on the provided datasets in order to better understand and gain insights from our datasets. Then we proceeded with the Feature Engineering (FE) in order to prepare and optimize our data for use in the Modelling phase. Several different Modelling methods were examined and evaluated and the best was selected based on their comparative RMSE.

It was concluded that LightGBM was the best model, which accomplished the highest accuracy in predicting the sales values for the items in the defined period.

Eleni Ntzarntanidi

February 2022

Keywords: Time series forecasting, M5, EDA, Feature Engineering, LightGBM, XGBoost

If you torture the data long enough, it will confess to anything.

Ronald Coase, 1960s

Table of contents

List of tables

List of figures

1. Introduction

1.1. Research scope

The scope of this thesis was to review the M5 competition, examine the existing solutions, implement a thorough exploratory data analysis and feature engineering and finally apply some selected models in practice.

The overarching goal is to investigate the use of both Statistical and Machine learning methods for predicting future sales of different items provided information on the historic sales and contextual information on the nature of the items and the locations they are being sold in, pricing details and calendar characteristics.

1.2. Structure of the thesis

The remainder of the thesis is structured as follows: the concept of the M competitions is presented in Section 2, providing some historical context on previous competitions and introducing the current iteration, its organisational difference compared to the previous once, the provided dataset and the metric that will be used for evaluation of the submitted forecasts.

In Section 3, we examine some selected solutions from Kaggle, with emphasis given on the elegant and/or innovative approaches to the problem they demonstrated. Six notebooks are selected, ranging from those including extensive Exploratory Data Analysis and/or Feature Engineering, those focused on the different Models and those including some combination of the three.

Section 4 presents the Exploratory Data Analysis, establishing the steps of the methodology and implementing them, starting with the data inspection and handling of the missing and null values, moving into the decomposition analysis of the time series and several plots of the aggregated sales from the point of view of the state, store, category and department, including their intersections. Closing the EDA is the seasonality study and the analysis of the explanatory variables, such as events, promotions and prices.

Section 5 discusses the value and necessity of the Feature Engineering step, in order to assist and even make possible the modelling of the forecasting problem. The practical implementation of this is to reduce the data size by applying a downcasting technique, to encode all categorical variables so that they are better digested by the models, and to finally engineer lag and window features which will be incorporated into the time series.

Section 6 includes the modelling, where we try out three classical statistical models for forecasting and two machine learning models; their performance is evaluated using the RMSE metric and compared against the baseline approach and each other. For visualizing each solution, we also plot its prediction for the validation period against the actual value, for a few sample items.

2. M competitions

2.1. Context and previous iterations

The M Competitions (also known as the Makridakis Competitions) are a series of competitions that have been organized since 1982 by forecasting researcher Prof. Spyros Makridakis, intending to evaluate and compare the accuracy of different forecasting methods. The competitions have attracted great interest throughout the years, from both academics and practitioners, by providing objective evidence of the most appropriate way of forecasting various variables of interest, focusing on which models produced good forecasts, rather than on the mathematical properties of those models (Makridakis Competitions, n.d.).

Table 1: History of M competitions

Name	Year	Number of series	Best method	According to
M1	1982	1001 time series	Parzen	MAPE
M2	1993	23 time series and 6 macroeconomic series	Comb	sMAPE
M3	2000	3003 time series	Theta	sMAPE
M4	2018	100,000 time series	Hybrid	sMAPE
M5	2020	42,840 time series & explanatory variables	LightGBM	WRMSSE

The common framework for the competitions is to compare the performance of a large number of time series methods, using experts who provide the forecasts for their method of expertise. Once the forecasts from each expert have been obtained, they are evaluated and compared with those from other experts as well as simple benchmark methods.

2.2. M1 competition

The first Makridakis Competition (Makridakis, et al., 1982), also known as the M Competition, was held in 1982 and it included 1001 time series for which 15 forecasting methods were used, including nine additional variations of those methods.

The following were the main conclusions of the M Competition:

- Statistically sophisticated or complex methods do not necessarily provide more accurate forecasts than simpler ones
- The relative ranking of the performance of the various methods varies according to the accuracy measure being used

- The accuracy when various methods are combined outperforms, on average, the individual methods being combined and does very well in comparison to other methods
- The accuracy of the various methods depends on the length of the forecasting horizon involved

The findings of the study were verified and replicated through other competitions and new methods by other researchers.

2.3. M2 competition

The M2 Competition (Makridakis, et al., 1993) was concluded in 1993, and whereas the first competition had used 1001 time series, M2 used only 29, including 23 from four collaborating companies and 6 macroeconomic series. Data from the companies were obfuscated through the use of a constant multiplier to preserve proprietary privacy.

The competition was organized as follows:

- A call to participate was published in the International Journal of Forecasting, announcements were made in the International Symposium of Forecasting, and a written invitation was sent to known experts on the various time series methods
- The first batch of data was sent to the participants in the summer of 1987
- Participants had the option of contacting the companies involved via an intermediary to gather additional information they considered relevant to making forecasts
- In October 1987, participants were sent updated data
- Participants were required to send in their forecasts by the end of November 1987
- A year later, participants were sent an analysis of their forecasts and asked to submit their next forecast in November 1988
- The final analysis and evaluation of the forecasts were done starting April 1991 when the actual, final values of the data including December 1990 were known to the collaborating companies

The purpose of the M2 Competition was to simulate real-world forecasting better in the following respects:

- Allow forecasters to combine their statistically based forecasting method with personal judgment

- Allow forecasters to ask additional questions requesting data from the companies involved to make better forecasts
- Allow forecasters to learn from one forecasting exercise and revise their forecasts for the next forecasting exercise based on the feedback

2.4. M3 competition

The M3 Competition (Makridakis & Hibon, 2000) was intended to both replicate and extend the M and M2 Competitions' features by including more methods and researchers (particularly researchers in the area of neural networks) and more time series.

A total of 3003 time-series were used with different time intervals between successive observations (such as yearly, quarterly, monthly, etc) in several domains (such as micro, industry, macro, finance, demographics, etc).

To ensure that enough data was available to develop an accurate forecasting model, minimum thresholds were set for the number of observations: 14 for yearly series, 16 for quarterly series, 48 for monthly series, and 60 for other series.

Table 2: Number of time series based on time interval and domain – M3

Time interval*	Micro	Industry	Macro	Finance	Demographic	Other	Total
Yearly	146	102	83	58	245	11	645
Quarterly	204	83	336	76	57	0	756
Monthly	474	334	312	145	111	52	1428
Other	4	0	0	29	0	141	174
Total	828	519	731	308	413	204	3003

The five measures used to evaluate the accuracy of different forecasts were:

1. sMAPE (symmetric mean absolute percentage error)
2. Average ranking
3. Median symmetric APE (absolute percentage error)
4. Percentage better
5. Median RAE (relative absolute error)

2.5. M4 competition

The M4 competition (Makridakis, Spiliotis, & Assimakopoulos, 2020), ran in 2018, utilizing an extended and diverse set of 100,000 real-life time series to identify the most

accurate forecasting method(s), incorporating all major forecasting methods, including those based on Artificial Intelligence (Machine Learning), as well as traditional statistical ones.

M4 aimed to replicate and extend the three previous competitions by (a) significantly increasing the number of series, (b) expanding the number of forecasting methods, and (c) including prediction intervals in the evaluation process as well as point forecasts.

Table 3: Number of time series based on time interval and domain – M4

Time interval*	Micro	Industry	Macro	Finance	Demographic	Other	Total
Yearly	6,538	3,716	3,903	6,519	1,088	1,236	23,000
Quarterly	6,020	4,637	5,315	5,305	1,858	865	24,000
Monthly	10,975	10,017	10,016	10,987	5,728	277	48,000
Weekly	112	6	41	164	24	12	359
Daily	1,476	422	127	1,559	10	633	4,227
Hourly	0	0	0	0	0	414	414
Total	25,121	18,798	19,402	24,534	8,708	3,437	100,000

The competition was organized as follows:

- The M4 competition was first announced in November 2017 and like the previous three it was an open competition with the aim of ensuring fairness and objectivity
- The M4 dataset was created on December 28th, 2017 – selecting 100,000 time-series from a database compiled at the Technical University of Athens (called ForeDeCk) that contained 900,000 continuous time-series, built from multiple, diverse and publicly accessible sources
- The training set was made available to the participants at the beginning of the competition, while the test set was kept secret till its end when it was released and used by the organizers for evaluating the submissions
- The participants were asked to produce several forecasts beyond the available data that they had been given, based on the time interval of the series (e.g., 6 for yearly, 8 for quarterly, 18 for monthly series, etc)

To evaluate the performances of the forecasting methods, the OWA (overall weighted average) was used, which is the average of two of the most popular accuracy measures:

1. sMAPE (symmetric mean absolute percentage error)
2. MASE (mean absolute scaled error)

Some major findings of the M4 competition were:

- All of the top-performing methods were combinations of mostly statistical methods, with such combinations being more accurate numerically than either pure statistical or pure ML methods
- There were significant differences between the six top-performing methods and the rest
- On average, the increase of the computational time to apply more sophisticated and complex methods led to greater forecasting accuracy, signifying that processing power was exploited beneficially (with some notable exceptions)
- The top-performing methods introduced information from multiple series (aggregated by frequency) to predict individual ones
- The pure ML methods submitted were less accurate than some simple statistical benchmarks; however, the paucity of entries in the ML category (only 5) should be taken into consideration

2.6. M5 competition

The M5 competition (Makridakis, Spiliotis, & Assimakopoulos, 2020) ran in 2020 on the Kaggle platform, attracting more than 7,000 participants from 101 countries.

As defined by its creators, the objective of the M5 competition remained similar to the previous four iterations: to advance the theory and practice of forecasting and improve its utilization by businesses and non-profit organizations. More specifically, the aim was to:

- Identify the methods that provide the most accurate forecasts for different types of situations, i.e., the different time series of the competition
- Estimate the uncertainty distribution of the realized values of these series as precisely as possible
- Compare the accuracy/uncertainty of Machine Learning methods, compared to that of standard statistical ones, and assess possible improvements versus the extra complexity and higher costs of using the various methods

The competition was split into two parallel tracks:

1. the **M5 Forecasting Competition - Accuracy**, where the participants were asked to provide 28 days ahead, point forecasts (PFs) for all the series of the competition, and

2. the **M5 Forecasting Competition - Uncertainty** where the participants were asked to provide 28 days ahead probabilistic forecasts for the median and four specified (50%, 67%, 95%, and 99%) prediction intervals (PIs).

Moving forward, we are going to focus on the *Accuracy* track.

2.7. Organization

The competition was divided into two phases, the validation and the test phase.

The validation phase took place from March 3rd, 2020 to May 31st, 2020. During this phase, the participating teams were allowed to train their methods using the training dataset consisting of 1913 days and validate the accuracy of their forecasting methods using a sample of 28 days (corresponding to days 1914-1941), which was not publicly available at the time. Each team could submit a maximum of five entries per day, with their results published onto Kaggle's real-time leader-board. The purpose of this phase was for the teams to assess their methods and effectively revise and resubmit their forecasts by utilising the feedback received, all while exchanging ideas and insights with the rest of the community.

The test phase took place from June 1st to 30th, 2020. During this phase, the teams were provided with the data for the 28 days from the validation phase and were asked to re-estimate and/or adjust their forecasting methods, to submit their final forecast (only one submission per team). No leader-board was available during this time with the final ranks of the teams being disclosed only at the end of the competition (June 30th).

2.8. Dataset

The M5 dataset refers to products sold in the Walmart stores in the USA for 5+ years.

The provided time series include the daily unit sales grouped hierarchically, starting at the product-store level and aggregated to the level of product departments, product categories, stores, and geographical states. More specifically:

- Unit sales of 3,049 unique products are provided
- Each product is classified into one of the three product categories (Hobbies, Foods, and Household) and the seven product departments in which the categories are disaggregated (Hobbies 1, Hobbies 2, Foods 1, Foods 2, etc.)
- We are looking at sales of the products across ten stores (CA 1, CA 2, TX 1, TX 2, etc.) which are located in three states (California, Texas, and Wisconsin)

- The result is 42,840 time series, most of which display intermittencies (i.e., the item sales include multiple zeros due to sporadic demand for each product)

All possible cross-sectional levels of aggregation are presented in the table below. The various level ids do not indicate an actual hierarchical structure, they merely highlight the extent of aggregation that takes place: high levels of aggregation generally correspond to low identification numbers (e.g., levels 1 to 5), while low levels of aggregation to higher identification numbers (e.g., levels 10 to 12).

Table 4: Number of M5 series per aggregation level

id	Level description	Aggregation level	# of series
1	Unit sales of all products, for all stores/states	Total	1
2	Unit sales of all products, for each State	State	3
3	Unit sales of all products, for each store	Store	10
4	Unit sales of all products, for each category	Category	3
5	Unit sales of all products, for each department	Department	7
6	Unit sales of all products, for each State and category	State-Category	9
7	Unit sales of all products, for each State and department	State-Department	21
8	Unit sales of all products, for each store and category	Store-Category	30
9	Unit sales of all products, for each store and department	Store-Department	70
10	Unit sales of product *x*, for all stores/states	Product	3,049
11	Unit sales of product *x*, for each State	Product-State	9,147
12	Unit sales of product *x*, for each store	Product-Store	30,490
Total			**42,840**

The dataset also includes explanatory variables, such as calendar-related information (weekends, special events, holidays, SNAP[1] activities) and selling prices for each product

[1] The United States federal government provides a nutrition assistance benefit called the Supplement Nutrition Assistance Program (SNAP). SNAP provides low-income families and individuals with an Electronic Benefits Transfer debit card to purchase food products. In many states, the monetary benefits are dispersed to people across 10 days of the month and on each of these days 1/10 of the people will receive the benefit on their card. More information about the SNAP program can be found here: https://www.fns.usda.gov/snap/supplemental-nutrition-assistance-program

on a store level, that typically affect unit sales and could improve forecasting accuracy if taken into consideration.

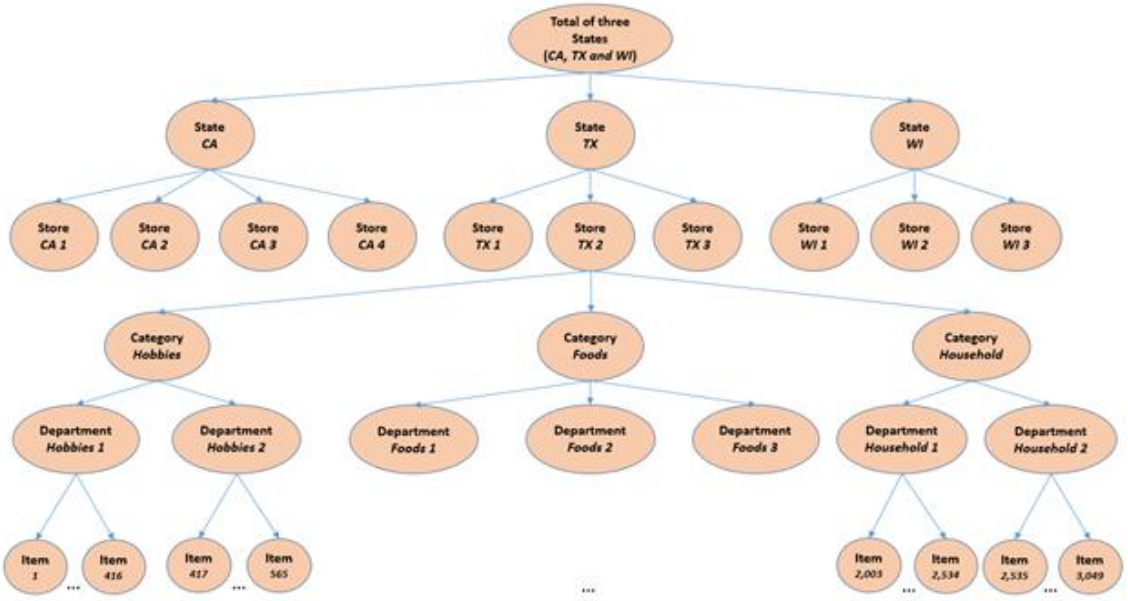


Figure 1: An overview of the hierarchical organization of the M5 time series

2.9. Performance metric

The M5 accuracy competition uses the WRMSSE (Weighted Root Mean Squared Scaled Error) metric to evaluate the performance of the submitted methods.

The justification for the selection of this metric was that such measures based on scaled errors seem to display the most preferable statistical properties. More specifically, the following points were considered:

- The competition aims at accurately forecasting average sales in a dataset that has a significant number of zeros (several products have many days with no sales), therefore there was a need for a measure based on squared errors and optimized for the mean instead of e.g., absolute errors optimized for the median which would assign lower scores to methods with forecasts close to zero (Kolassa, 2016)
- The selected metric can be safely computed for all time-series due to its lack of reliance on divisions with values close or equal to zero, contrary to other measures such as relative errors, percentage errors, etc

First, the RMSSE metric is estimated for each of the 42,840 time-series:

$$RMSSE = \sqrt{\frac{\frac{1}{h}\sum_{t=n+1}^{n+h}(y_t - \hat{y}_t)^2}{\frac{1}{n-1}\sum_{t=2}^{n}(y_t - y_{t-1})^2}}$$

- y_t is the actual value of the examined time series at point t
- $\hat{y}_t$ is the forecasted value at point t
- n is the number of available observations
- h is the forecasting horizon (28 days in our case)
- the denominator is computed only for periods during which the examined product is actively sold, i.e., periods following the first non-zero demand observed for the examined time-series; this is necessary due to many products not being available at the beginning of the examined period, but rather being sold later within the >4 years.

Once the RMSSE for each time series is estimated, their average is calculated using appropriate weights to get the final accuracy measure:

$$WRMSSE = \sum_{t=1}^{42,840} w_i \times RMSSE_i$$

- w_i is the weight of the ith series of the competition, computed as the sum of units sold in the last 28 days of the training sample multiplied by their respective price; this way, the more valuable products to the company are given higher importance within the dataset
- $RMSSE_i$ is the score of the ith series of the competition

The lower the WRMSSE scores are, the more accurate the forecasts are considered.

Incidentally, the metric used for the uncertainty competition was WSPL (Weighted Scaled Pinball Loss).

3. Examination of solutions in Kaggle

According to the official results and findings published by the M5 competition organizers (Makridakis, Spiliotis, & Assimakopoulos, 2020), the 5,507 participating teams from 101 countries made in total 88,136 submissions, and for 22% of the participants (including 15 in the top 100) this was their first time participating in a Kaggle competition.

Figure 2 summarizes the analysis of the participating teams and submission, presenting in the **top-left** the daily number of submissions made (black line) and the cumulative number of participating teams (blue line) with the red dotted line indicating the end of the validation phase; in the **top-right** we can see the number of participants per country as estimated based on their IP address; in the **bottom-left** we can see the distribution of the accuracy of the teams that did better than the Naive benchmark (greed dotted line indicated the accuracy of the ES_bu benchmark and the purple dotted line the accuracy of the sNaive benchmark); in the **bottom-right** we can see the accuracy of the teams that did better than the top-performing benchmark (ES_bu[2]), along with their respective ranks.

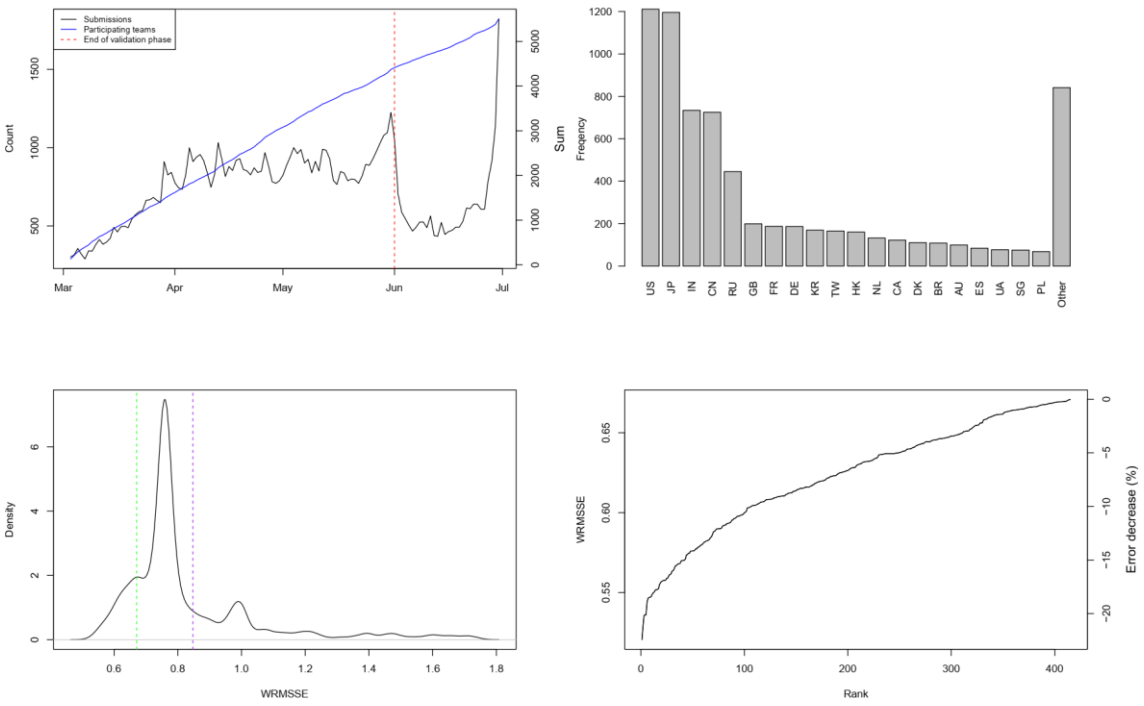


Figure 2: Summary of the participating teams and submissions made

Below are some further observations regarding the information presented in these charts:

[2] The Exponential Smoothing with bottom-up reconciliation benchmark (ES_bu) employs an algorithm to automatically select the most appropriate exponential smoothing model for forecasting the product-store series of the dataset (level 12), and then the rest of the series (levels 1-11) are predicted using the bottom-up method to ensure that the forecasts derived across the various aggregation levels are coherent.

- The majority of the teams (78%) made most of their submissions during the validation phase until June, when they still had access to the public leader-board and could receive live feedback for their submissions. Meanwhile, during the test phase they probably mostly fine-tuned their methods in private and the last spike of submissions was made in the four days before the end of the competition in July.
- The majority of the participants were located in the USA (17%), Japan (17%), India (10%), China (10%) and Russia (6%) showing a large, active community in both developed and developing countries that is interested in forecasting.
- Out of the participating teams, 48% managed to outperform the Naive benchmark in their final submission, 36% outperformed the sNaive benchmark, and only 7.5% beat the top-performing ES_bu benchmark. However, it is worth mentioning that many teams failed to choose the best amongst the methods they developed for the final submission, probably due to overreliance to potentially misleading validation scores.
- From the 415 teams that managed to outperform all the benchmarks of the competition, 5 displayed an improvement greater than 20%, 42 greater than 15%, 106 greater than 10%, and 249 greater than 5%. These improvements are substantial and demonstrate the superiority of these methods over standard forecasting approaches. Moreover, the five winners of the competition were the only teams to accomplish an accuracy improvement greater than 20%, thus achieving a clear victory over the rest.

In the next chapters we are going to examine some of the kernels/notebooks from Kaggle users that were used in the current thesis and have been singled out due to their remarkable appearance, cohesive structure and straightforward solution presentation – most of them use Python but this was not a prerequisite. In the bibliography, there are also references to some additional noteworthy notebooks that we have not reviewed here.

3.1. Back to (predict) the future - Interactive M5 EDA

We will start with the description of arguably one the most extensive notebooks in Kaggle, namely the "Back to (predict) the future - Interactive M5 EDA" (Notebook by Heads or Tails, 2020).

The primary focus of this notebook was a thorough exploratory analysis of the competition, diving deep into all relevant aspects and using the R programming language for this purpose.

The notebook starts with some background information regarding the competition, presenting its goal and dataset on a high level. It moves on with loading all the necessary libraries, defining a helper function to be used later on for computing binomial confidence intervals, and sequentially loading the data into some dataframes (namely `train`, `price`, `calendar` and `sample_submit`).

Next, there is a quick look into the structure and content of the three dataframes, getting some summary statistics and printing some sample rows, while also conducting a missing and zero values analysis on the dataframes. Before moving on, the sales data are transformed from a wide into a long format, and then starts the visual overview of the sales time series, with the use of several interactive plots on different aggregation levels.

In order to get some insight into the time series data, 50 random item-level time series are selected and plotted in an interactive plot (Figure 3) where the user can select an item and see its daily sales for the examined period; it is possible to zoom in to more closely examine a date range. The abovementioned aggregated sales plots follow the individual items analysis, including some interesting grouped arrangement of individual plots, facetted views and grid plots for demonstrating the sales per state, store, category and department.

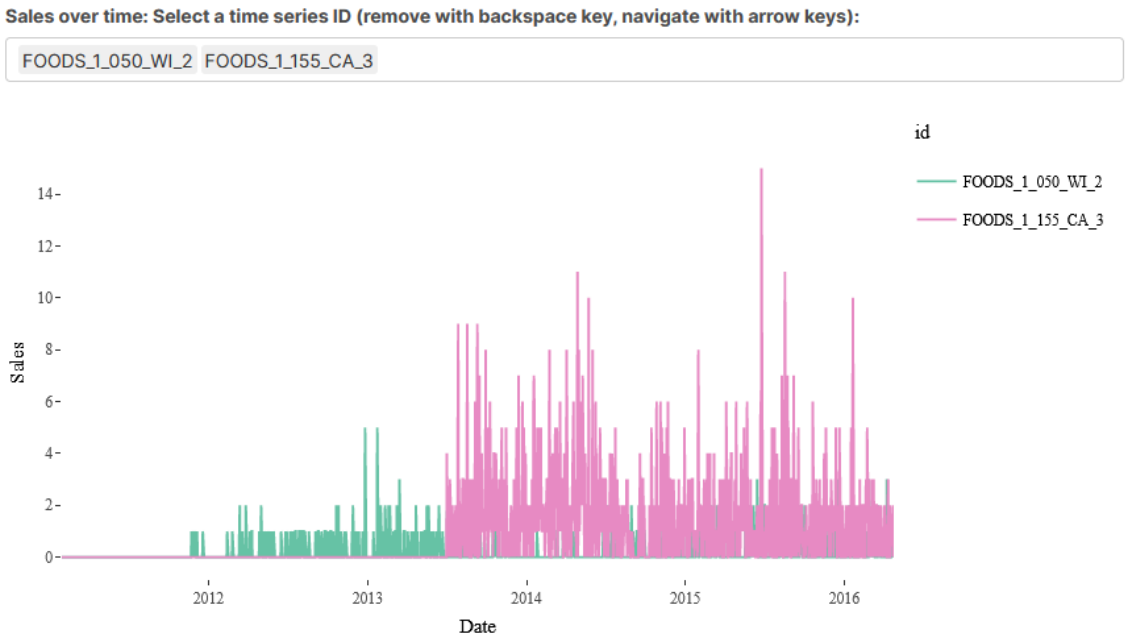


Figure 3: Interactive plot for item-level sales over time

Moving on from the time series views, a heat map is used to study weekly and yearly seasonality, showing the relative changes in sales instead of absolute sales values due to the general increasing trend in sales distracting from the purpose of the plot otherwise. This smoothing approach is further used in the analysis at the state and category levels.

Next off, there is a section focusing on the item prices and calendar events, in order to study their basic properties and eventually connect them to the time series data. Several plots are used to examine and present the days with events, the types of events and the SNAP days performance for which an elegant calendar view is also created to showcase their occurrence per state throughout the year (Figure 4). For the item prices, there are some facet grids with overlapping density plots for price distributions within the groups of category, department, and state. After these analyses, the influence of the explanatory variables on the time series is explored with the use of some custom views per aggregation level that include some useful line charts, boxplots and bar charts.

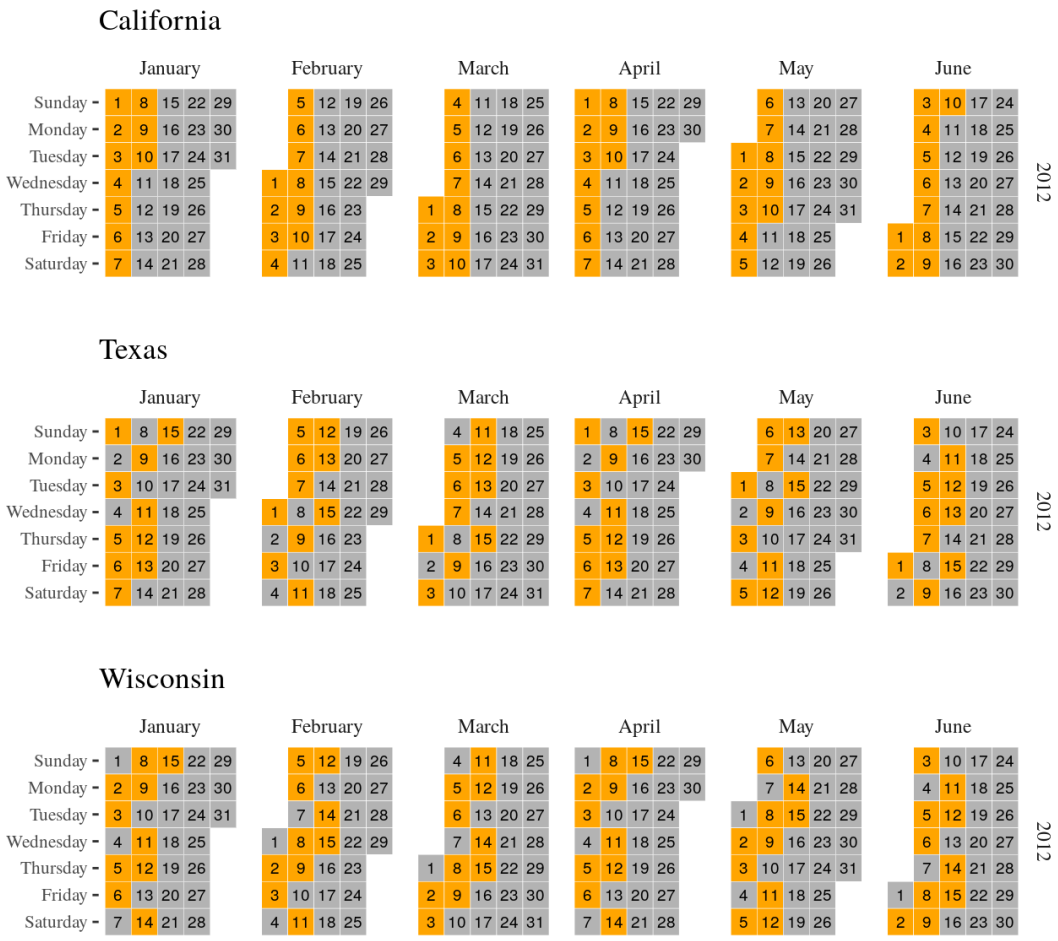


Figure 4: Calendar view of SNAP days

After showing again some individual time series and the effect of the explanatory variables on the, the analysis concludes with some summary statistics such as the density plots of zero values percentage, mean sales, mean item price, etc.

Finally, as a bonus, an external dataset is included in the analysis – specifically the dataset of natural disaster declarations in the US in order to examine their impact on the provided sales (showcased in the custom view at Figure 5).

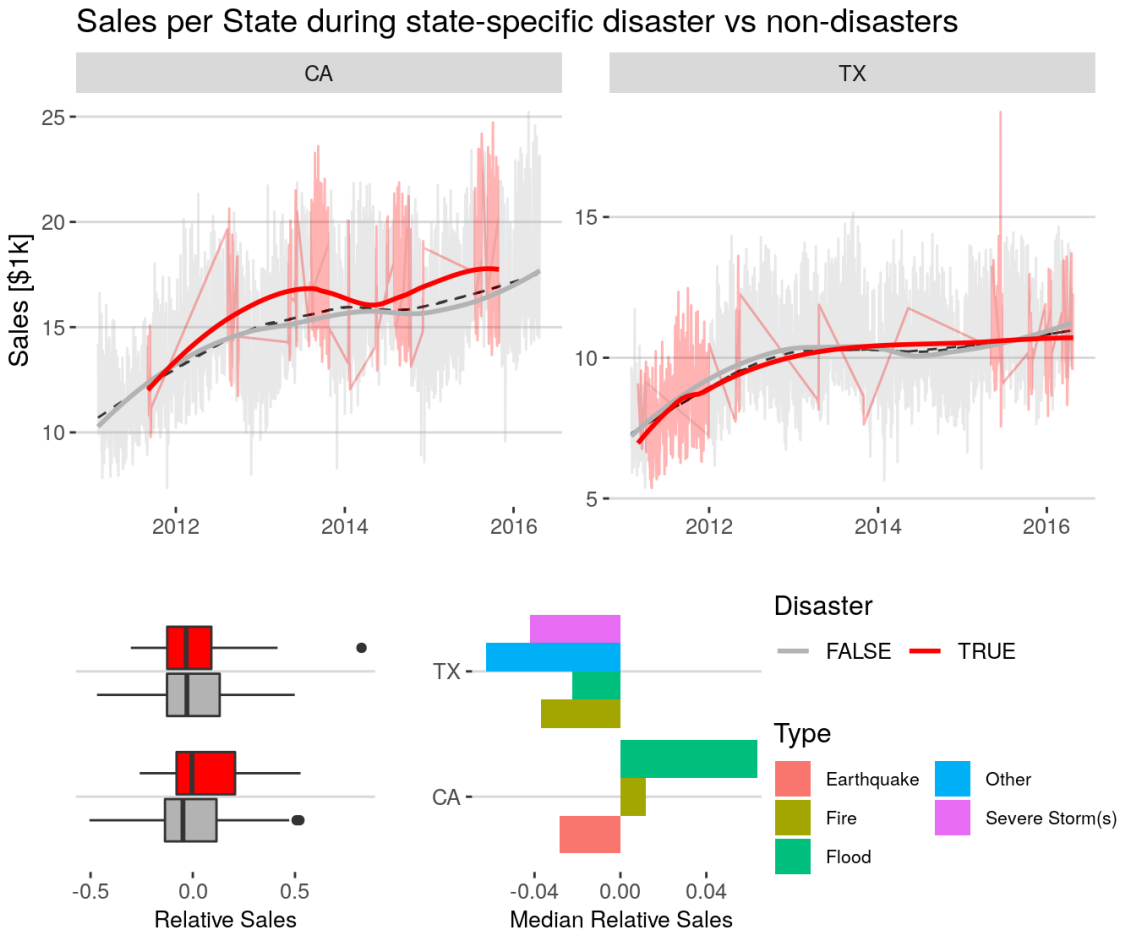


Figure 5: Custom view of sales performance

3.2. M5 Forecasting Exhaustive EDA Beginner

We continue with another EDA-focused notebook, namely the "M5 Forecasting Exhaustive EDA Beginner" (Notebook by Anirban Sen, 2020).

This notebook also starts with an introduction of the competition, stating the problem to be addressed and continuing with a first look into the dataset, after loading all necessary libraries; this notebook was built in Python.

After providing some summary statistics on the sales data, the notebook continues by plotting some basic pie charts and 100% bar charts to get a look into the relative share of

sales for the different states, stores, categories etc. There is also an analysis of the price data by plotting a density plot of the prices and some boxplots showing the price change over the years, while the analysis of the calendar data is focused on the different event types and the monthly occurrence of SNAP days within each state.

Following are some views on the time-series, including the decomposition analysis, and some line charts for different aggregation levels over time. There are also some bar charts and heatmaps showing sales over the days of the week, the months of the year and the days of the month across different dimensions (such as categories, seen in Figure 6).

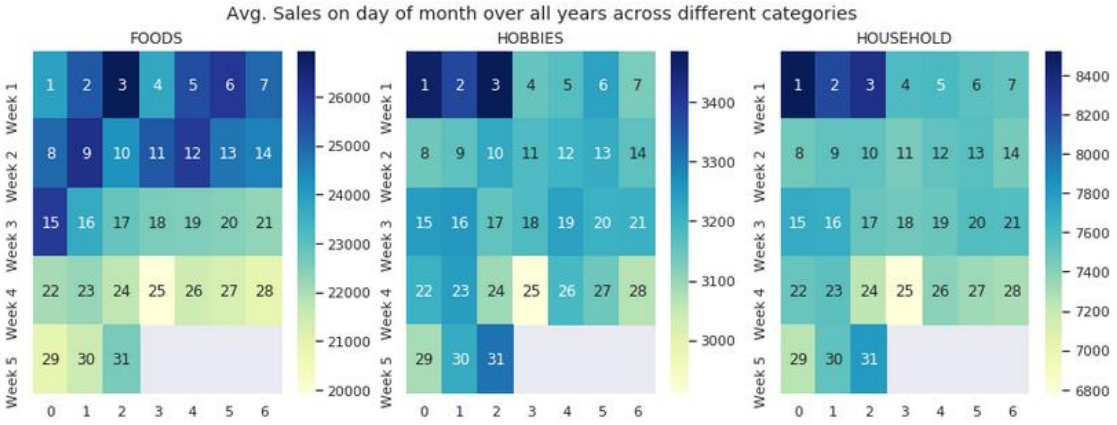


Figure 6: Heatmap of sales across years and categories

Furthermore, there is an analysis of the impact of events and SNAP days on sales, plotting sales of the weekends that preceded the different event types or event names (Figure 7), and on SNAP days.

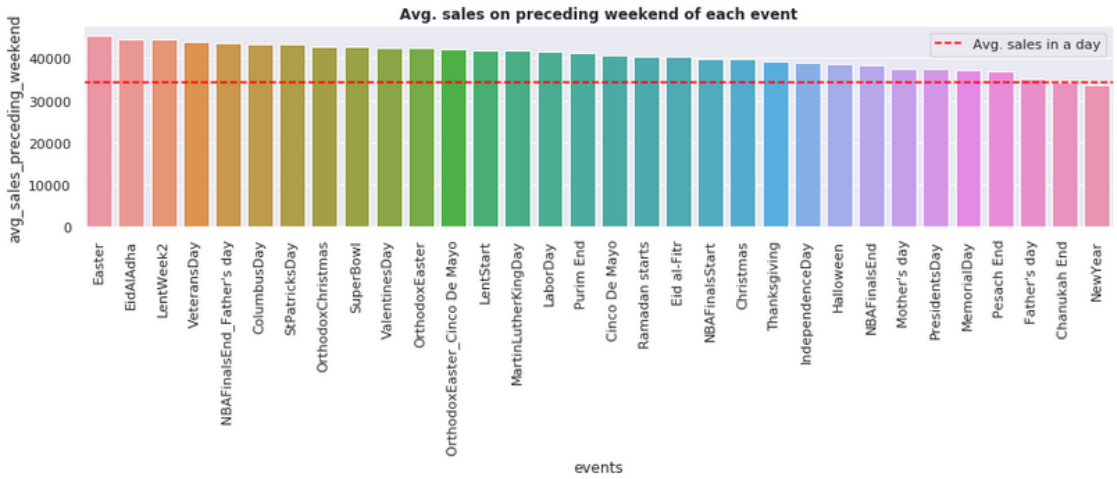


Figure 7: Bar chart of sales on events-preceding weekends

Finally, an analysis of price changes is conducted, using the distribution of median prices over the years grouping each item as either Cheap or Costly and looking at the changes in prices for each category and price group (Figure 8).

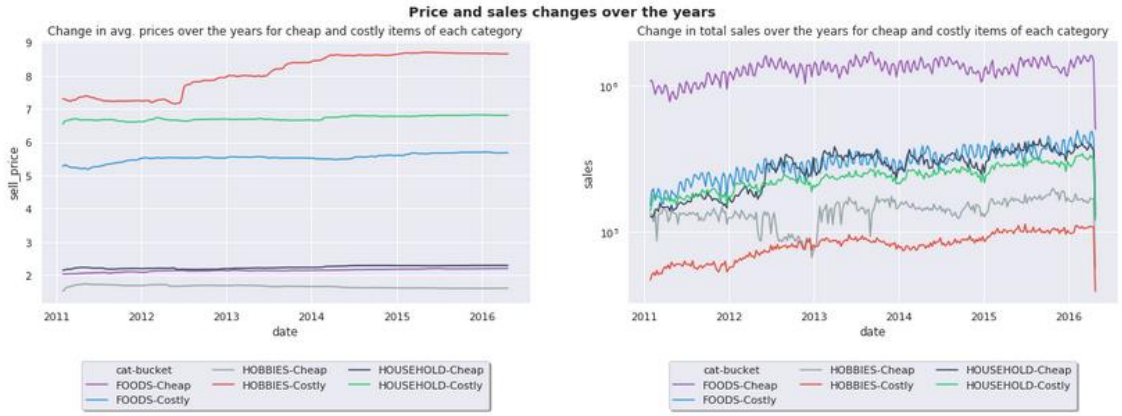


Figure 8: Price and sales changes over the years

3.3. M5 Forecasting - Starter Data Exploration

Next we are reviewing the "M5 Forecasting - Starter Data Exploration" notebook (Notebook by Rob Mulla, 2020) which had the goal of providing an overview of the competition for the competitors.

As previously, the notebook starts with an introduction to the competition, summarizing its objectives and then loads the necessary libraries before loading the datasets and visualizing some random items to get some initial quick looks into the time-series and with different time breakdowns (average sales per week day, month, year).

The analysis continues with some rollouts of inventory sale percentage per category (Figure 9), only taking into account the items that are being sold since many of the products were not sold at the beginning of the examined period. There are also some plots of rolling averages of total sales per store which examine any existing trends in time.

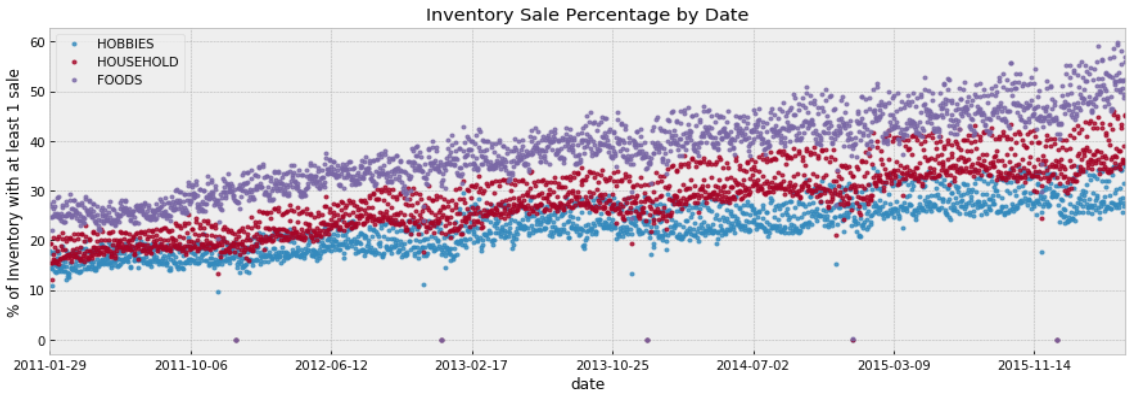


Figure 9: Scatter plot of inventory sale percentage by date

To conclude, there is the categorical variables analysis, using heatmaps to visualise the sales per category throughout the calendar years (Figure 10) and histograms to show the distribution of prices per category.

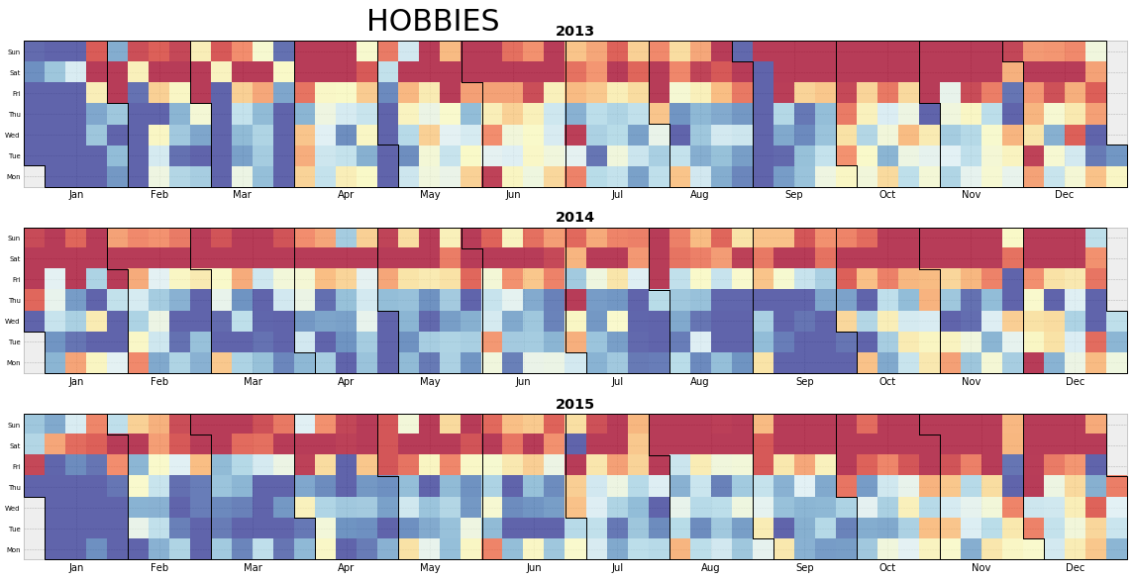


Figure 10: Heatmap calendar of sales

3.4. Time Series Forecasting-EDA, FE & Modelling

Our next notebook, "Time Series Forecasting-EDA, FE & Modelling 📈" (Notebook by Anshul Sharma, 2020), includes besides the EDA also the Feature engineering and Modelling sections.

This notebook, after loading the data, dives into downcasting the dataframes to reduce their use of storage and to expedite the operations performed on them. This is accomplished by changing the subtype for the numerical columns based on their min/max values and by means of a virtual mapping table for the categorical columns with low

cardinality (i.e., when the number of unique values is lower than 50% of the count of these values).

Melting the data is the next step, converting them from wide to long format by using some of the columns as id variables, as showcased in Figure 11 below. Then all the dataframes (sales, calendar and prices) are combined into a single complete dataframe with all the data required.

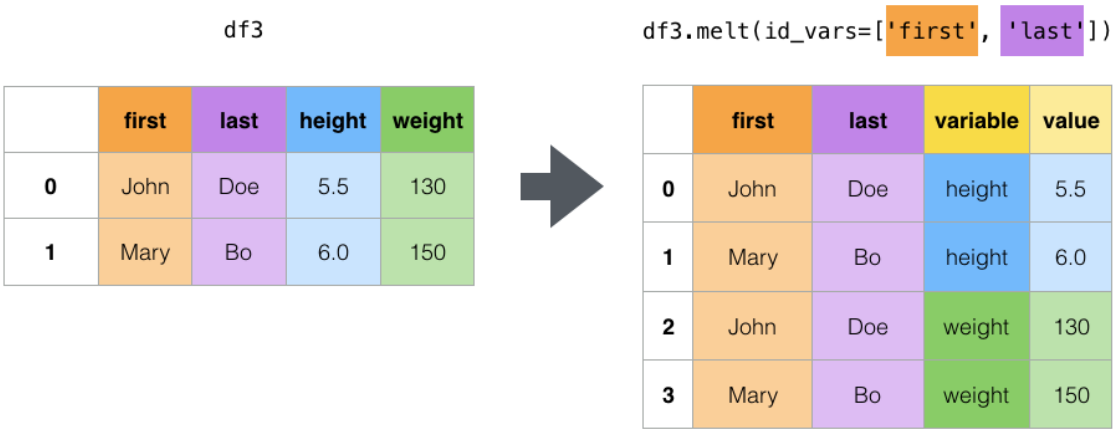


Figure 11: Example of melting a dataframe

Next, it proceeds with an extensive EDA section, starting with an interactive visualization which shows all items across different aggregation levels.

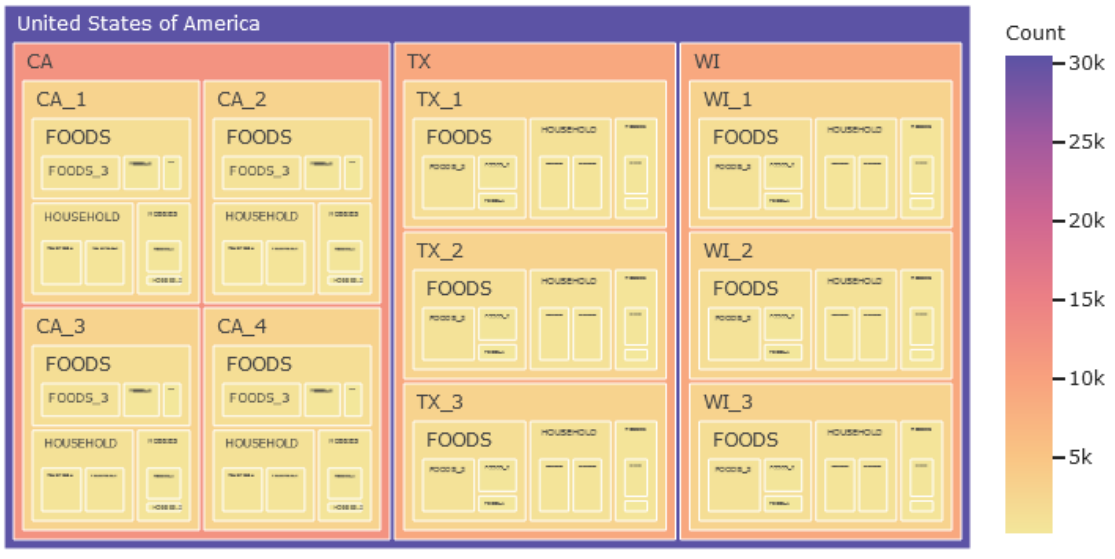


Figure 12: Interactive plot of all items

The analysis moves one with some violin plots (Figure 13) showing the distribution of prices and items sold across stores and categories. The state wide analysis follows, making use of some daily sales and daily revenue line charts – with separate plotting for

SNAP and non-SNAP days – and a daily sales heatmap; there are individual plots for all the stores across each of the three states.

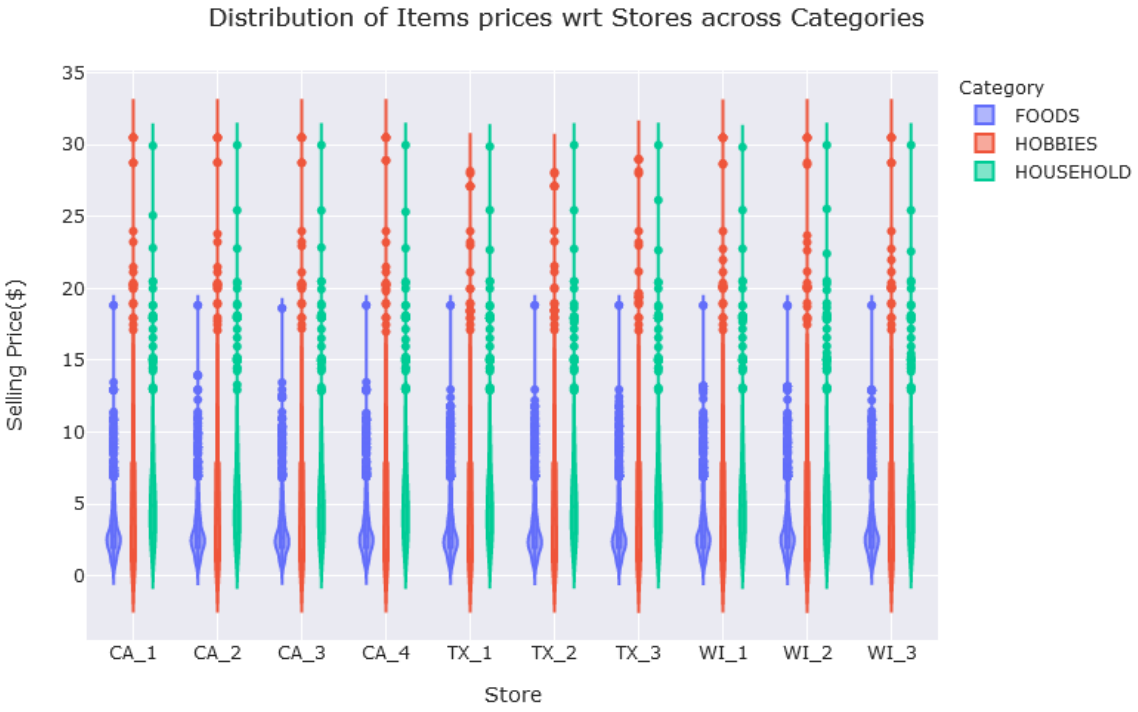


Figure 13: Violin plot of item prices across stores and categories

Next up is the Feature Engineering section, where data are re-framed as a supervised learning dataset before the modelling ensues. This section includes the following:

- Label encoding categorical variables to reduce RAM usage for further processing
- Introducing lag features to transform the time-series forecasting problem into a supervised learning problem
- Mean encoding on the basis of some selected features, which represents the probability of your target variable, conditional on each value of these features
- Calculation of weekly rolling average and expanding average of the items sold
- Creation of a selling trend feature which will be positive if the daily items sold are greater than the entire duration average and negative if they are not

The last section involves the modelling, making the predictions individually for each store and then combining all results into the submission file. Data are split into training, validation and testing subsets, a LightGBM regressor is used to construct the model which is then fitted with the training and validation subsets, and eventually used to generate the validation and testing predictions.

3.5. M5 Competition: EDA + Models

We continue with the review of the "M5 Competition : EDA + Models 📈" (Notebook by Tarun Paparaju, 2020), which contains both an EDA and Modelling.

The EDA section starts by plotting some sample randomly selected items, to get a quick look into the sales data, including some further zoomed in snippets. Next, wavelet denoising and average smoothing is applied to the daily sales, which granted loses some information but is useful in extracting certain features regarding the underlying trends in the time series. Figure 14 and Figure 15Figure 14 illustrate the denoised sales alongside the original sales, for some sample items.

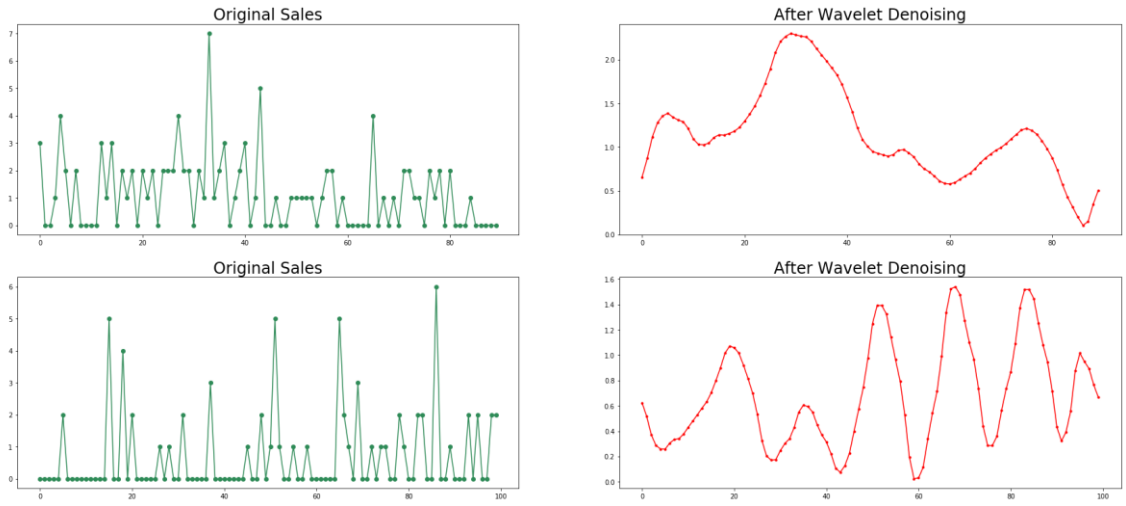


Figure 14: Original vs denoised sales (wavelet)

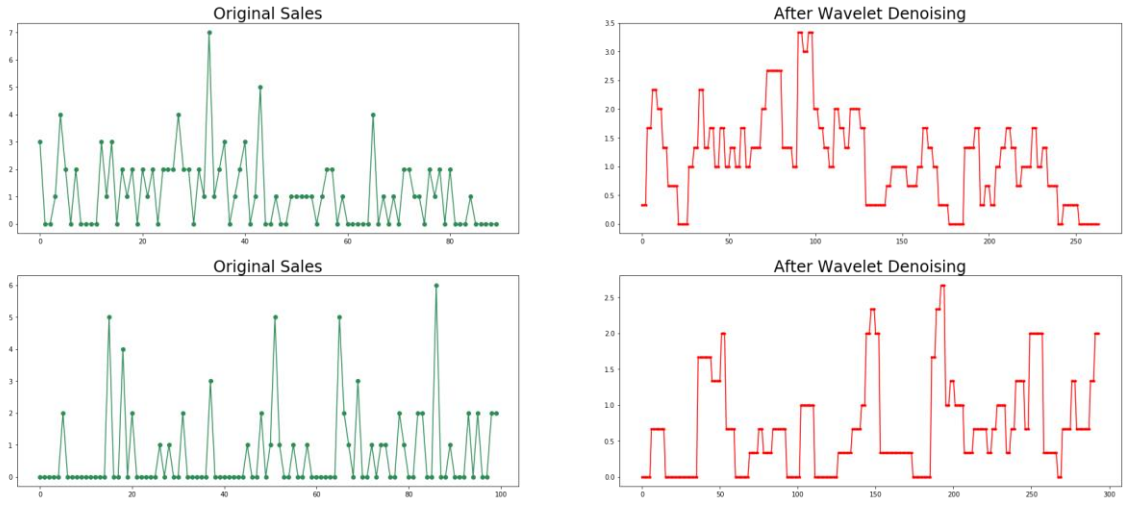


Figure 15: Original vs denoised sales (average smoothing)

Then there are some line charts, box plots and bar charts illustrating the mean and rolling average sales and prices across different states and stores.

In the Modelling section, after separating the dataset into the training and validation subsets, various forecasting methods are used, namely: naive approach, moving average, Holt linear, exponential smoothing, ARIMA, and Prophet. The efficiency of each method is illustrated in a line chart showing the sales over time, complete with the predicted value for a small date range, against the actual value for the same range (Figure 16).

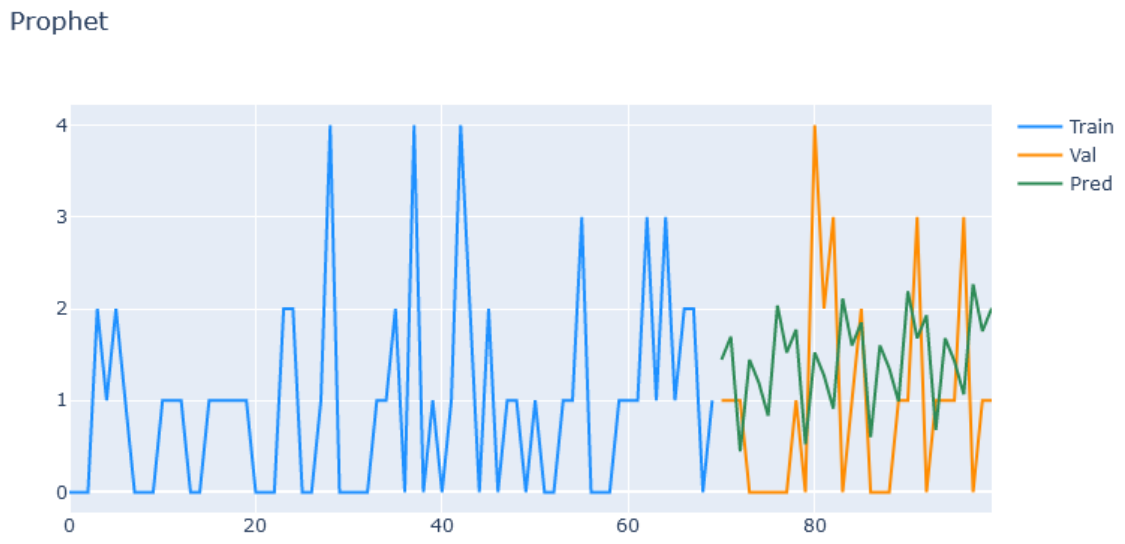


Figure 16: Prophet method performance

Ultimately, the loss from the predictions that each method generates is calculated and compared across the examined methods.

3.6. M5: EDA (Plotly) + LSTM Neural Network Vs. XGBoost

The last notebook we are going to examine is call "M5: EDA (Plotly) + LSTM Neural Network Vs. XGBoost" (Notebook by Jestelrod, 2020) and includes, as the title suggests, a feature selection section, followed by exploratory data analysis, and finally two different models for forecasting are examined – LSTM neural network and XGBoost.

After the necessary libraries and the dataset are loaded, some defining features are selected with the use of the `LabelEncoder` transformer and a custom function which applies a number of transformations to the dataset in order to reduce the memory usage and bring the data into an easier to handle format.

In the EDA section, some sample items are plotted first, followed by the analysis of sales per state using a choropleth, a line chart versus time, bar charts with several aggregation levels (day of week, month, year).

There are also some plots showing the rolling average over 28 days for some sample items (Figure 17).

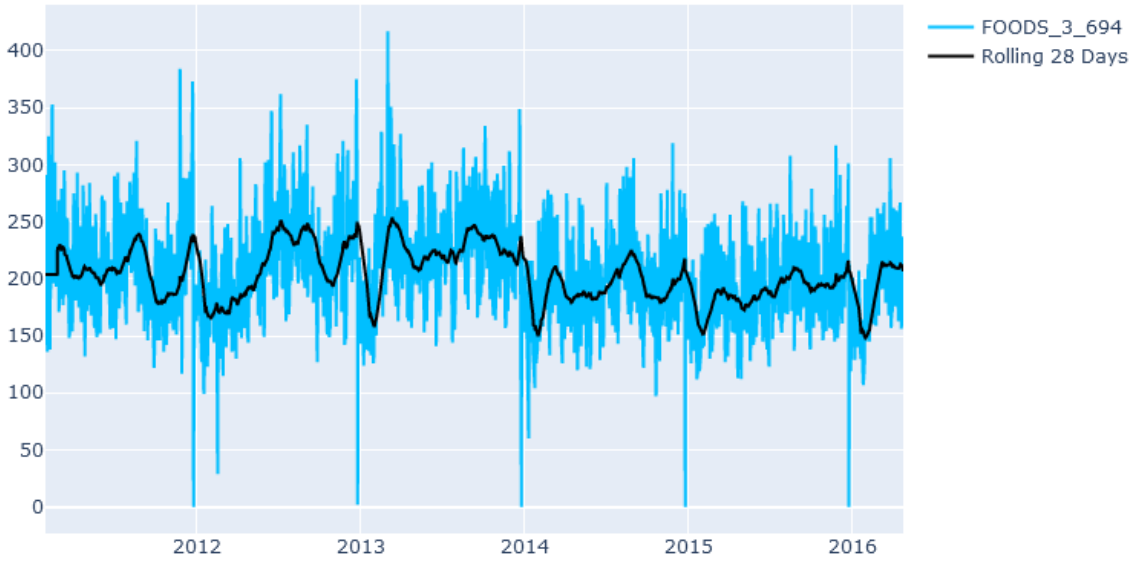


Figure 17: Rolling 28-days average

Finally, the two models are configured, trained and used to make the predictions. The predictions from each model are evaluated based on the RMSE and are plotted against each other and the actual data (Figure 18).

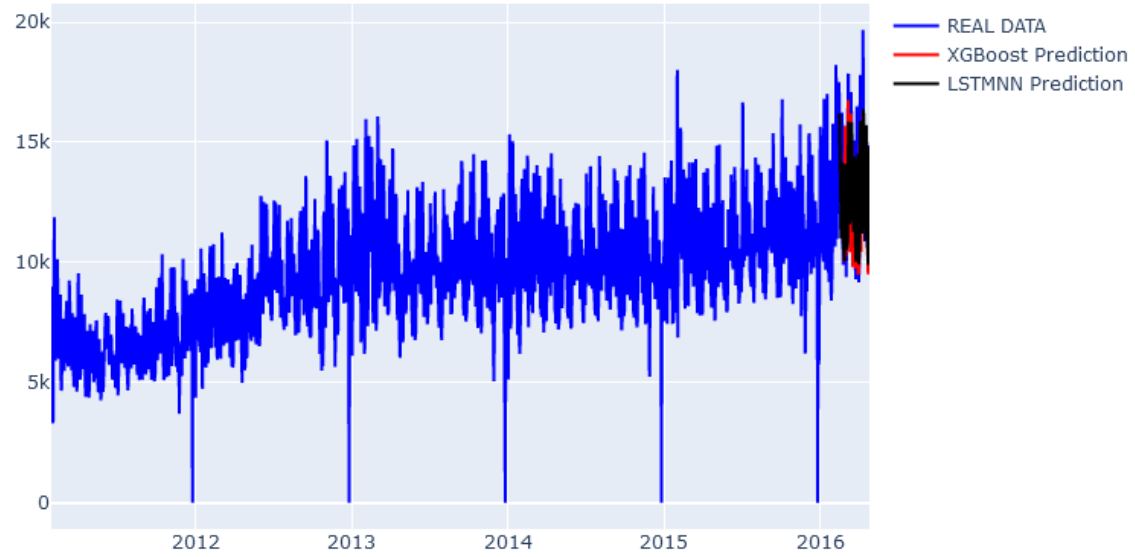


Figure 18: Predicted sales vs real

4. Exploratory Data Analysis

4.1. Introduction

John Tukey first coined the term exploratory data analysis, or EDA for short, in the early 1970s for describing the act of "looking at data to see what it seems to say" and stating that "it concentrates on simple arithmetic and easy-to-draw pictures" (Young, Faldowski, & McFarlane, 1993).

EDA is presently recognized as an essential process performed by data scientists to closely examine an unknown dataset, better understand its nature and characteristics, by utilizing various analysis operations (such as filtering, aggregation, and visualization) with the purpose of spotting patterns, detecting anomalies, checking assumptions, or testing some hypotheses and eventually extracting some preliminary insights from it. (Milo & Somech, 2020).

There are several types of EDA approaches, which can be roughly grouped into *graphic*, *quantitative*, *univariant* and *multivariant* analysis. Graphic approaches use diagrams and visual representations to summarize the data, quantitative approaches rely on depicting summary statistics, in multivariate approaches multiple variables are correlated simultaneously, while in a univariate approach a single variable is correlated simultaneously.

Regarding which approach to use, for any given dataset, there could be multiple "right answers" since different data analysts will typically have different views based on their specific goals, overall attitude and knowledge; the creative element is therefore recognized as a crucial component of EDA (Morgenthaler S. , 2009).

Some of the common exploration goals are as follows:

- Profiling: understanding what the data contain and assessing their quality
- Discovery: gaining new insights in the context of open-ended analyses

Several areas of the modern world of data analysis have been influenced by EDA, such as the areas of computational statistics, data visualization, data mining, and machine learning while many applied areas rich in data have also obtained an EDA flavour (Morgenthaler S. , 2009).

4.2. Typical EDA steps

The are some distinct steps for conducting EDA, according to Tufféry (2011), which are namely: (i) Distinguish Attributes, (ii) Univariate Analysis, (iii) Bi-/Multivariate Analysis, (iv) Detect Aberrant and Missing Values, (v) Detect Outliers, and (vi) Feature Engineering, as shown in Figure 19 (Ghosh, Nashaat, Miller, Quader, & Marston, 2018).

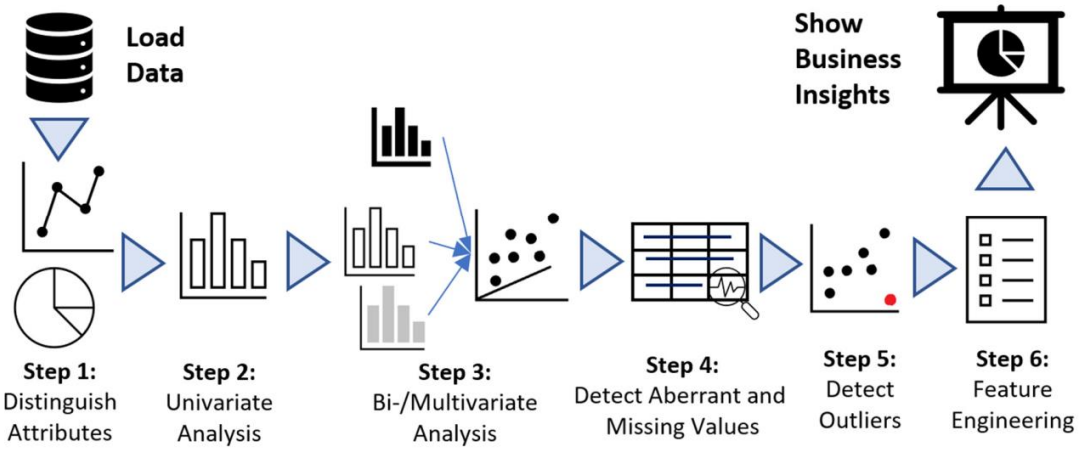


Figure 19: Fundamental EDA steps

These steps can be further grouped and described as follows (Rahmany, Mohd Zin, & Sundararajan, 2020; Tufféry, 2011; Ghosh, Nashaat, Miller, Quader, & Marston, 2018):

1. **Descriptive analytics**

   The dataset is first inspected to identify the attributes, i.e., whether the variables are numerical/qualitative, categorical/quantitative, ordinal, discrete or continuous).

   Once the attributes are identified, further univariate analysis is performed to detect descriptive statistics such as:

   - the shape (i.e., dimensions) which shows the number of attribute values,
   - the centrality (i.e., mean, median) which helps determine an approximate average for the attribute values, and,
   - the dispersion (i.e., range, variance, standard deviation, skewness, and kurtosis) which helps identify the spread of the values between its lowest and highest bound

   which allow for a deeper understanding of the dataset in whole.

2. **Relationships between attributes and insights**

   Insights into the relationships in the data can be derived by visualizing the dataset with plots such as histograms, heatmaps, map charts, line charts, pie charts, etc.;

these can help determine incompatibilities among attribute values and any linkages such as correlation or covariance which would generate optimal feature combinations for subsequent analysis.

Analysis can start with a pair of attributes (bivariate), and, based on the observed compatibility, continue with combining more than two attributes (multivariate) for a deeper exploration.

3. **Missing and null values**

Data are rarely clean and homogeneous usually due to inadequate extraction and/or collection. The missing and/or null values must be handled cautiously since they can result in a biased model with erroneous predictions or misclassifications.

Various approaches exist for handling missing values, selecting the most suitable depends on the nature of the data and the number of values that are missing:

- Drop the missing and null values; this approach deletes all observations where any of the attributes is missing, which reduces the sample size and hence the quality of our model which can add further bias into the analysis especially if the dataset has some special significance for the observations with missing values
- Use statistics to fill in the missing values; this approach replaces the missing values with a summary statistic like the mean or median, selected for the particular feature
- Utilize machine learning to predict the missing values; this approach replaces the missing values with a value predicted from either a regression or a classification model (depending on the class of the missing data)

4. **Outliers**

An outlier is an observation that deviates further away from other observations in the dataset. Outliers can be a result of a mistake during data collection or they can be just an indication of variance in the data and they can influence the analysis, adding bias, leading to misinterpretation of attribute properties and generating inaccurate conclusions, therefore they should also be handled with caution.

There are three types that outliers can be primarily categorized as: univariate, bivariate, and multivariate outliers. Univariate and bivariate outliers can be easily depicted using box-plots, interactive histograms and scatter plots while detecting multivariate outliers is more challenging and analysts need to inspect correlations among different attributes to find the suitable visualization.

Upon detection of the outliers, similarly to the missing/null values, they can be rectified by either removing the observations including them or performing statistical or prediction-driven imputations of the attributes.

Feature engineering, which is presented here as the final step of the EDA process, will be covered separately in this study, at the dedicated chapter 5.

4.3. Common plot types

A good practice when plotting data, regardless of the type of plot, is to have the response variable plotted on the vertical axis (y-axis), and the explanatory variable plotted on the horizontal axis (x-axis). If there are multiple explanatory variables then they can be represented by different colours, shapes, or facets on the plot (Cox, 2017; Juggins & Telford, 2012; Ribecca, n.d.).

- Bar charts (horizontal or vertical) are suitable for discrete numerical data and counts of nominal data and they are very efficient at comparing the frequency of different groups

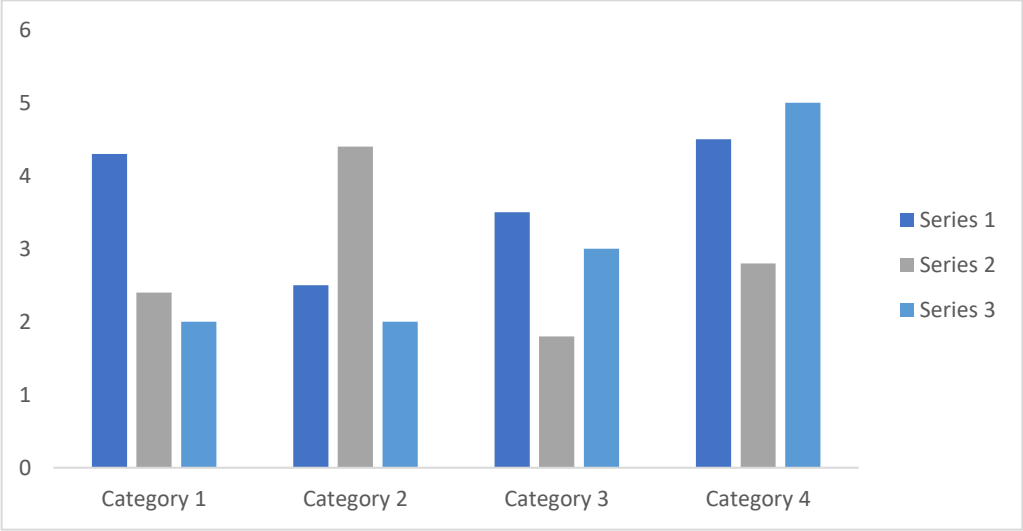


Figure 20: Bar chart example

- 100% Stacked bar graphs show the percentage-of-the-whole of each group and are plotted by the percentage of each value to the total amount in each group; they are useful in showing the relative differences between quantities in each group although they become harder to read the more segments each bar has

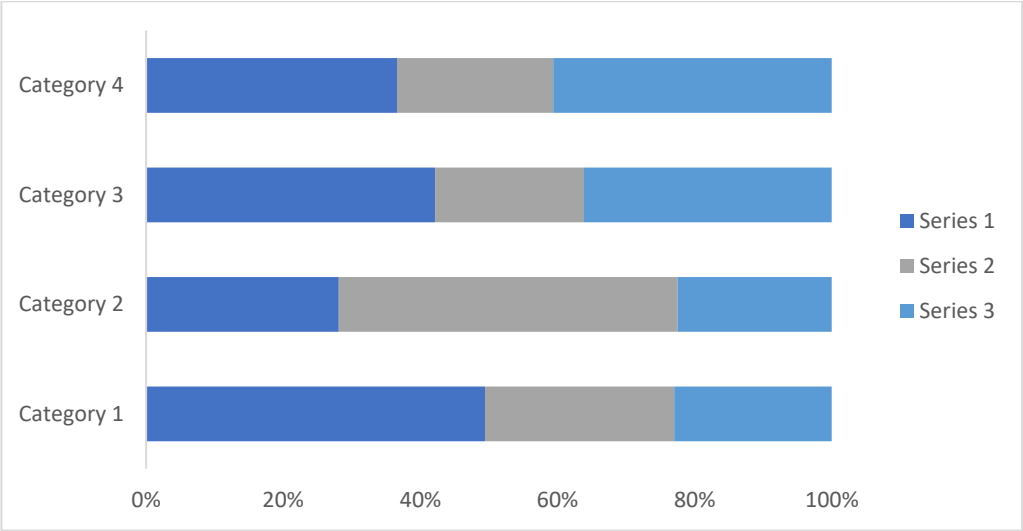


Figure 21: 100% stacked bar example

- Scatter plots are suitable for exploring relationships between two correlated variables, usually with the independent variable plotted on the x-axis and the depended plotted on the y-axis. They are handy for highlighting trends in the data such as positive (values increase together), negative (one value decreases as the other increases), null (no correlation), linear, exponential and U-shaped; The strength of the correlation can be determined by how closely packed the points are to each other on the graph.

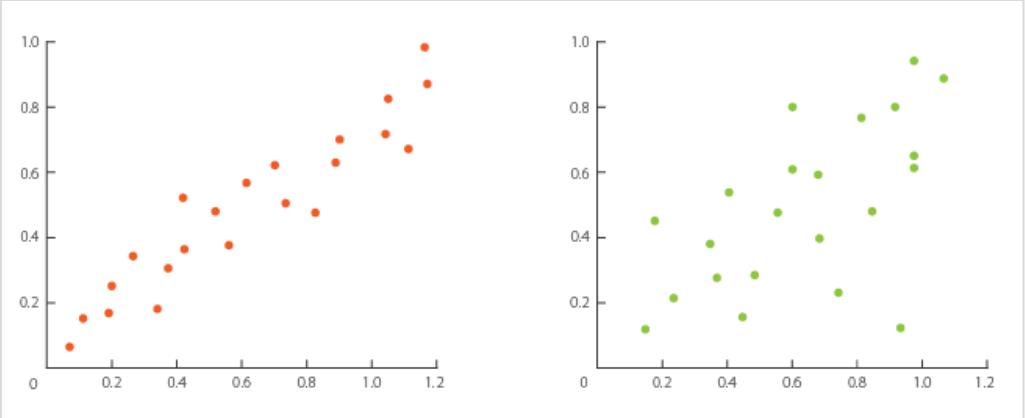


Figure 22: Scatter plot example

- Line charts, similarly to scatter plots, are suitable for showing continuous variables; they often have a time element across the x-axis and are useful in showcasing the progress of a variable over time. When grouped with other lines (other data series), individual lines can be compared to one another; best practice is to show no more than 3-4 lines per chart.

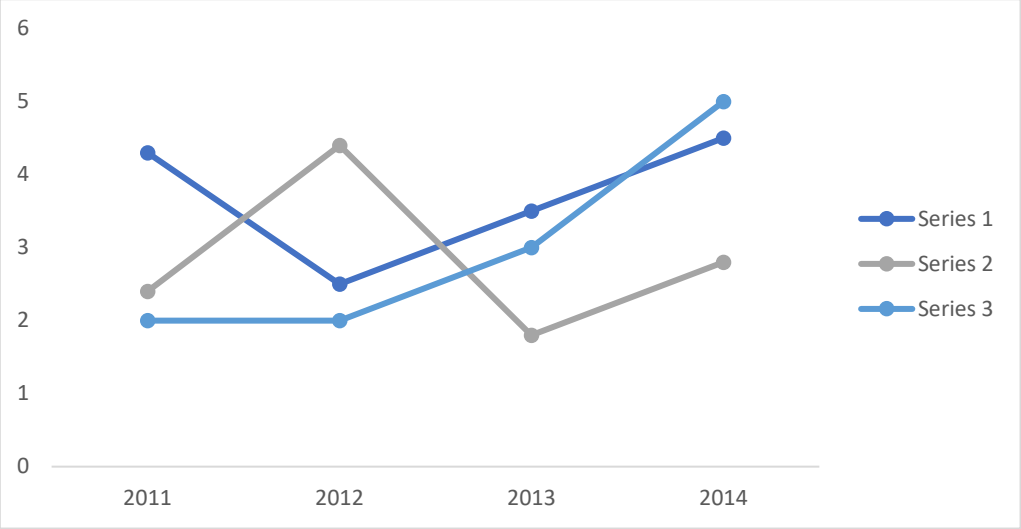


Figure 23: Line chart example

- Box Plots (also known as box and whiskers plots) are suitable for a continuous variable and a nominal variable; they are useful at showing condensed information and highlighting differences between nominal groups. Box plots contain the following information:
    - Median: the line within the box
    - 1st Quartile (Q1) and 3rd Quartile: the bottom line and top line of the box
    - Interquartile range (IQR): the length of the box itself
    - Range (minus statistical outliers): the length of the whiskers
    - Statistical outliers: any points outside the whiskers
    - The limits for the outliers are usually calculated as

      $Lower\ limit = Q1 - 1{,}5 \cdot IQR$ and $Upper\ limit = Q3 + 1{,}5 \cdot IQR$

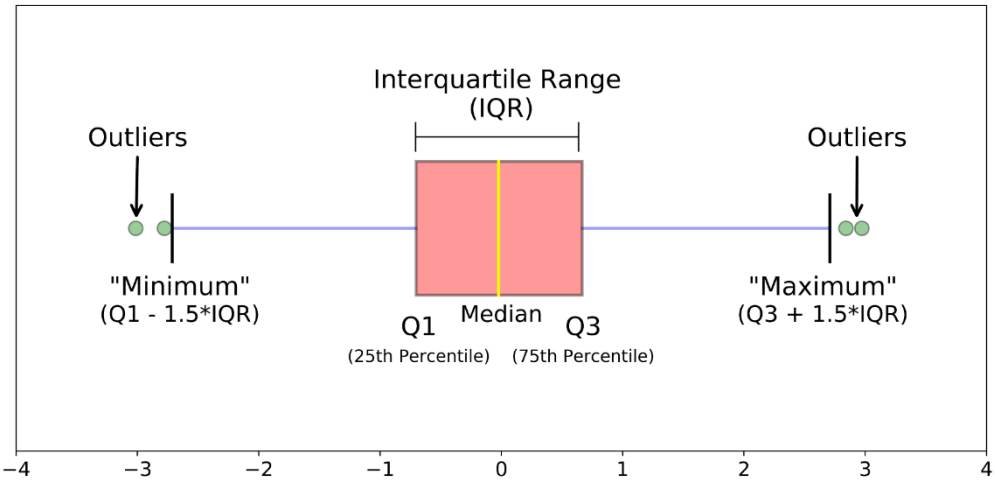


Figure 24: Box plot example

- Histograms are suitable for continuous data; they show the frequency counts in the form of rectangles (bins or classes) and can be useful for highlighting distributions. The number of bins needs some consideration, since too few and broad bins might result in obscuring relevant details, while too numerous and narrow bins might start to capture random fluctuations in the data.

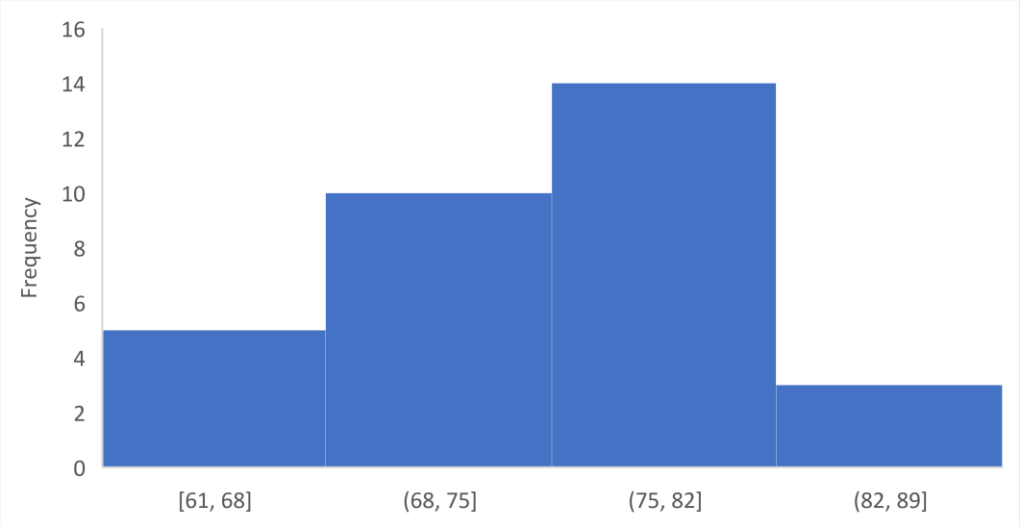


Figure 25: Histogram example

- Density plots are suitable for continuous data; they are essentially a smoothed version of a histogram and are used in a similar concept to study the distribution of a variable with the peaks indicating where the values are concentrated

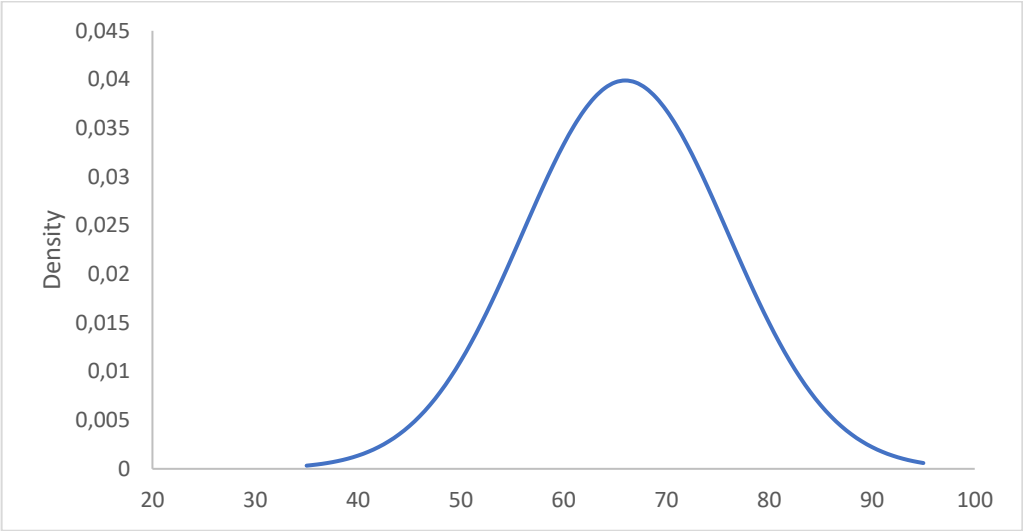


Figure 26: Density plot example

- Pie charts are suitable for presenting categorical data if there are few categories, with values of a similar magnitude, and if the emphasis is on representing proportional, rather than absolute differences between categories

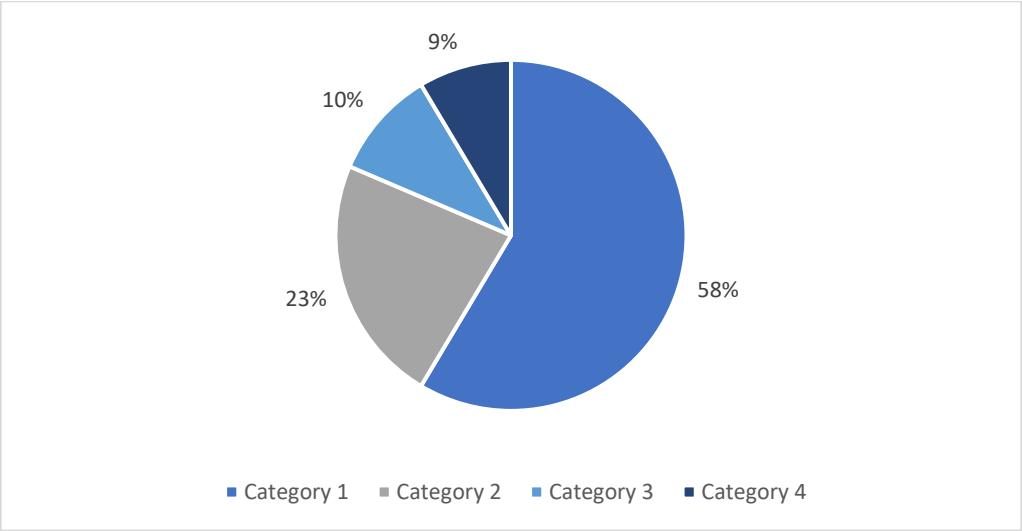


Figure 27: Pie chart example

- Choropleth maps display divided geographical areas or regions that are coloured, shaded or patterned with the data variable represented in the colour progression (typically from one colour to another, a single hue progression, transparent to opaque, light to dark or an entire colour spectrum); they are useful to investigate variation or patterns across the displayed location with the downside of emphasizing larger regions more than smaller ones, which can be avoided by using normalized values

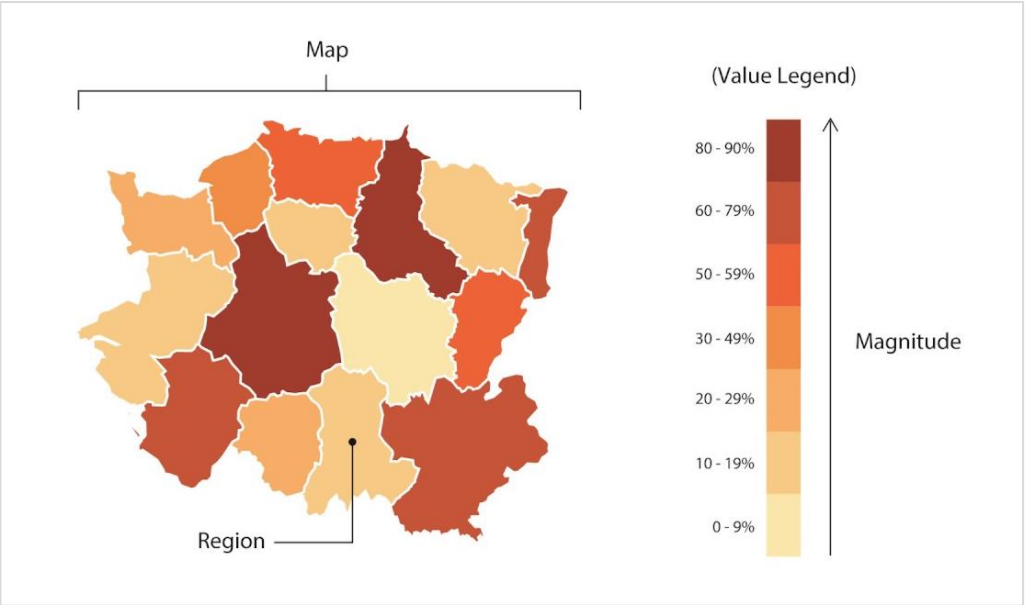


Figure 28: Choropleth map example

4.4. Tools for EDA

There is a large number of tools and techniques for performing EDA, with different functionalities to assist both with the identification of hidden patterns and correlations

among attributes, but also with the formulation of hypotheses from the data and their validation (Ghosh, Nashaat, Miller, Quader, & Marston, 2018).

Python and R are the two most widely used programming languages for EDA, due to being oriented at data preparation, exploration and visualization.

Python (used in the thesis) is an open-source general purpose tool with applications across a variety of industries and programming disciplines (software development, education, academia, health diagnostics etc). Python capabilities are extended through its robust collection of packages, with its official package repository exceeding the 100,000. (Brittain, Cendon, Nizzi, & Pleis, 2018)

Some common packages used for data analysis with Python are: Pandas (ideal for data manipulation), statsmodels (for modelling and testing), scikit-learn (for classification and machine learning tasks and especially favoured for data mining), NumPy (for numerical operations) and SciPy (for common scientific tasks).

Python also provides an extensive list of Integrated Development Environments (IDE) such as Spyder, a cross-platform IDE distributed through Anaconda (an open-source distribution for large-scale data) and Jupyter Notebook, an IDE which allows end-users to integrate live code, equations, computational output, visualizations, and other multimedia resources, along with explanatory text in a single sharable document.

R is an open-source programming language and free software environment for statistical computing and graphics supported by the R foundation for statistical computing. The R language is widely used among statisticians in developing statistical observations and data analysis.

While it should be noted that both are good options for EDA and share a lot of similarities, below are a few points that highlight how they differ (Paruchuri, 2020):

- R is more functional and has more functionality built-in while Python is more object-oriented and relies heavily on packages
- Python has packages for related tasks that offer a consistent API and are well-maintained while R has a larger ecosystem of small packages, with a great diversity but also greater fragmentation and less consistency
- R has more statistical support in general, since it was built as a statistical language, while it is more straightforward to do non-statistical tasks in Python and is a better choice when there is need for combination with other kinds of programming tasks

Furthermore, there are numerous visual interactive data exploration tools, that can assist the users with each step of the analysis process with limited need for programming skills or analytical knowledge. There exist commercial tools, used across industries (e.g., Tableau, Domo, MS Power BI and QlikView) and tools proposed by academic researchers (e.g., DataVoyager, Keshif, Domino, SketchStory and ForeSight).

4.5. Data inspection

Our dataset consists of the five files described below:

- `sales_train_validation.csv` contains the sales of daily units for the validation period which was from January 29th, 2011 until May 22nd, 2016 (i.e., `d_1 - d_1913`)
- `sales_train_evaluation.csv` contains the sales of daily units for the evaluation period which was from January 29th, 2011 until June 19th, 2016 (i.e., `d_1 - d_1941`)
- `calendar.csv` contains information such as the day-of-the-week, date, events and holidays and whether or not SNAP food stamps were allowed on a specific date
- `sell_prices.csv` contains information about the price of the products sold per store and date (in the form of the weekly average per item)
- `submission.csv` demonstrates the correct format for submission to the competition

In order to inspect the dataset at the outset, we generated some initial statistics and previewed the top and bottom rows for each dataframe to gain some knowledge on the included attributes, types of data, values distribution etc. For this purpose, we used the following Python recourses:

- Loaded the data by using the `pandas.read_csv` function which reads a comma-separated values file (csv) into a dataframe, provided the file path/location
- Used the `DataFrame.shape` function which returns a tuple representing the dimensionality of the dataframe in question (i.e., its columns and rows)
- Used the `pandas.DataFrame.head` function which returns the first n rows of the dataframe; the default value for n is 5 while for negative values of n, this function returns all rows except the last n rows

- Used the `pandas.DataFrame.tail` function which returns the last n rows of the dataframe; the default value for n is 5 while for negative values of n, this function returns all rows except the first n rows
- Used the `pandas.DataFrame.info` method which prints a concise summary of a DataFrame, including the index data type and columns, non-null values and memory usage
- Used the `pandas.DataFrame.describe` function which generates descriptive statistics for both numeric and object data, as well as columns with mixed data types
    - for numeric data, the results will include count, mean, std, min, max as well as percentiles (by default 25%, 50% and 75%)
    - for object data (e.g., strings or timestamps), the results will include count, unique, top (most common value), freq (the most common value's frequency) and first-last items (for timestamps)
    - for mixed data types, the default is to return only an analysis of numeric columns unless include='all' is provided as an option so that the result includes a union of attributes of each type
- Used the `pandas.DataFrame.dtypes` function which returns a series with the data type of each column, e.g., float64, int64, datetime64[ns], object, etc. (columns with mixed types are marked with the object dtype)

The observations and findings from the application of the abovementioned steps on the dataframes are summarized below.

The two **sales dataframes** that contain the daily units sold (i.e., quantity and not revenue), are saved as `df_sales` and `df_sales_eval` with the following characteristics:

- Every row corresponds to one item in one of the stores which gives us 30,490 rows for all combinations of 3,049 items and 10 stores
- There is one column of daily units sold for each day from 2011-01-29 until 2016-05-22 in the `df_sales` dataframe and until 2016-06-19 in the `df_sales_eval` dataframe
- There are additional columns for the item ID, the department and category it belongs to, the store it is sold in, and the state where the store is located
- Each item/row bears an ID generated from a combination of its attributes and the dataframes it is located in (validation or evaluation):

`CategoryName_DepartmentNumber_ItemNumber_StateName_StoreNumber_validation/evaluation`

- The columns corresponding to each date start with the prefix `d_` followed by an increasing number (from 1 until 1,913 or 1,941 depending on the timeframe)
- Many items have several days with zero sales (i.e., high amount of zero values noticed for the date columns): we can, e.g., see in Table 6 that at least 75% of the items in the first 10 days have 0 sales

Table 5: Sales dataframe (top rows)

	id	item_id	dept_id	cat_id	store_id	state_id	d_1	d_2	d_3	d_4	...
0	HOBBIES_1_001_CA_1_validation	HOBBIES_1_001	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...
1	HOBBIES_1_002_CA_1_validation	HOBBIES_1_002	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...
2	HOBBIES_1_003_CA_1_validation	HOBBIES_1_003	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...
3	HOBBIES_1_004_CA_1_validation	HOBBIES_1_004	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...
4	HOBBIES_1_005_CA_1_validation	HOBBIES_1_005	HOBBIES_1	HOBBIES	CA_1	CA	0	0	0	0	...

Table 6: Sales dataframe (describe table)

	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_10	...
count	30490.000000	30490.000000	30490.000000	30490.000000	30490.000000	30490.000000	30490.000000	30490.000000	30490.000000	30490.000000	...
mean	1.070220	1.041292	0.780026	0.833454	0.627944	0.958052	0.918662	1.244080	1.073663	0.838701	...
std	5.126689	5.365468	3.667454	4.415141	3.379344	4.785947	5.059495	6.617729	5.917204	4.206199	...
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...
max	360.000000	436.000000	207.000000	323.000000	296.000000	314.000000	316.000000	370.000000	385.000000	353.000000	...

A detailed description of the columns observed in Table 5:

- *id*: The id of each row which includes the department id and the store id and is unique throughout the dataset (there is also an identifier for validation/evaluation depending on which of the two dataframes is being examined)
- *item_id*: The id of the product which includes the department id and is unique per store id (same item id is used across stores)
- *dept_id*: The id of the department the product belongs to (several departments within each category)
- *cat_id*: The id of the category the product belongs to (several categories within each store)
- *store_id*: The id of the store where the product is sold (several stores within each state)

- *state_id*: The id of the state where the store is located
- *d_i*: The number of units (quantity) sold at day *i*, counting from day 1 (i.e., 2011-01-29) until day 1,913 (i.e., 2016-05-22) or day 1,941 (2016-06-19) depending on which of the two dataframes (validation or evaluation) is being examined

The **sell prices dataframe**, which contains information about the price of each item per store and per date, is saved as `df_prices` with the following characteristics:

- There are 6,841,121 distinct products across stores, as seen in Table 7
- The sell price of each item is provided as a weekly average across seven days for the specific product in the specific store; if the sell price is not available, this means that the product was not sold at all during the corresponding week in that store
- Although prices are constant on a weekly basis, they may change through time (from week to week) and across stores
- Prices range starting from as low as $0.01 to as high as $107.32 (US dollars)
- Most of the items (75% of them) are sold at an average of $6
- We can interlink the `df_prices` and `df_sales` dataframes using the `store_id` and `item_id` fields

Table 7: Sell prices dataframe (top rows)

	store_id	item_id	wm_yr_wk	sell_price
0	CA_1	HOBBIES_1_001	11325	9.58
1	CA_1	HOBBIES_1_001	11326	9.58
2	CA_1	HOBBIES_1_001	11327	8.26
3	CA_1	HOBBIES_1_001	11328	8.26
4	CA_1	HOBBIES_1_001	11329	8.26

Table 8: Sell prices dataframe (describe table)

	wm_yr_wk	sell_price
count	6841121.00	6841121.00
mean	11382.94	4.41
std	148.61	3.41
min	11101.00	0.01
25%	11247.00	2.18
50%	11411.00	3.47
75%	11517.00	5.84
max	11621.00	107.32

A detailed description of the columns observed in Table 7:

- *store_id*: The id of the store where the product is sold (same as the one included in the sales dataframe)
- *item_id*: The id of the product (same as the one included in the sales dataframe)
- *wm_yr_wk*: The id of the week the selling price is provided for, which consists of the number 1 followed by the year (2 digits) and the week (2 digits)
- *sell_price*: The price of the product for the given week/store (two decimals provided)

The **calendar dataframe**, which contains information on the dates in scope, is saved as `df_calendar` with the following characteristics:

- It includes related features like day-of-the-week, month, year, and binary flags (1 or 0) for whether the stores in each state allowed purchases with SNAP food stamps at this date (1) or not (0)
- The dates are provided in the yyyy/dd/mm format
- The `wm_yr_wk` field can be used to interlink `df_calendar` to `df_prices` and `df_sales` dataframes
- We have information on 1,969 days spanning from 2011-01-29 until 2016-06-19, which covers the training (days 1-1,913), validation (days 1-1,941) and evaluation (days 1,942-1,969) periods
- New weeks start on Saturday and there are 282 Saturdays in the data (i.e., there are data for 282 weeks overall)
- There are overall 162 days that have at least one event
- There are 4 distinct types of events
- "Religious" is the most common event type occurring 55 times
- Only 5 days have a second event (e.g., both Easter and Orthodox Easter occurring on the same day)
- There are overall 30 distinct event names

Table 9: Calendar dataframe (top rows)

	date	wm_yr_wk	weekday	wday	month	year	d	event_name_1	event_type_1	event_name_2	event_type_2	snap_CA	snap_TX	snap_WI
0	2011-01-29	11101	Saturday	1	1	2011	d_1	NaN	NaN	NaN	NaN	0	0	0
1	2011-01-30	11101	Sunday	2	1	2011	d_2	NaN	NaN	NaN	NaN	0	0	0
2	2011-01-31	11101	Monday	3	1	2011	d_3	NaN	NaN	NaN	NaN	0	0	0
3	2011-02-01	11101	Tuesday	4	2	2011	d_4	NaN	NaN	NaN	NaN	1	1	0
4	2011-02-02	11101	Wednesday	5	2	2011	d_5	NaN	NaN	NaN	NaN	1	0	1

Table 10: Calendar dataframe (describe table)

	date	wm_yr_wk	weekday	wday	month	year	d	event_name_1	event_type_1	event_name_2	event_type_2	snap_CA	snap_TX	snap_WI
count	1969	1969.000000	1969	1969.000000	1969.000000	1969.000000	1969	162	162	5	5	1969.000000	1969.000000	1969.000000
unique	1969	NaN	7	NaN	NaN	NaN	1969	30	4	4	2	NaN	NaN	NaN
top	2011-01-29	NaN	Saturday	NaN	NaN	NaN	d_1	SuperBowl	Religious	Father's day	Cultural	NaN	NaN	NaN
freq	1	NaN	282	NaN	NaN	NaN	1	6	55	2	4	NaN	NaN	NaN
mean	NaN	11347.086338	NaN	3.997461	6.325546	2013.288471	NaN	NaN	NaN	NaN	NaN	0.330117	0.330117	0.330117
std	NaN	155.277043	NaN	2.001141	3.416864	1.580198	NaN	NaN	NaN	NaN	NaN	0.470374	0.470374	0.470374
min	NaN	11101.000000	NaN	1.000000	1.000000	2011.000000	NaN	NaN	NaN	NaN	NaN	0.000000	0.000000	0.000000
25%	NaN	11219.000000	NaN	2.000000	3.000000	2012.000000	NaN	NaN	NaN	NaN	NaN	0.000000	0.000000	0.000000
50%	NaN	11337.000000	NaN	4.000000	6.000000	2013.000000	NaN	NaN	NaN	NaN	NaN	0.000000	0.000000	0.000000
75%	NaN	11502.000000	NaN	6.000000	9.000000	2015.000000	NaN	NaN	NaN	NaN	NaN	1.000000	1.000000	1.000000
max	NaN	11621.000000	NaN	7.000000	12.000000	2016.000000	NaN	NaN	NaN	NaN	NaN	1.000000	1.000000	1.000000

A detailed description of the columns observed in Table 10:

- *wm_yr_wk*: The id of the week the date belongs to (same as the one included in the sell prices dataframe)
- *weekday*: The nominal description of the day (Saturday, Sunday, Monday, etc.)
- *wday*: The numeric id of the weekday (1 for Saturday, 2 for Sunday, etc.)
- *month*: The month of the date, as a 1- or 2-digit number (1 for January, 2 for February, etc.)
- *year*: The year of the date represented with 4 digits
- *event_name_1*: The name of the event, if the date includes an event (otherwise null)
- *event_type_1*: The type of the event, if the date includes an event (otherwise empty)
- *event_name_2*: If the date includes a second event, the name of this event (otherwise empty)
- *event_type_2*: If the date includes a second event, the type of this event (otherwise empty)
- *snap_CA*, *snap_TX*, and *snap_WI*: A binary variable (0 or 1) indicating whether all stores in the state (CA, TX or WI) allow SNAP purchases on the examined date (1 indicates that SNAP purchases are allowed)

4.6. Handling missing and null values

Already from the data inspection we discovered that in all dataframes there were either missing and/or null therefore we conducted dedicated analysis for the missing and null values. Some of the Python recourses we used were:

- The `pandas.DataFrame.isna` function, which detects missing values by returning a Boolean object (True/False) indicating if the values are NA (such as

None or numpy.NaN); characters such as empty strings or numpy.inf are not considered NA values

- The seaborn.distplot function, which allows to flexibly plot a univariate distribution of observations

From the missing values analysis for the columns with nominal data (Figure 29) we saw that neither `df_sales` nor `df_prices` dataframes have any missing values, while `df_calendar` dataframe has 7,542 missing values.

```
# Count and print the missing values in each dataframe
print(df_sales.isna().sum().sum())
print(df_prices.isna().sum().sum())
print(df_calendar.isna().sum().sum())
```

```
0
0
7542
```

Figure 29: Missing values per dataframe

Therefore, we further analysed the missing values in the `df_calendar` dataframe per column and as we can see in Figure 30, out of the 1,969 distinct days that are included in the dataframe:

- Only 162 days (8.2% of all days) include an event while 1,807 days have no value at the event field at all
- Only 5 days have a second event which is only 0.2% of all days

```
# Check which columns include the missing values in the df_calendar dataframe
print(df_calendar.isna().sum())
```

```
date               0
wm_yr_wk           0
weekday            0
wday               0
month              0
year               0
d                  0
event_name_1    1807
event_type_1    1807
event_name_2    1964
event_type_2    1964
snap_CA            0
snap_TX            0
snap_WI            0
dtype: int64
```

Figure 30: Missing values per column in calendar dataframe

We continued with the zero values analysis for the columns with numerical data which showed us (Figure 31) that the `df_sales` and `df_calendar` dataframes have several columns with zero values while the `df_prices` dataframe has no zero values at all, which means that all items had some specified non-zero price for every reported day.

```
# Count zero values for each dataframe
zeros_sales = ((df_sales == 0).sum(axis=0)/len(df_sales.index)).to_frame('%zeros')
zeros_prices = ((df_prices == 0).sum(axis=0)/len(df_prices)).to_frame('%zeros')
zeros_calendar = ((df_calendar == 0).sum(axis=0)/len(df_calendar)).to_frame('%zeros')
```

```
print(zeros_sales)
```

	%zeros
id	0.000000
item_id	0.000000
dept_id	0.000000
cat_id	0.000000
store_id	0.000000
...	...
d_1909	0.590849
d_1910	0.597704
d_1911	0.561069
d_1912	0.515513
d_1913	0.511742

```
print(zeros_prices)
```

	%zeros
store_id	0.0
item_id	0.0
wm_yr_wk	0.0
sell_price	0.0

```
print(zeros_calendar)
```

	%zeros
date	0.000000
wm_yr_wk	0.000000
weekday	0.000000
wday	0.000000
month	0.000000
year	0.000000
d	0.000000
event_name_1	0.000000
event_type_1	0.000000
event_name_2	0.000000
event_type_2	0.000000
snap_CA	0.669883
snap_TX	0.669883
snap_WI	0.669883

Figure 31: Zero values per dataframe

Next, we generated some density plots for `df_sales` data and observed the following:

- Density plot of zero items sold per day (Figure 32): We can see that on most days between 60-80% of the items were not sold at all

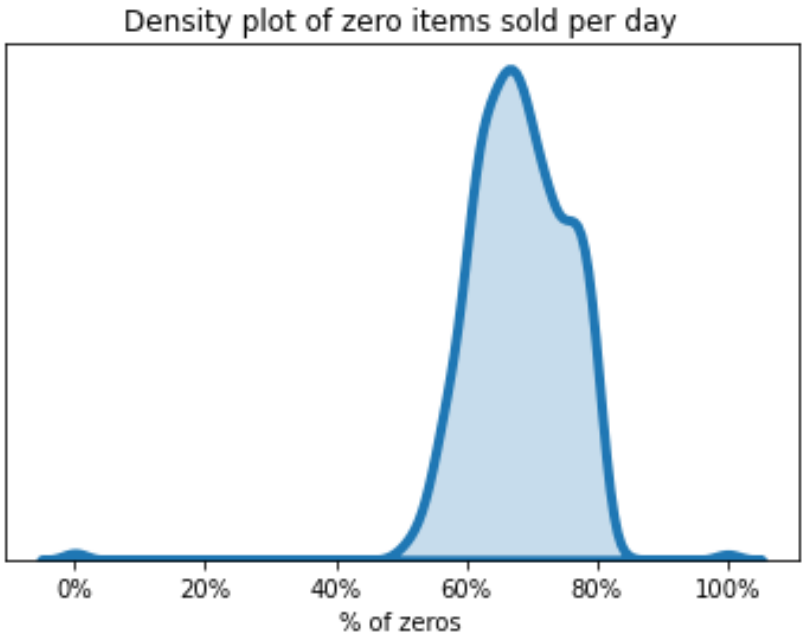


Figure 32: Density plot of zero items sold per day

- Density plot of zero sales day per item (Figure 33): We see that few items are sold for more than half of the days - the position of the peak shows that most items do not have a sale for about 90% of the days

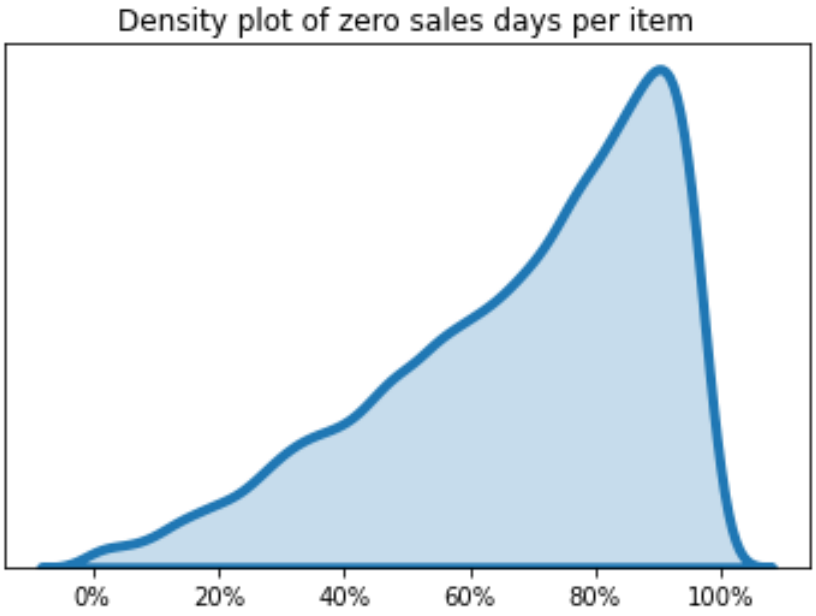


Figure 33: Density plot of zero sales days per item

Upon further analysis of the `df_calendar` dataframe we understand that each state has 650 SNAP days since there are 67% zero values in each snap_state column, as seen in Table 11, which means that 1,319 days in each state are non-SNAP days.

Table 11: Percentage of zeros per column in calendar dataframe

```
print(zeros_calendar)
```

	%zeros
date	0.000000
wm_yr_wk	0.000000
weekday	0.000000
wday	0.000000
month	0.000000
year	0.000000
d	0.000000
event_name_1	0.000000
event_type_1	0.000000
event_name_2	0.000000
event_type_2	0.000000
snap_CA	0.669883
snap_TX	0.669883
snap_WI	0.669883

To conclude, we defined a helper function that would allow us to plot the share of zero values in each of the six years alongside the active vs inactive pie (Figure 34); inactive products are considered those that had only zero sales for all days in the examined year.

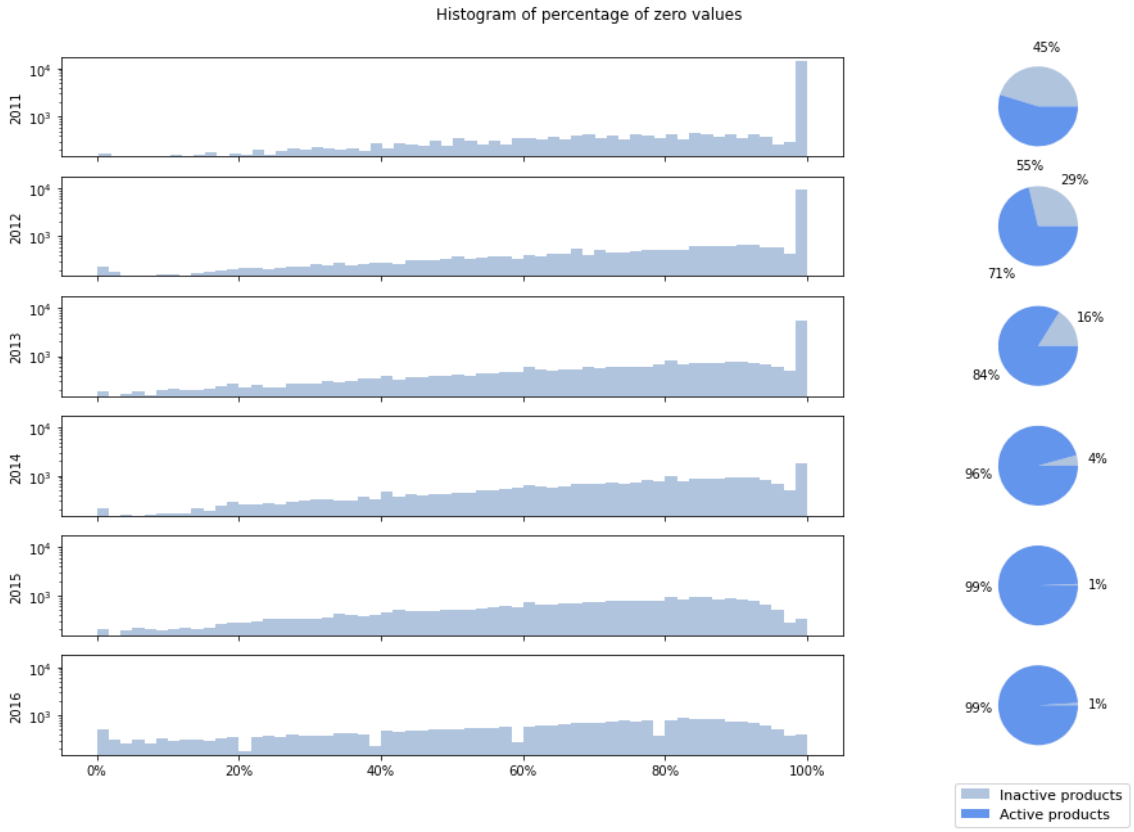


Figure 34: Percentage of zero values per year

We can see that the percentage of inactive products is higher in the first examined years and decreases in time. This can be explained by the fact that some of the products found in later years would not have had any sales in the first years since they were probably newer products introduced later to the stores. Notably, almost half of the products appear inactive in the first examined year while almost all of the products are active in the last examined year.

4.7. Time series analysis

There are several Python libraries for analysing time series and using them for forecasting purposes. Below we are talking a closer look into the defining attributes of some of them (Brownlee, Introduction to Time Series Forecasting with Python, 2020) which we have used in this thesis.

NumPy is a library that provides extended functionalities to Python and provides a user-friendly ambiance. It allows efficient operations on homogeneous data stored in specially designed arrays called NumPy arrays. It also helps manipulate numerical data.

SciPy is a library that contains a variety of sub-packages and has a collection of scientific functions, including clustering, image processing, integration, differentiation, gradient optimization, etc. The reason it is preferred over other tools is its speed. All the numerical computing in Python is done via SciPy.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Pandas is a library that provides high-performance tools for loading and handling data in Python. It is built upon and requires the SciPy ecosystem but provides convenient and easy to use data structures like DataFrame and Series for representing the data. Key features relevant for time series forecasting in Pandas include:

- The Series object for representing a univariate time series
- Explicit handling of date-time indexes in data and date-time ranges
- Transforms such as shifting, lagging, and filling
- Resampling methods such as up-sampling, down-sampling, and aggregation

Statsmodels is a library that provides tools for statistical modelling, as well as tools dedicated to time series analysis that can also be used for forecasting. It is built upon and requires the SciPy ecosystem and supports data in the form of NumPy arrays and Pandas Series objects. Key features relevant for time series forecasting in Statsmodels include:

- Statistical tests for stationarity such as the Augmented Dickey-Fuller unit root test
- Time series analysis plots such as autocorrelation function (ACF) and partial autocorrelation function (PACF)
- Linear time series models such as autoregression (AR), moving average (MA), autoregressive moving average (ARMA), and autoregressive integrated moving average (ARIMA)

Scikit-learn is a library used for developing and practicing machine learning in Python, with a focus on machine learning algorithms for classification, regression, clustering, etc. It also provides tools for related tasks such as evaluating models, tuning parameters, and pre-processing data. Key features relevant for time series forecasting in scikit-learn include:

- The suite of data preparation tools, such as scaling and imputing data
- The suite of machine learning algorithms that could be used to model data and make predictions
- The resampling methods for estimating the performance of a model on new data

4.7.1. Decomposition components

Any given time-series is characterized by some systematic and non-systematic components (Hyndman & Athanasopoulos, 2018; Brownlee, Introduction to Time Series Forecasting with Python, 2020):

- Systematic components of the time series are those that have consistency or recurrence and can be described and modelled.
    - **Level** refers to the average/mean value in the series
    - **Trend** refers to the long-term increasing or decreasing value in the series; it does not have to be linear and it can "change direction," going from an increasing trend to a decreasing trend
    - **Seasonality** refers to the repeating short-term cycle in the series; it occurs when a time series is affected by seasonal factors such as the time of the year or the day of the week and should be of a fixed and known frequency
- Non-Systematic components are those that cannot be directly modelled.
    - **Noise** refers to the random variation in the series that cannot be explained, also mentioned as residual

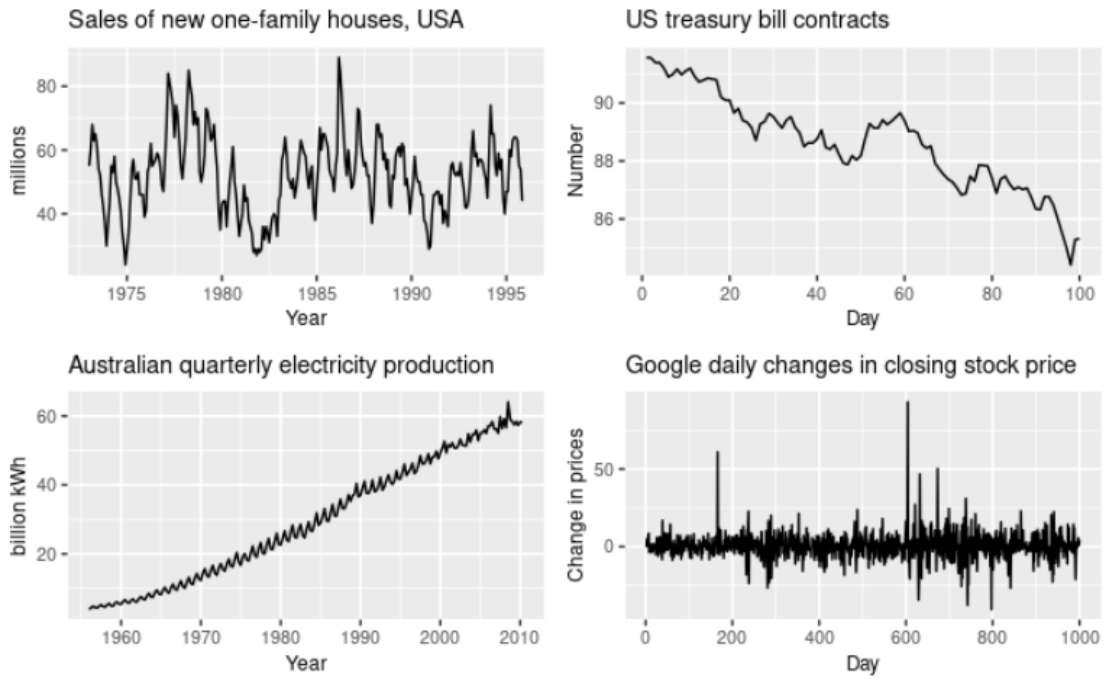


Figure 35: Examples of time series that exhibit (a) yearly seasonality, (b) downward trend, (c) increasing trend, and (d) no pattern observed

Any given time series is considered to be an aggregate or combination of these four components since all series have a level and most have a noise while the trend and seasonality components are optional.

The different components of the time series can combine either additively or multiplicatively. An additive model is linear where changes over time are consistently made by the same amount while a multiplicative model is nonlinear, such as quadratic or exponential, and changes increase or decrease over time.

Whether our model is **additive** or **multiplicative** (Figure 36) depends on whether the amplitude of the data's seasonality is level dependent. If the seasonality's amplitude is independent of the level, then we should use the additive model, and if the seasonality's amplitude is dependent on the level, then we should use the multiplicative model. In practice, the additive model is useful when the seasonal variation is relatively constant over time while the multiplicative model is useful when the seasonal variation increases over time.

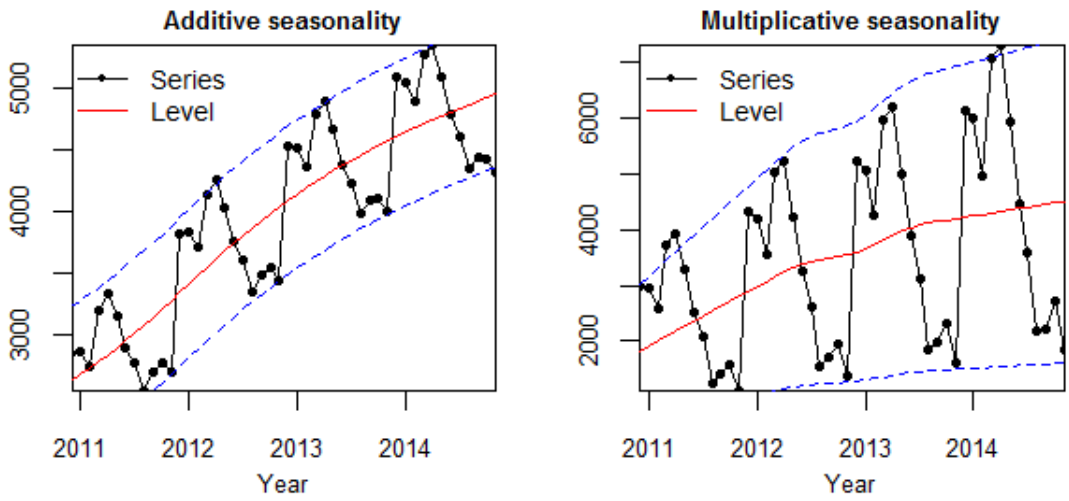


Figure 36: Additive vs multiplicative seasonality

Therefore, to get a better understanding of the time series, we proceeded to their decomposition using both the additive and multiplicative models and then plot the decomposition results from both methods.

For this purpose, we used the `seasonal_decompose` Python function from the `statsmodels` library, which returns and object with seasonal, trend, and residual attributes following a naïve decomposition method. The results are obtained by first estimating the trend by applying a convolution filter to the data, then the trend is removed from the series and the average of this de-trended series for each period is the returned seasonal component.

- The additive model is defined as: $Y[t] = T[t] + S[t] + e[t]$
- The multiplicative model is defined as: $Y[t] = T[t] * S[t] * e[t]$

The four decomposition components were then plotted from the result object, using the `plot()` Python function.

We run the decomposition function on the pointedly created `daily_sales` dataframe which includes the count of all items sold per day.

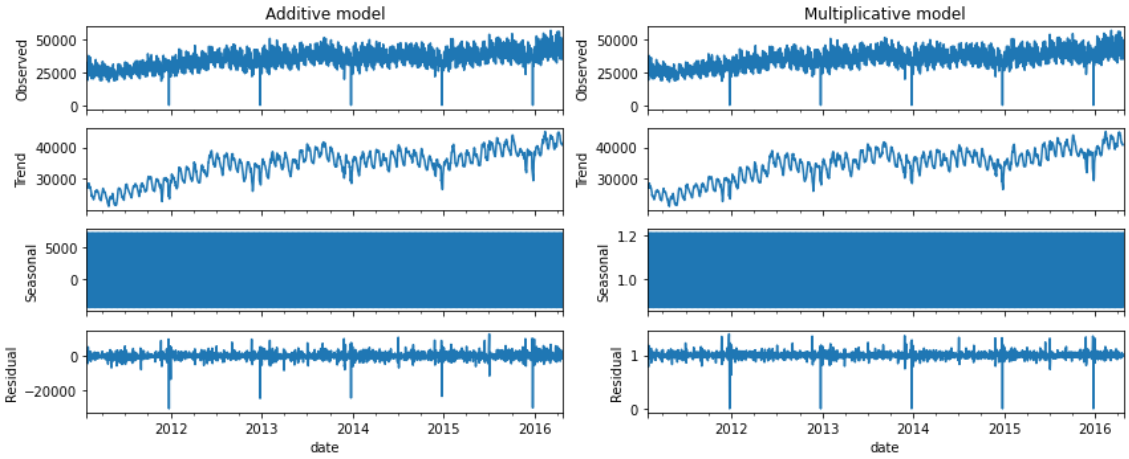


Figure 37: Decomposition results for additive vs multiplicative model

Both models have similar results in terms of trend and seasonality, while when looking at the residuals of the additive decomposition closely, it looked more random, which is why we selected the additive decomposition for the next step where we further compared different frequency levels for the additive model (30, 100 and 300 frequency) and plotted the results in a comparative chart.

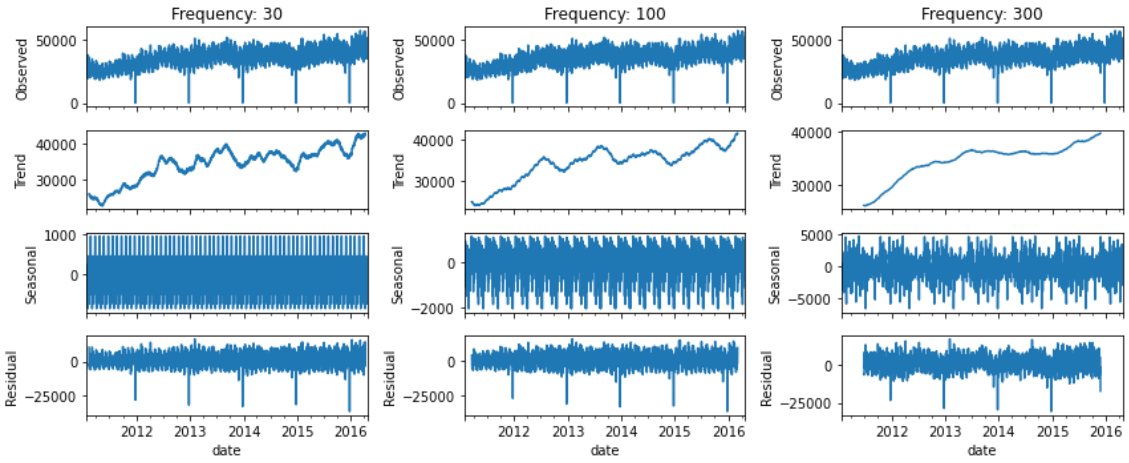


Figure 38: Decomposition results for different frequencies

From the first "Observed" graph that shows the actual time series, we understand that sales have an increasing trend over the years with the average moving upwards in time while there seem to be some periodical drops in sales which we need to further investigate. The second graph shows the trend, confirming that it is increasing over the years. The third graph shows there is strong seasonality which is clear for all examined frequency levels. The last graph shows that there is a lot of residual noise (randomness) in our time-series.

4.7.2. Aggregated sales

Since we are investigating time series data, the obvious graph to start with is a line chart with the observations plotted against the time of observation, with consecutive observations joined by straight lines. We used different aggregation levels for gaining different insights from the data.

To have a baseline for our analysis we started by plotting the daily sales (Figure 39), and by zooming in on the periodical drops in sales at our interactive plot (Figure 40: Daily overall sales line chart (focus on traffic drop)Figure 40), we could see that these were all recorded on Christmas day - the only day when all Walmart stores are officially closed.

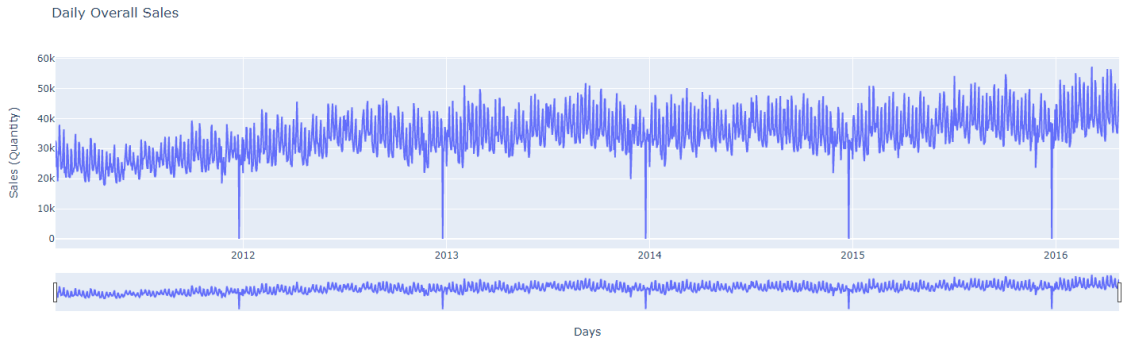


Figure 39: Daily overall sales line chart

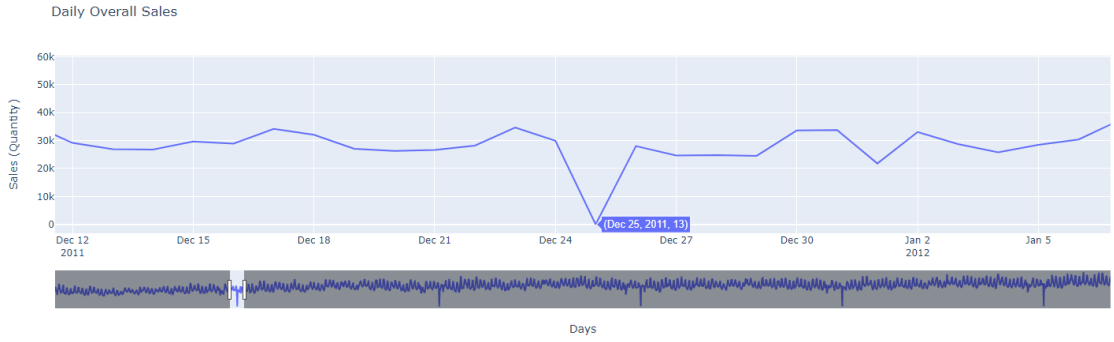


Figure 40: Daily overall sales line chart (focus on traffic drop)

4.7.2.1. Sales per State

By plotting the overall sales by state into a choropleth map (Figure 41) we could see a visual representation of the sales per state on the map of the USA.

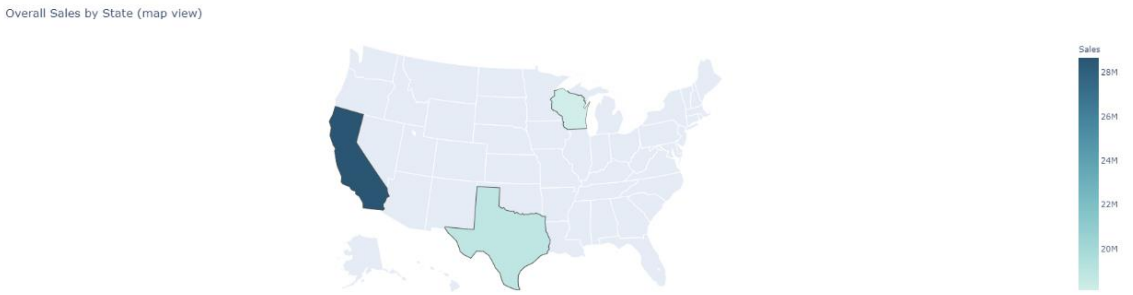


Figure 41: Choropleth map of sales per state

By plotting the overall sales per state in a bar chart and pie chart (Figure 42) we could see that California accounted for most of the sales with 28.7M items sold overall, which represents 43.6% of the overall sales. Texas was 2nd in overall sales with 18.9M items sold (28.8% share), followed closely by Wisconsin at 18.1M items sold overall (27.6% share).

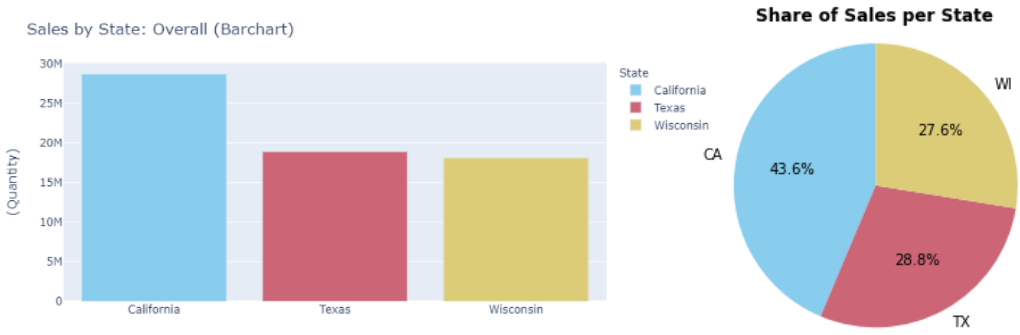


Figure 42: Overall sales by state (bar chart and pie chart)

We continued by analysing the sales per state on several aggregate levels, such as daily, weekly, monthly and yearly (Figure 43, Figure 44, Figure 45, and Figure 46 respectively).

Sales for California have consistently been the highest throughout the observed period, and they were the most impacted by seasonality - the peaks in August are most evident in California in comparison to the other states.

Wisconsin has shown the highest increase in sales over the years; while it had initially (until 2013) lower sales than Texas, it slowly approached and remained at similar levels until Aug-2015 and eventually surpassed Texas towards the end of the observed period.

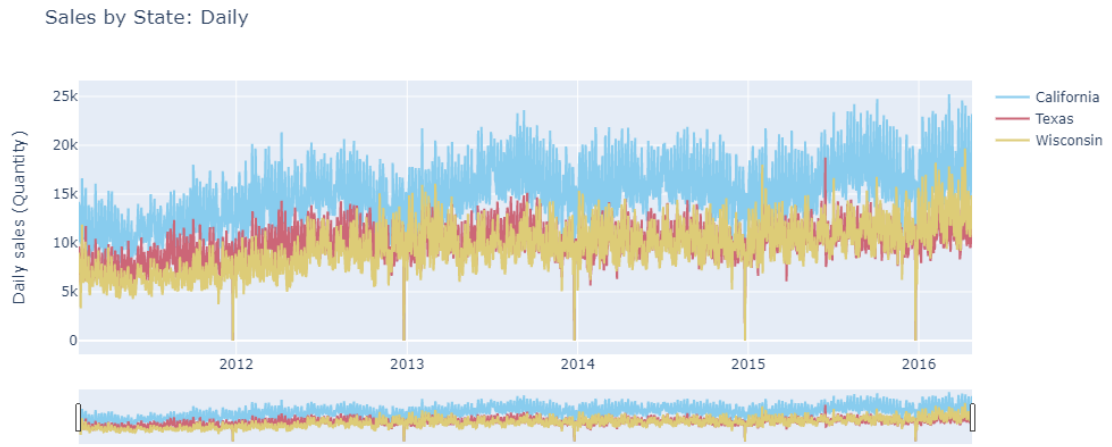


Figure 43: Daily sales by state

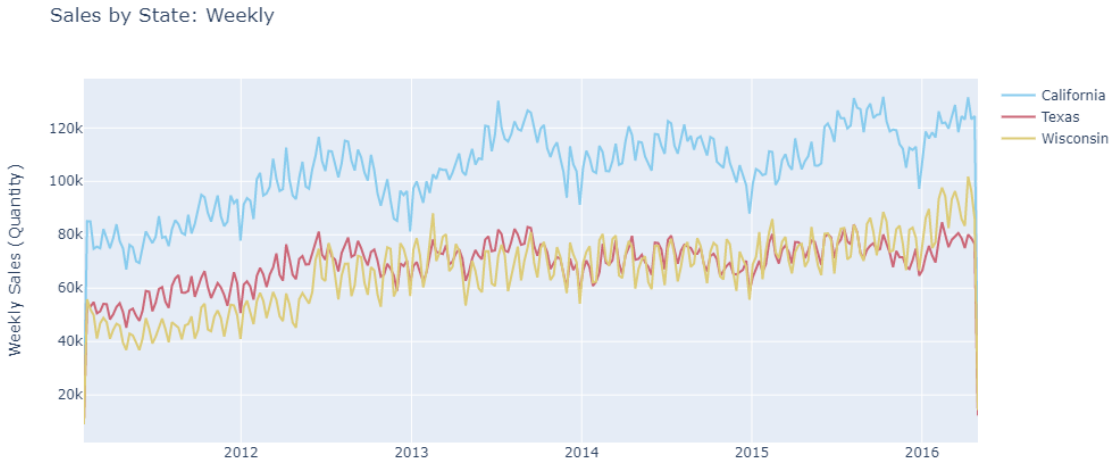


Figure 44: Weekly sales by state

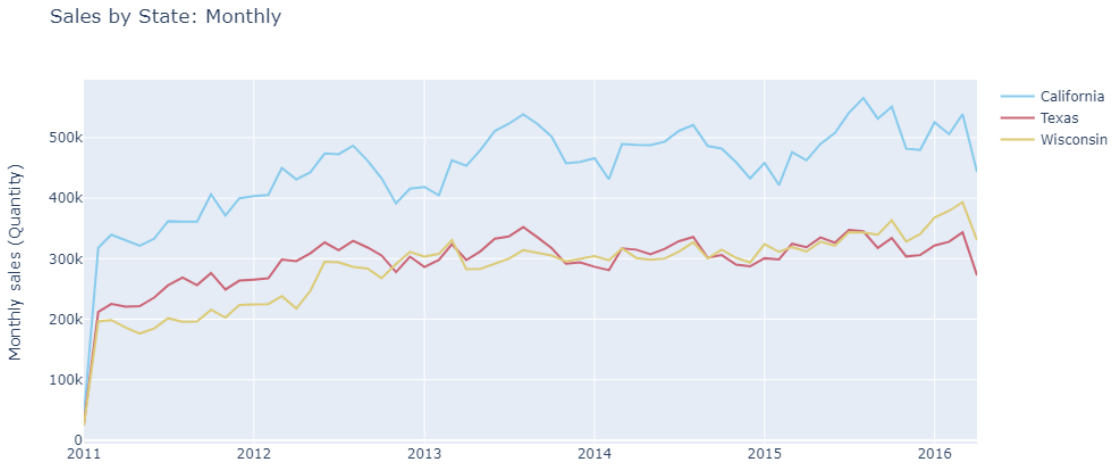


Figure 45: Monthly sales by state

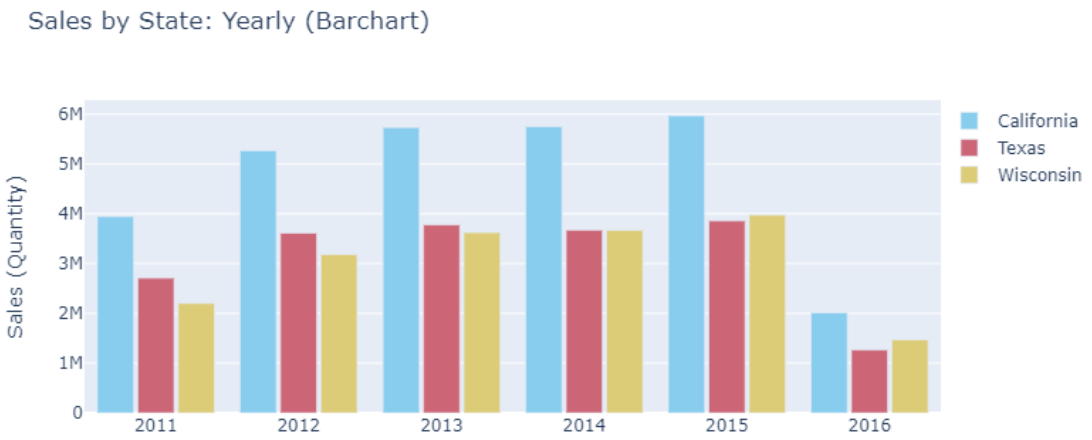


Figure 46: Yearly sales by state

We also conducted a state-specific decomposition analysis, comparing the components across the three states, as seen in the figure below:

- California and Wisconsin exhibited a gradually increasing trend while Texas grew fast in the beginning and then remained relatively stagnant
- California and Texas have a similar yearly seasonality, peaking around July-August and dipping during December-January
- Wisconsin exhibits a somewhat different seasonality, peaking in March, dipping in April and then peaking again in August

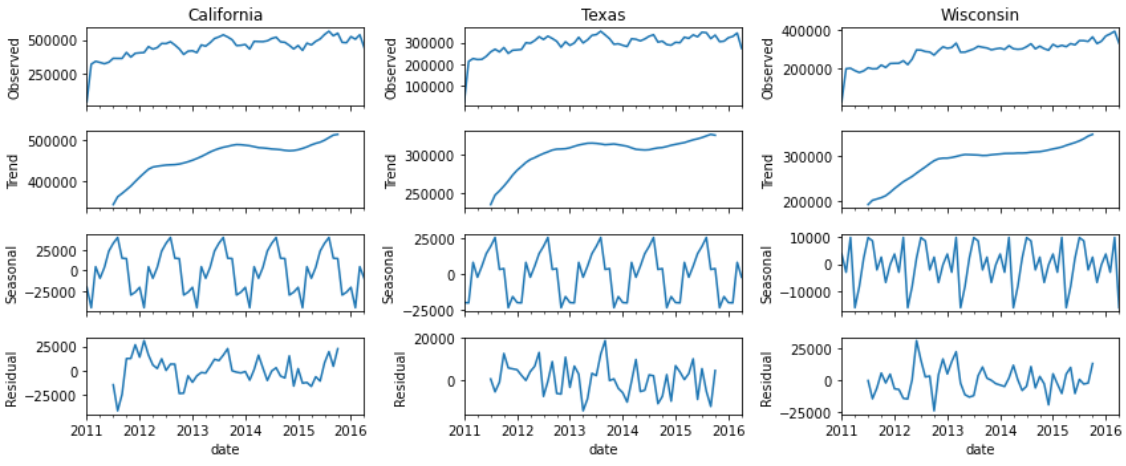


Figure 47: Decomposition results per state

4.7.2.2. Sales per Store

We had a total of 10 stores in the analysed dataframe: 4 were located in California, 3 in Texas and 3 in Wisconsin. The California stores seem to have higher overall average sales, with Texas and Wisconsin following at similar levels in general.

Figure 48: Average sales per store

When plotting the daily sales per store the chart was expectedly not very insightful (Figure 49), therefore, to have a more helpful view of all 10 stores we used the 30-days rolling average to plot the sales for each store (Figure 50).

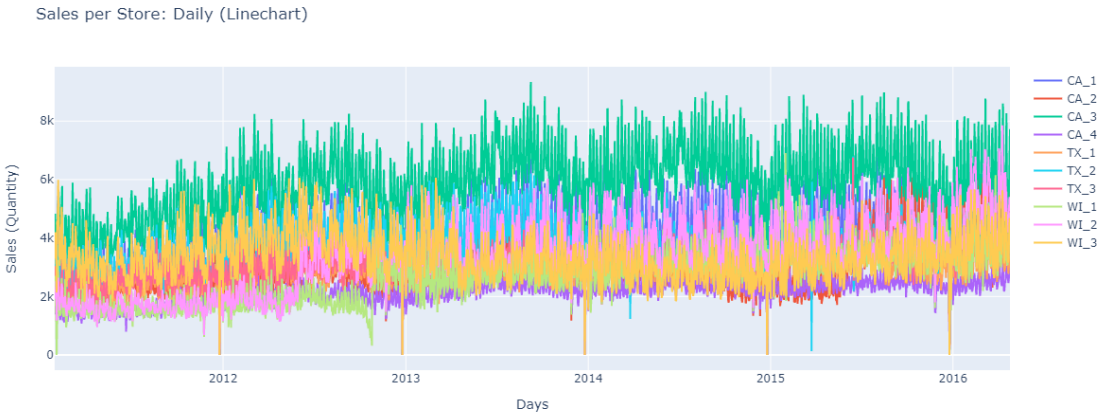


Figure 49: Daily sales per store

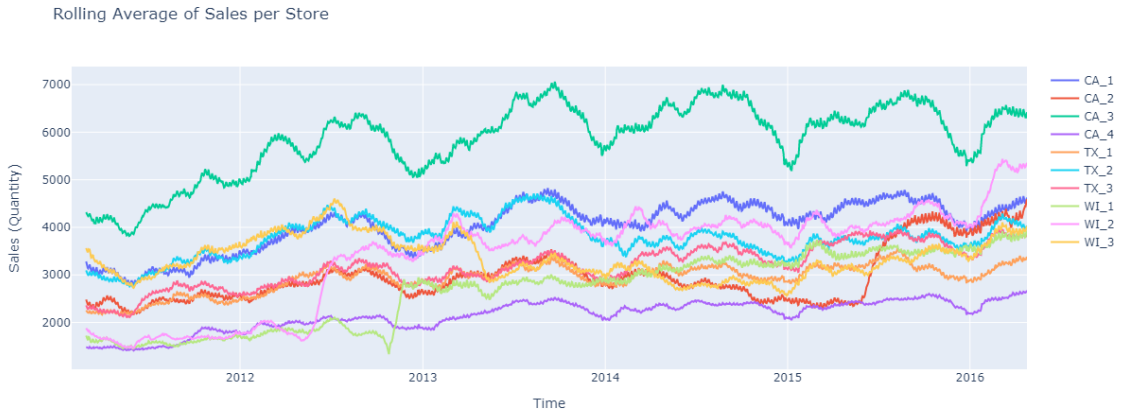


Figure 50: Rolling average of sales per store

In general, most sales curves tend to exhibit a "linear oscillation" trend where sales grow linearly in the long run despite some short-term fluctuations. This seem to also be the case for the sales in our dataset.

The stores in California seem to have the highest variance in sales, indicating that some of the state's stores grew significantly faster than others. The stores in Wisconsin and Texas were quite consistent in sales, without much variance, indicating that development was more stable in these states

By a similar logic as the line charts above, when plotting a boxplot of the sales data per store we could use either the actual or the rolling average for our observations. As seen in the figures below, the rolling average box plots are less affected by the outliers and more clearly represent each store's performance.

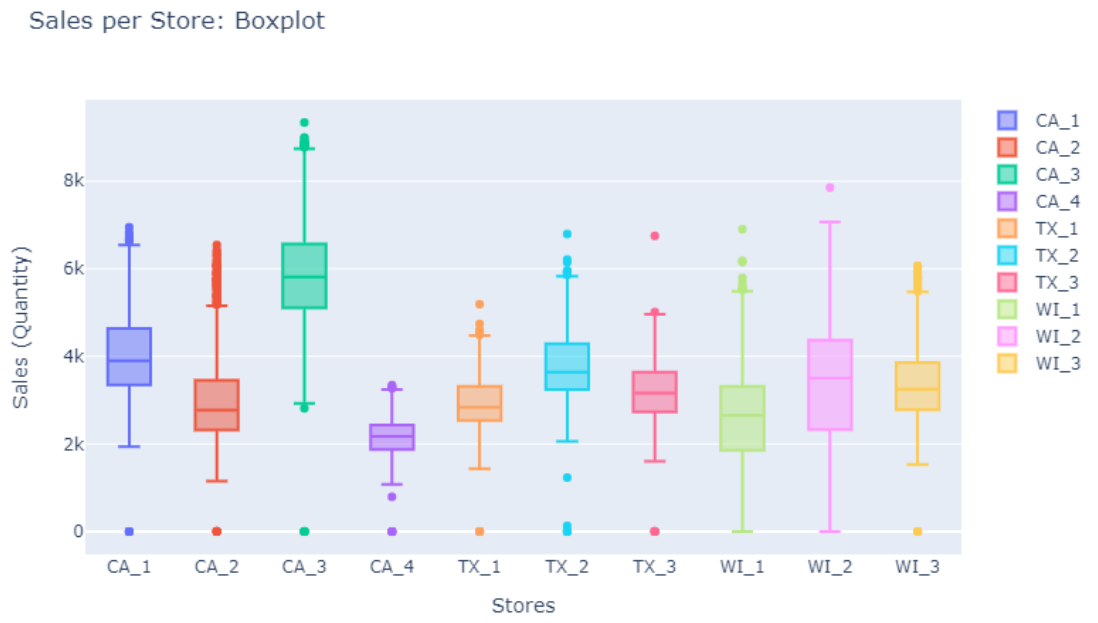


Figure 51: Box plots of sales per store

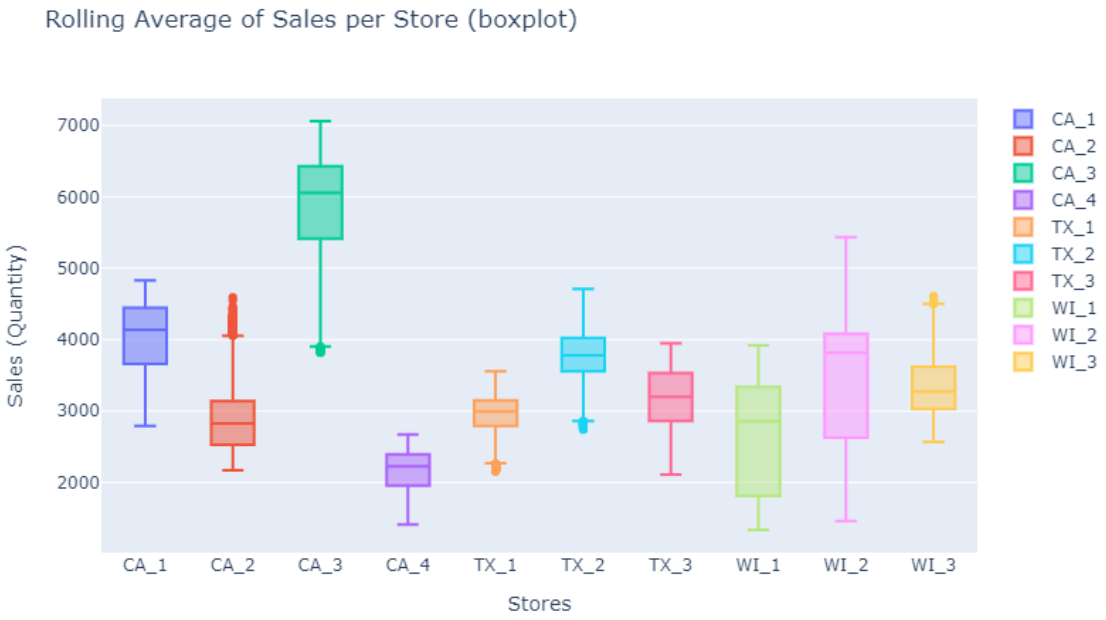


Figure 52: Box plots of rolling average of sales per store

4.7.2.3. Sales per Category

Products sold in each store are separated into 3 categories: Foods, Hobbies, and Household. As seen in the pie chart in Figure 53, Foods' sales accounted for 69% of all items sold, followed by Household that accounted for 22% and Hobbies that contributed the remaining 9%.

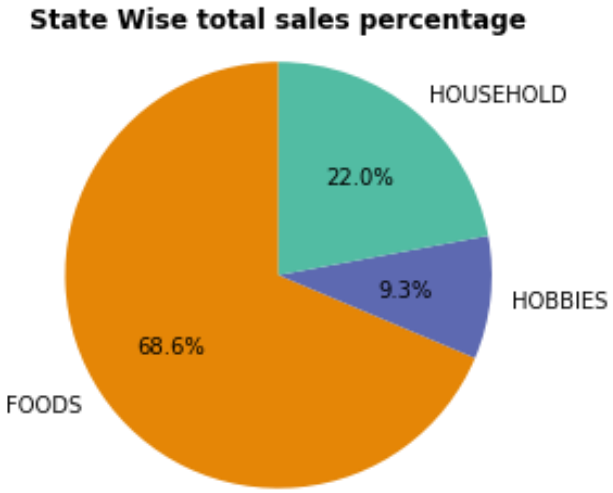


Figure 53: State wise total sales percentage

By plotting the monthly sales by category in a line chart (Figure 54), we could see that the sales of Foods items have historically been much higher than sales for Household and Hobbies, throughout the whole observed period.

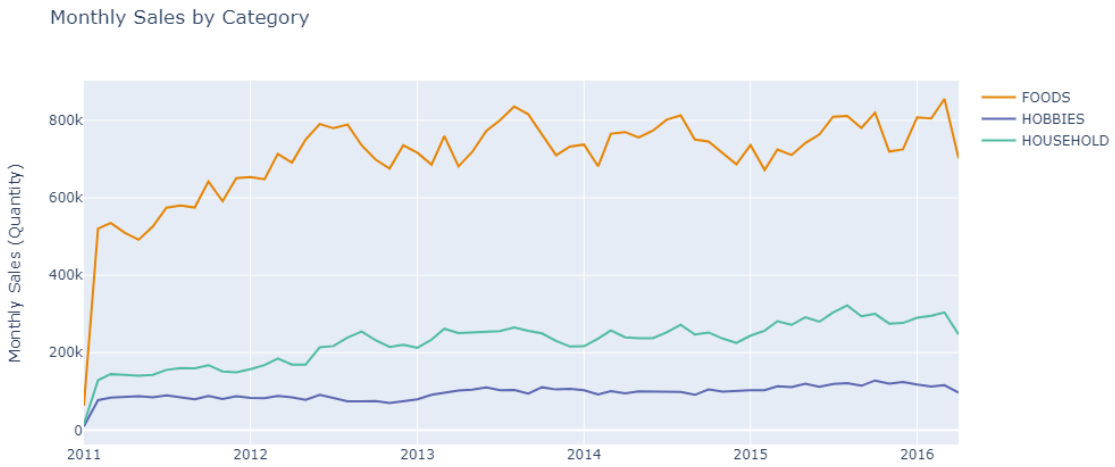


Figure 54: Monthly sales by category

By further plotting the decomposition components for each category time-series on a monthly aggregation level we note the following observations:

- Foods' sales had a sharper increase from 2011 to 2012 and have remained relatively stagnant since then, exhibiting a clear yearly seasonality with two spikes that occur in August and March
- Hobbies sales have had a flatter trend with a less clear seasonality - the spikes in March are closer to the August spikes in terms of volume. Sales increased from Aug-2012 to Aug-2013 and then from Aug-2014 to Aug-2015, remaining stagnant in-between
- Household sales have had the most increasing trend over the years, exhibiting a clear seasonality in March and in August

Figure 55: Decomposition results per category

4.7.2.4. Sales per Category and State

We proceeded by plotting the sales for the combination of each of the categories in each of the states (Figure 56) and then analysing them from either a state (Figure 57) or a category (Figure 58) perspective.

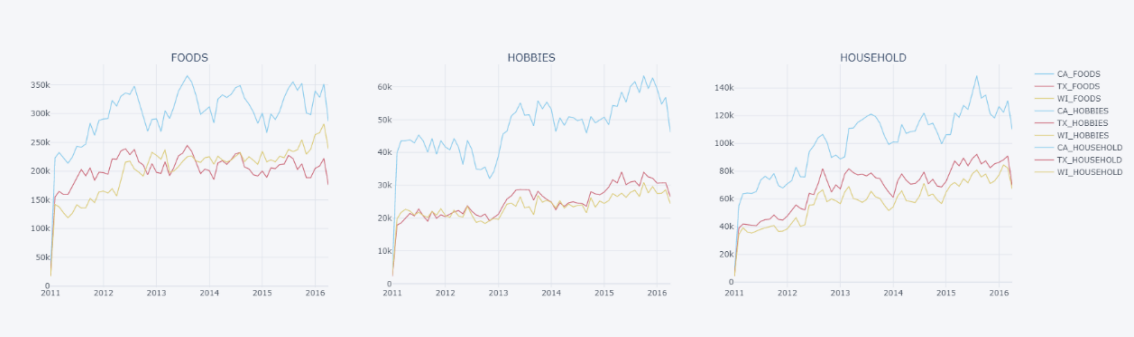


Figure 56: Monthly sales by category and state

From a state perspective,

- California spent 67% on Foods, 11% on Hobbies and 22% on Household items; it also accounted for more than 40% of the items sold in the Foods and Household categories and about 50% of items sold in the Hobbies category.
- Texas spent 69% on Foods, 8% on Hobbies and 23% on Household items; it has a 30% share of the items sold in Foods and Household categories and about 25% of the items sold in the Hobbies category
- Wisconsin spent 72% on Foods, 8% on Hobbies and 20% on Household items; it accounted for 25% of the items sold in Hobbies and Household categories and about 30% of the items sold in the Foods category

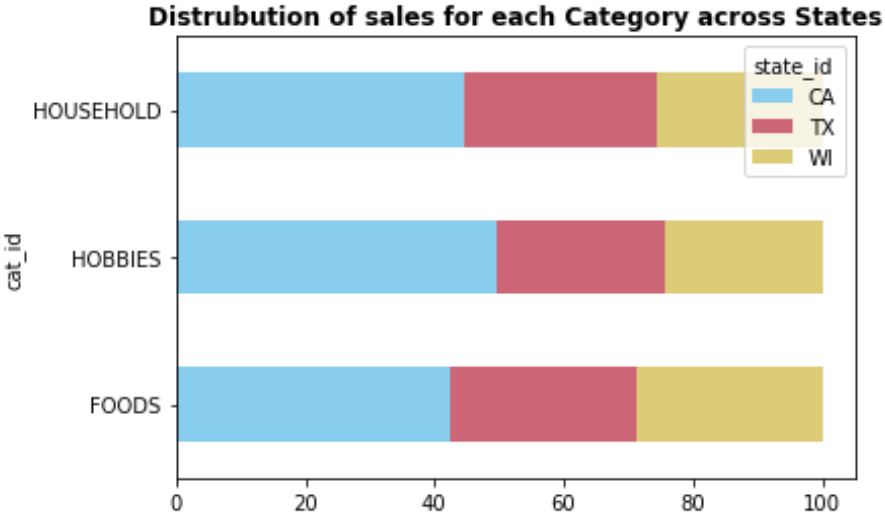


Figure 57: Share of sales for each category across states

From a category perspective,

- Sales of Foods items in California have had a high yearly seasonality with peaks in August but overall sales have not increased significantly after 2012. Texas sales have had a similar trend as California (although at a lower scale) with high monthly seasonality peaking in August and dipping around Jan-Feb with the overall trend remaining the same after 2012. Wisconsin sales of Foods items have not had any clear seasonality while increasing significantly over the years, especially comparing 2011 to 2016 being essentially the only state showing an increase in Foods items sold over the years
- Looking at all states, sales of Hobbies items did not seem to have any visible seasonality. In California, sales of Hobbies items have had an increasing trend, more specifically, they appear to increase every second year and relatively decrease in-between; this is especially noticeable at the drop starting in Aug-12 which was followed by an increase starting Jan-13. The sales of Hobbies items in Texas and Wisconsin also showed a similar increasing trend as California, with Texas performing better than Wisconsin.
- Household sales have had the most consistently and increasing trend amongst the three categories, in all three states. California had the highest increase over the years and a clear yearly seasonality peaking in August and dipping around Dec-Jan. Texas had overall better sales than Wisconsin throughout the years but both show similarly increasing trends; they also both have a weak seasonality with relatively visible peaks in August and March.

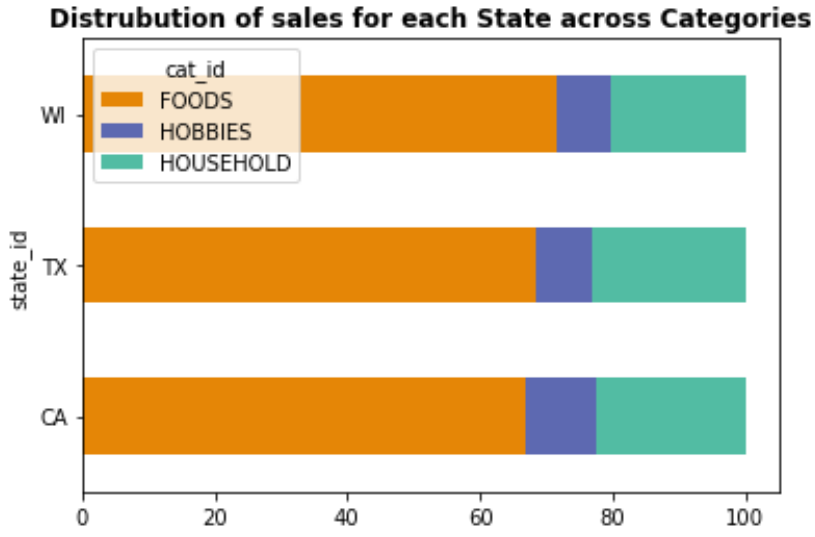


Figure 58: Share of sales for each state across categories

4.7.2.5. Sales per Department

Each of the categories is further separated into 2 or 3 departments: Foods is separated into 3 departments, Hobbies is separated into 2 departments, and Household is separated into 2 departments.

We started by plotting the monthly sales by department as seen in Figure 59 and proceeded with the 100% stacked bar plots showing the share of sales for each store across departments (Figure 60) and each department across stores (Figure 61).

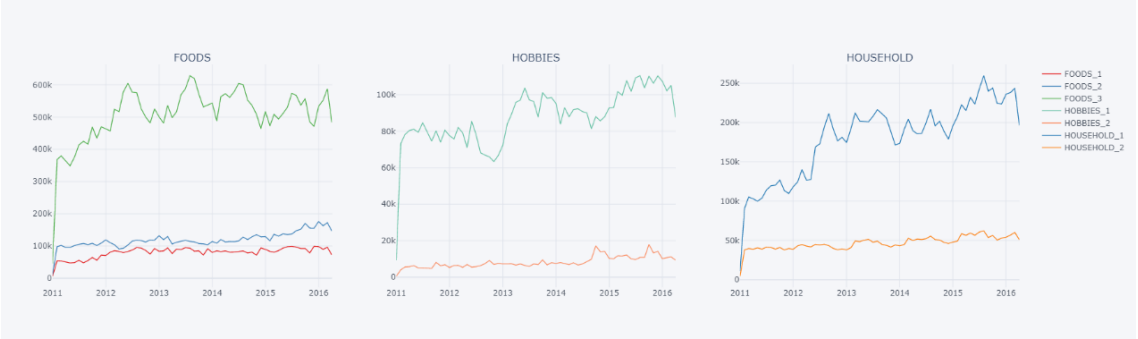


Figure 59: Monthly sales by department

Some observations that were made from these exploratory charts:

- FOODS_1 remained relatively stagnant over the years, especially after 2012
- FOODS_2 increased especially over the years 2015-2016, going from a higher value of 130k to more than 170k
- FOODS_3 was the department with the most items sold, it also had a high seasonality ranging from 500k to 600k throughout the year, with peaks around the summer months
- HOUSEHOLD_1 had the highest increase in sales over the years in comparison to all other departments, doubling from about 100k in 2011 to more than 200k in 2016
- HOUSEHOLD_2 has been steadily increasing with a very clear seasonality of peaking around the summer months
- HOBBIES_1 included most of the items sold under the Hobbies category, while HOBBIES_2 was at much lower levels
- HOBBIES_2 had a high seasonality with peaks in October of each year

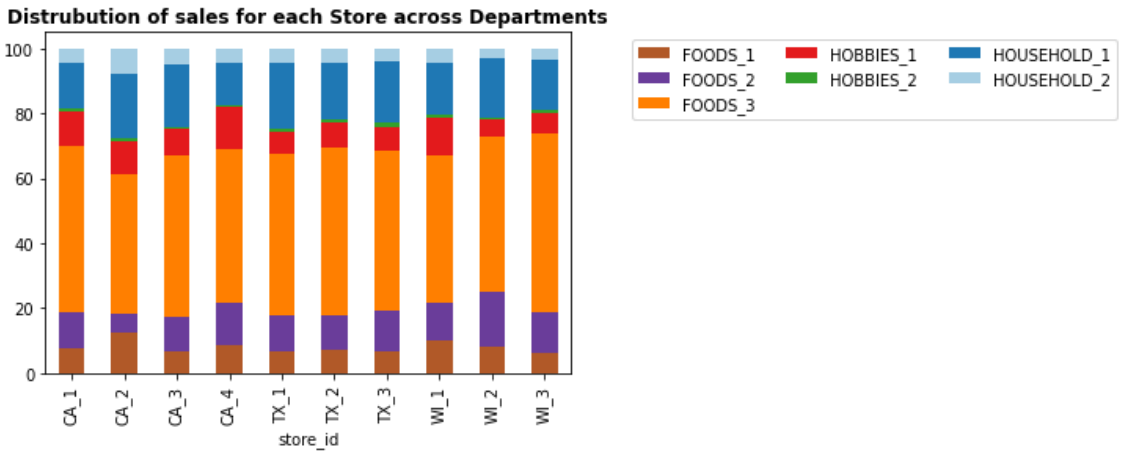


Figure 60: Share of sales for each store across departments

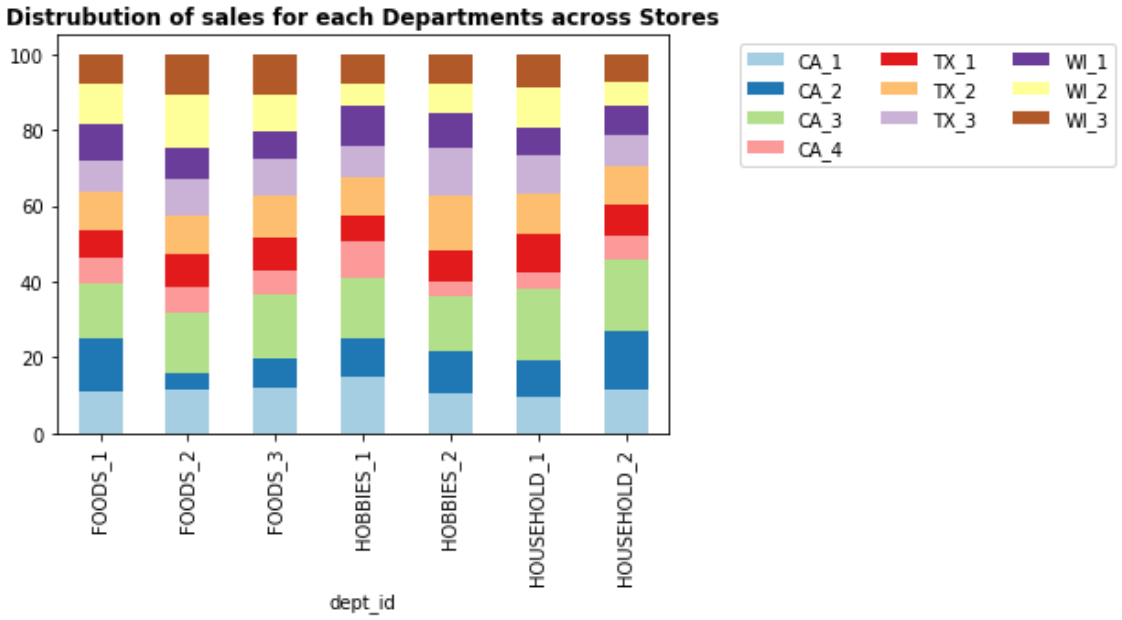


Figure 61: Share of sales for each department across stores

4.7.3. Seasonality study

There is a clear weekly seasonality in sales by state with sales being higher during and closer to the weekend (Friday through Monday) and lower in the middle of the week (Tuesday through Thursday), as indicated by the bar chart in Figure 62.

The weekly seasonality is also reflected in sales by category (Figure 63), with sales for all three categories being higher around the weekend (Friday through Monday) and lower in the middle of the week (Tuesday through Thursday). Foods items have more sales on Sundays (7.8M), followed by Saturdays (7.7M). Hobbies and Household items have more sales on Saturdays compared to Sundays, however, the differences between the two days are not that significant.

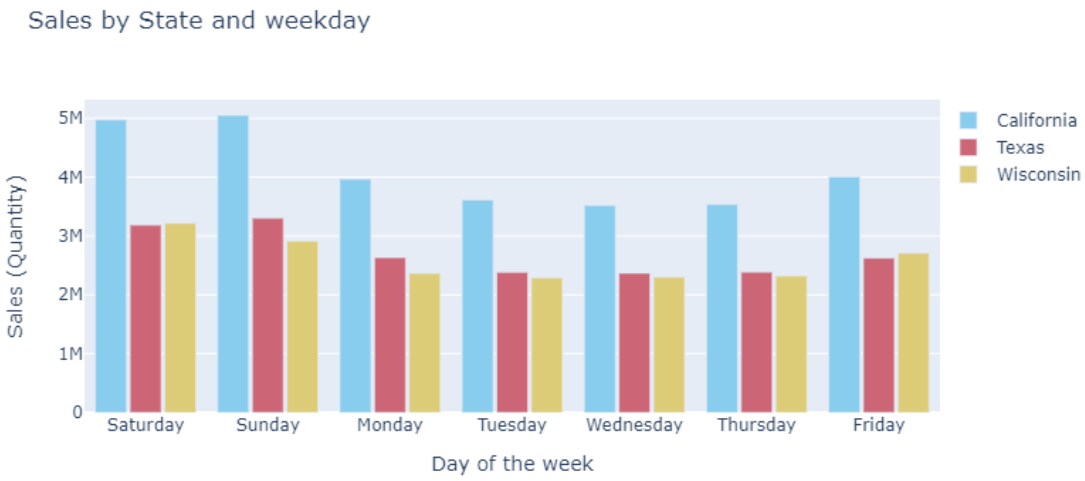


Figure 62: Sales by state and weekday

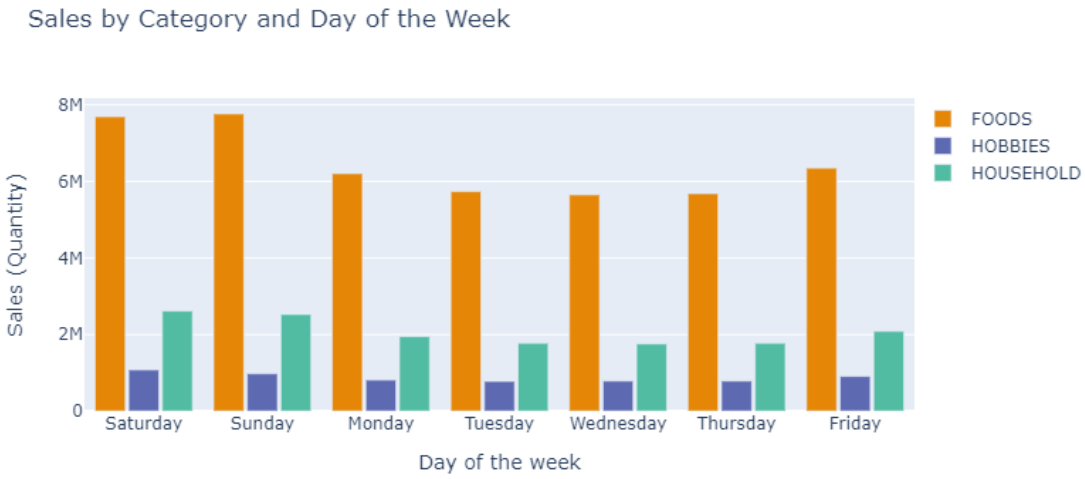


Figure 63: Sales by category and day of the week

As showcased in Figure 64, in both California and Texas, sales were higher during March and April and lower on the months towards the end of the year (November-December) while in the case of Wisconsin, the month of May had the lowest sales throughout the year.

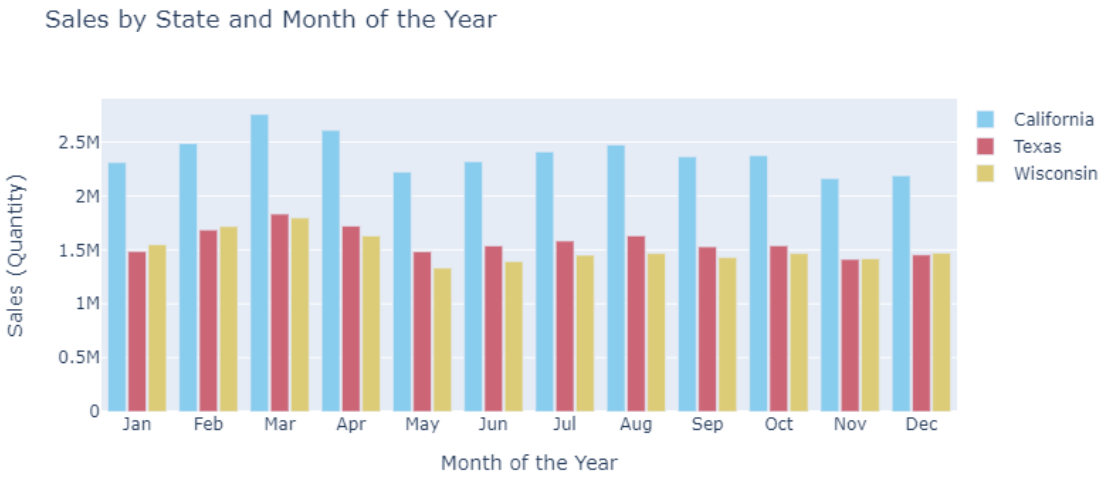


Figure 64: Sales by state and month of the year

4.8. Explanatory variables analysis

As previously discussed, the dataset also involved exogenous/explanatory variables, including calendar-related information and selling prices. Thus, apart from the past unit sales of the products and the corresponding timestamps (e.g., date, weekday, month, and year), there was also information available about:

- Special events and holidays (e.g., Super Bowl, Valentine's Day), organized into four classes, namely Sporting, Cultural, National, and Religious
- Selling prices, provided as an average across seven days on a week-store level.
- SNAP activities that serve as promotions

4.8.1. Events

There are 4 different event types in the calendar dataset and each of them includes several different event names, ranging from 3 to 10 per event type with the Religious type including the most events and the Sporting having the least. In total, there are 27 different event names which are presented in Table 12.

When looking at the days when sales where close to zero (Table 13), we confirm that Christmas day was the only day throughout the year that all Walmart stores were closed. Additionally, the WI_1 store was also closed on Feb 2nd, 2011 which was due to a strong blizzard that occurred on that day shutting down southern Wisconsin (Groundhog Day Blizzard, 2011).

Table 12: Types of events and the event names they include

	Cultural	National	Religious	Sporting
0	ValentinesDay	PresidentsDay	LentStart	SuperBowl
1	StPatricksDay	MemorialDay	LentWeek2	NBAFinalsStart
2	Cinco De Mayo	IndependenceDay	Purim End	NBAFinalsEnd
3	Mother's day	LaborDay	OrthodoxEaster	NaN
4	Father's day	ColumbusDay	Pesach End	NaN
5	Halloween	VeteransDay	Ramadan starts	NaN
6	Easter	Thanksgiving	Eid al-Fitr	NaN
7	NaN	Christmas	EidAlAdha	NaN
8	NaN	NewYear	Chanukah End	NaN
9	NaN	MartinLutherKingDay	OrthodoxChristmas	NaN

Table 13: Days with close to zero sales

	date	event_name_1
4	2011-02-02	0
330	2011-12-25	Christmas
696	2012-12-25	Christmas
1061	2013-12-25	Christmas
1426	2014-12-25	Christmas
1791	2015-12-25	Christmas

Regarding SNAP activities, the official description of SNAP (Supplemental Nutrition Assistance Program) is that it can be used by USA households to buy nutritious foods such as bread and cereals, fruits and vegetables, meat and fish and dairy products. SNAP benefits cannot be used to buy any kind of alcohol or tobacco products or any non-food items like household supplies and vitamins and medicines (What is SNAP and How to Apply, n.d.).

An assumption from this description is that SNAP days would affect positively sales of items mostly in the Foods category. By plotting the average sales of SNAP vs non-SNAP days per state (Figure 65) we see that all states had higher average sales on SNAP days

versus non-SNAP days. California average sales were 8.0% more on SNAP vs non-SNAP days, in Texas the difference between SNAP vs non-SNAP days was at 11.5% while in Wisconsin average sales were 21.8% more on SNAP vs non-SNAP days.

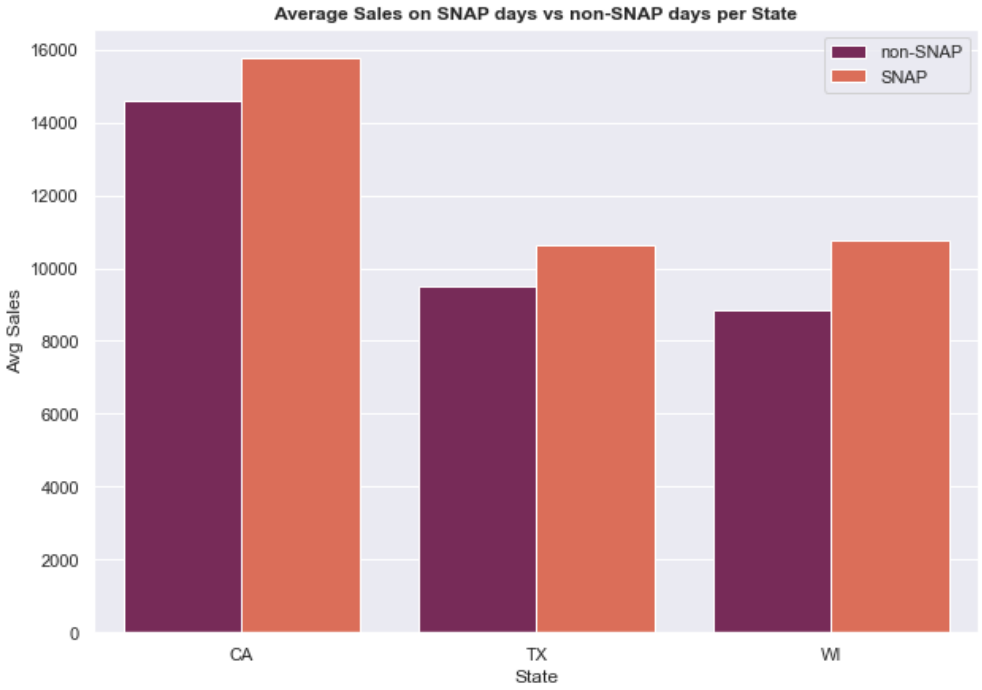


Figure 65: Average Sales on SNAP days vs non-SNAP days per State

4.8.2. Selling prices

To explore the impact of item selling prices on each of the categories we plot some distribution charts of the prices by category (Figure 66 and Figure 67).

A logarithmic price scale is preferred for the analysis so that the smaller prices are represented equally on the chart. This type of scale is generally used for the long-term perspective analysis of price changes (Chen, 2021). Furthermore, we use log1p here (natural algorithm+1) due to looking at very low prices (NumPy v1.22 Manual, n.d.).

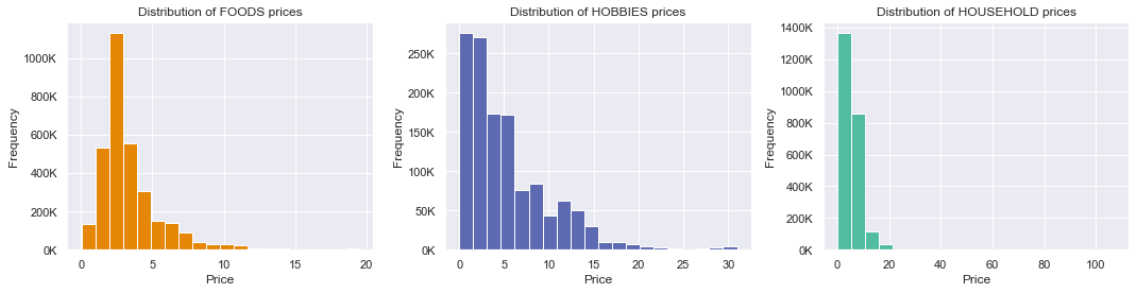


Figure 66: Distribution of prices for each category

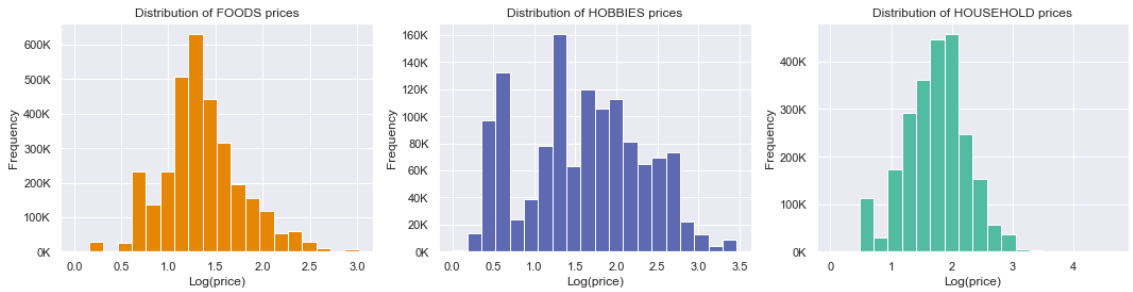


Figure 67: Distribution of prices for each category (logarithmic scale)

These charts reveal that most of the prices for Foods items lie between $1-$10. The peak at $3 shows that more than 1 million Foods items are priced around that value. Hobbies items exhibit a wider range of prices with about 900 thousand of them below $5. Household items are generally more expensive than Foods and Hobbies items, peaking closer to $10.

The box plot of max price change across categories (Figure 68), allows us to deduct that most products do not change their price throughout the years since the majority of price changes are concentrated at the base of the box plot. More specifically, 27% of all items exhibited no price change throughout the years and the changes that did happen were mostly restricted to below $15.

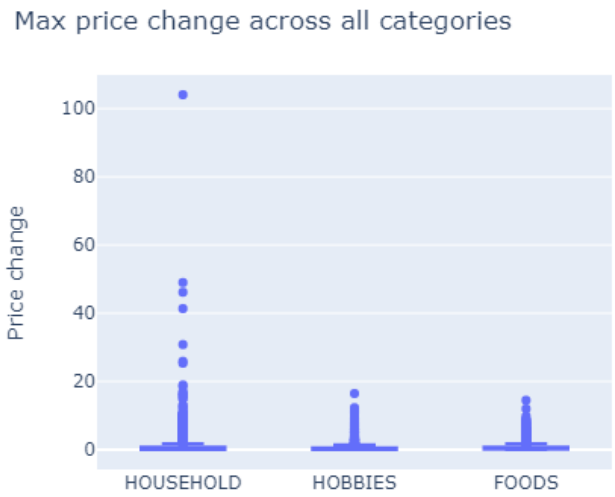


Figure 68: Max price change across all categories

Household items have had the highest price changes over the years which is evident from the above box plot and their prominence at the top 10 items with the biggest price change over the years (Table 14).

Table 14: Top 10 items with biggest price change over the years

	item_id	store_id	sell_price_max	sell_price_min	price_change	category
0	HOUSEHOLD_2_406	WI_3	107.32	3.26	104.06	HOUSEHOLD
1	HOUSEHOLD_2_406	WI_2	61.46	12.46	49.00	HOUSEHOLD
2	HOUSEHOLD_2_466	TX_1	52.62	6.46	46.16	HOUSEHOLD
3	HOUSEHOLD_2_178	TX_1	44.36	3.00	41.36	HOUSEHOLD
4	HOUSEHOLD_2_250	WI_2	34.18	3.36	30.82	HOUSEHOLD
5	HOUSEHOLD_2_406	WI_1	35.88	9.97	25.91	HOUSEHOLD
6	HOUSEHOLD_2_250	WI_1	30.32	4.97	25.35	HOUSEHOLD
7	HOUSEHOLD_1_469	WI_3	19.97	1.00	18.97	HOUSEHOLD
8	HOUSEHOLD_2_514	TX_2	19.54	1.00	18.54	HOUSEHOLD
9	HOUSEHOLD_1_342	WI_3	17.97	1.00	16.97	HOUSEHOLD

5. Feature Engineering

5.1. Introduction

After obtaining detailed insights about the dataset through the EDA process, feature engineering is carried out. Feature engineering is primarily divided into two parts: creation of derived variables (which summarize the linear relationships among attributes and simplify the more complex ones) and transformation (which converts complex non-linear relationships into linear and standardizes values for better understanding) (Ghosh, Nashaat, Miller, Quader, & Marston, 2018).

The majority of practical machine learning uses supervised learning which is when there are some input variables (X) and an output variable (y) and an algorithm is used to learn the mapping function from the input to the output. It is called supervised learning because the process of the algorithm learning from some known dataset can be thought of as a teacher supervising the learning process. (Brownlee, Introduction to Time Series Forecasting with Python, 2020).

A usual dataset for machine learning consists of observations which are treated equally regardless of their order – meanwhile in time-series data have an explicit dependence on the occurrence of the observations in time, which each successive data point depending on its past values and possibly exhibiting certain trends or seasonality.

Therefore, to utilize machine learning algorithms, it is necessary to reframe the time-series forecasting problem as a supervised learning problem, by creating or inventing new input variables, known as features which would assist us in modelling the strong and simple relationships between the input (X) and the output (y).

In our case, the output variable to be predicted is the sales per day for each item and the input features are information such as the sales values of previous days/weeks/etc, the category of the product, any event occurring on the to-be-predicted day, historic prices etc. which will be used to make predictions.

Before moving on, we should also briefly define some standard terms used when describing time series data: `t` represent the current time, `t-n` refers to a prior time which is also called lag time (n=1, 2, 3…), `t+n` refers to a future time or the time of the forecast, and `lags` stands for observations made at prior times.

5.2. Reducing the data size

After calculating the memory usage of the dataframes, we applied some transformations to reduce (downcast) the data. For numerical columns, we used a subtype of int and float which is less memory consuming, and an unsigned subtype if there was no negative value. For categorical columns, we forced pandas to use a virtual mapping table with low cardinality where all unique values are mapped via an integer instead of a pointer using the category datatype.

After downcasting, the memory usage of all three dataframes significantly decreased:

- `df_sales` decreased by -79%
- `df_prices` decreased by -78%
- `df_calendar` decreased by -50%

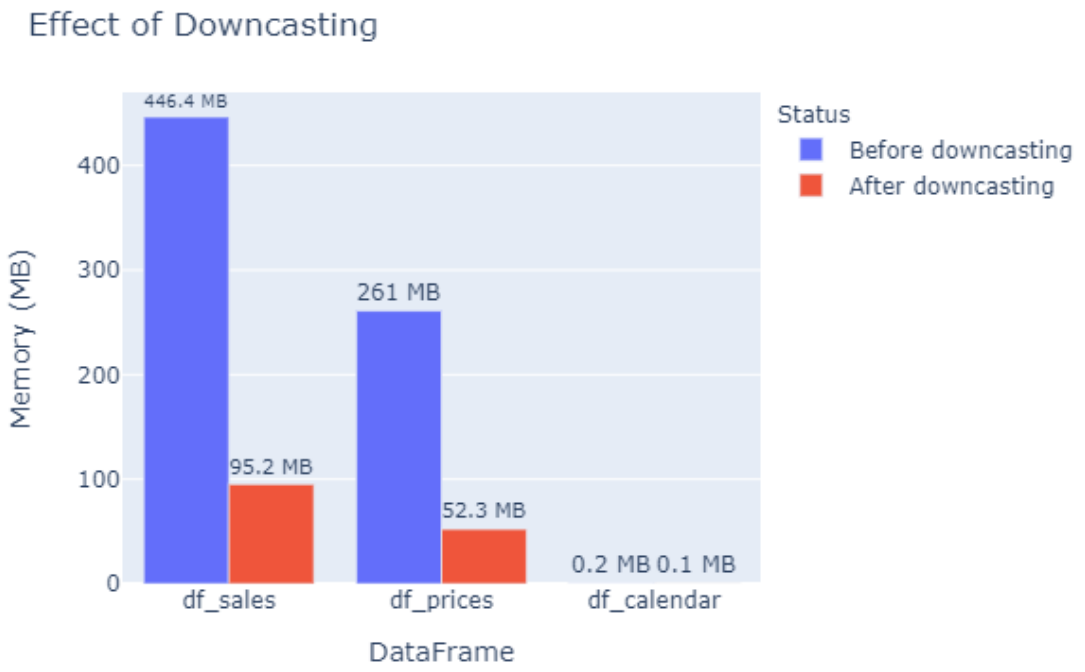


Figure 69: Effect of downcasting on the dataframes

Next, we proceeded to melting the sales dataframe, by unpivoting it from wide to long format, setting some columns as identifier variables (`id`, `item_id`, `dept_id`, `cat_id`, `store_id`, `state_id`). We also merged all dataframes into a single one that includes all data, joining them on the `d`, `store_id`, `item_id` and `wm_yr_wk` columns.

5.3. Encoding the categorical variables

Most of the forecasting models will perform better if the data provided for training, testing, fitting, etc. includes only numerical instead of categorical values, prompting the encoding of all categorical data (Greate Learning Team, 2021; Laerd Statistics team, n.d.).

Categorical data can be of different types, namely, nominal and ordinal.

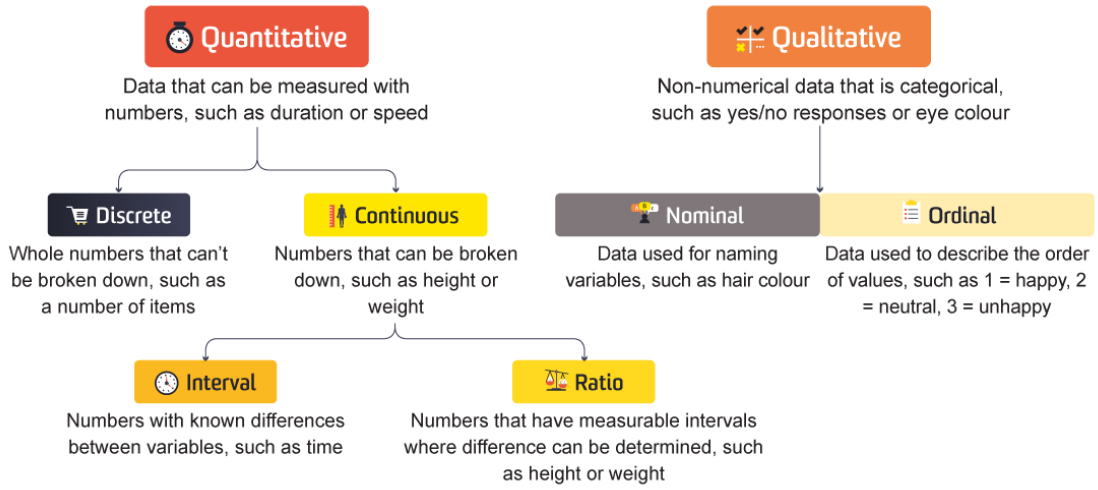


Figure 70: Types of data (UNSW online, 2020)

Nominal data refer to variables that have two or more categories, but which do not have an intrinsic order (Figure 71). The different categories can also be referred to as groups or levels of the nominal variable and are usually names and labels. We can assign each variable to a code which will not have any numerical significance, meaning that the basic mathematical operations such as addition, subtraction, multiplication, or division are pointless; the codes should not overlap with each other.

Primary colours	Do you own a phone?	Where do you live?
• Red • Yellow • Blue	• Yes • No	• Thessaloniki • Athens • Xanthi • Patras • Crete

Figure 71: Examples of nominal variables

Ordinal data refer to variables that have two or more categories which can be ordered or ranked (Figure 72). However, whilst we can rank the levels, we cannot place a "value" to them; we cannot e.g., say that "Good" is twice as positive or negative as "Fair". Once the data is collected, we can assign a numerical code to represent the ordinal variable.

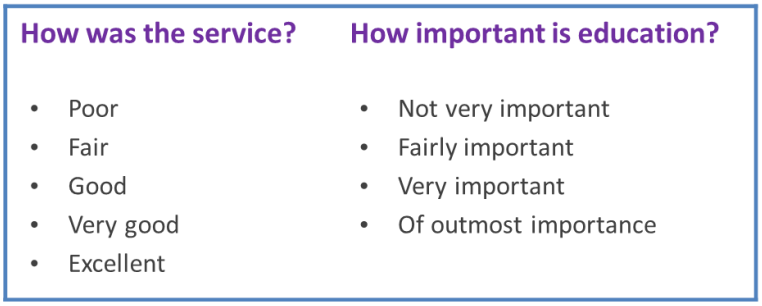


Figure 72: Examples of ordinal variables

In our implementation, we first applied label encoding to all categorical values, replacing them with their corresponding codes based on alphabetical ordering. We stored the labels along with their codes so that they can later be retrieved for submission. In general, the challenge with label encoding is to avoid the model capturing any relationship between the different codes although they don't have any mathematical significance. For example, when label encoding state names, California comes before Wisconsin alphabetically however due to them being converted to codes, the model might deduct that California>Wisconsin.

Next, we mean encoded some selected groups of variables. Unlike label encoding, which encodes the variables efficiently but randomly, the approach mean encoding takes is more logically using the target variable as the basis to generate the encoded feature (Monteiro, 2018). From a mathematical point of view, mean encoding represents a probability of the target variable, conditional on each value of the feature, essentially embodying the target variable in its encoded value. Mean encoding increases the quality of a classification model, however, it increases the danger of overfitting due to the fact that we are encoding the feature based on target classes which may lead to data leakage, rendering the feature biased.

To accomplish the above-described tasks, we used several Python functions as below:

- `pandas.DataFrame.memory_usage` function returns the memory usage of each column in bytes (this can optionally include the index and object dtypes)
- `pandas.DataFrame.astype` function to change one or more of the dataframe's columns to the defined column-specific types (e.g., int8, float16, category, etc.)
- `pandas.to_datetime` function to convert the values of a column with a valid date format, to datetime object which allows for easier date-related manipulations

- pandas.DataFrame.to_pickle function to save the downcasted dataframes as binary files on the hard drive, and load them whenever they are needed
- del keyword to remove unwanted objects which were e.g., created during EDA and are not further needed, and free memory for further processing; the del statement can be used to remove variables, user-defined objects, lists, items within lists, dictionaries, entire tuples and strings, but it cannot delete items of tuples and strings because tuples and strings are immutable (i.e., objects that can't be changed after their creation)
- `pandas.melt` function to pivot a dataframe from wide to long format, where one or more columns are used as identifier variables (id_vars), while all other columns, considered measured variables (value_vars), are "unpivoted" to the row axis
- pandas.DataFrame.merge function merges dataframes or series with a join on columns or indexes. If joining columns on columns, the dataframe indexes will be ignored. Otherwise, if joining indexes on indexes or indexes on a column or columns, the index will be passed on. If both key columns contain rows where the key is a null value, those rows will be matched against each other. The type of merge to be performed can be left, right, outer, inner and cross:
    - left: use only keys from left frame, preserve key order
    - right: use only keys from right frame, preserve key order
    - outer: use union of keys from both frames, sort keys lexicographically
    - inner: use intersection of keys from both frames, preserve the order of the left keys
    - cross: creates the cartesian product from both frames, preserves the order of the left keys
- `pandas.Series.cat.codes` returns a series of codes and an index for the given dataframe column
- `pandas.DataFrame.transform()` function transforms the data into a dataframe with the same shape (number of rows and columns), with the `mean` function as input to calculate the mean/average of the given data

5.4. Engineering lag and window features

There are several classes of features that can be created in order to shape our time series into a format suitable for supervised learning. Some of them are described below (Brownlee, Introduction to Time Series Forecasting with Python, 2020; Singh, 2019):

- Date-related features: components of the time itself (day of the week, month, etc.)
- Lag Features: values at prior time steps since value at time t is affected by past values a time t-1, t-2, etc. The chosen lag value depends on the correlation of individual values with past values (weekly/monthly, etc. trend). Then, e.g., a linear regression model assigns appropriate weights (coefficients) to the created lag features.
- Window Features: a summary of values over a fixed window of prior time steps.
    - Rolling window features (e.g., rolling mean, rolling sum, weighted average, etc.) are based on past values with a different window for every data point (Figure 73)
    - Expanding window features are created by taking all the past values into account since the window size increases for every new value (Figure 74)
- Domain-specific features: engineering domain-specific features for the model requires a good understanding of the problem statement, clarity of the end objective and knowledge of the available data (taking into consideration the state, store, category, events, etc.)

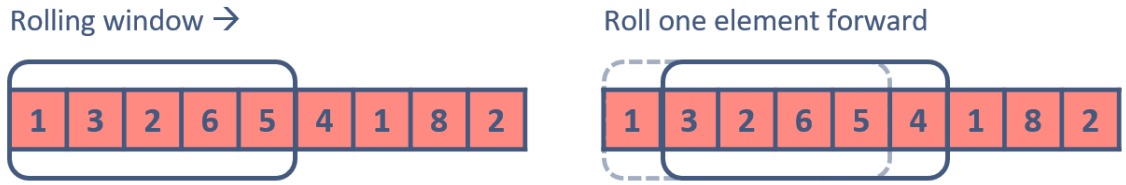


Figure 73: Rolling window illustration

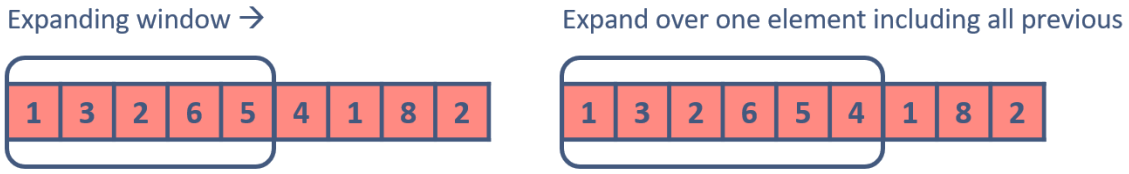


Figure 74: Expanding window illustration

The simplest approach to create lags is to predict the value at a future time (t+n) given the value at the current time (t); this is essentially a simple rolling (or sliding) window method with a window width of n. As showcased in Figure 75, shifting the dataset by 1 creates the lag_1 column, shifting it by 2 creates the lag_2 column and so on – adding a NaN (unknown) value for the first 1, 2, etc rows.

	Datetime	lag_1	lag_2	lag_3	lag_4	lag_5	lag_6	lag_7	Count
0	2012-08-25 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	8
1	2012-08-25 01:00:00	8.0	NaN	NaN	NaN	NaN	NaN	NaN	2
2	2012-08-25 02:00:00	2.0	8.0	NaN	NaN	NaN	NaN	NaN	6
3	2012-08-25 03:00:00	6.0	2.0	8.0	NaN	NaN	NaN	NaN	2
4	2012-08-25 04:00:00	2.0	6.0	2.0	8.0	NaN	NaN	NaN	2
5	2012-08-25 05:00:00	2.0	2.0	6.0	2.0	8.0	NaN	NaN	2
6	2012-08-25 06:00:00	2.0	2.0	2.0	6.0	2.0	8.0	NaN	2
7	2012-08-25 07:00:00	2.0	2.0	2.0	2.0	6.0	2.0	8.0	2
8	2012-08-25 08:00:00	2.0	2.0	2.0	2.0	2.0	6.0	2.0	6
9	2012-08-25 09:00:00	6.0	2.0	2.0	2.0	2.0	2.0	6.0	2

Figure 75: Example of lag features

The next step would be to go beyond adding the raw prior values and instead calculate some statistics (such as the mean of previous n values, the max, the sum, etc) and use that to predict the next value. It should be noted that, depending on the window width, the first rows of data will not be usable for the forecasting since they will include many NaN values.

	ID	Datetime	Count	rolling_mean
0	0	2012-08-25 00:00:00	8	NaN
1	1	2012-08-25 01:00:00	2	NaN
2	2	2012-08-25 02:00:00	6	8.0
3	3	2012-08-25 03:00:00	2	6.0
4	4	2012-08-25 04:00:00	2	6.0

	Datetime	Count	expanding_mean
0	2012-08-25 00:00:00	8	NaN
1	2012-08-25 01:00:00	2	5.000000
2	2012-08-25 02:00:00	6	5.333333
3	2012-08-25 03:00:00	2	4.500000
4	2012-08-25 04:00:00	2	4.000000
5	2012-08-25 05:00:00	2	3.666667
6	2012-08-25 06:00:00	2	3.428571
7	2012-08-25 07:00:00	2	3.250000
8	2012-08-25 08:00:00	6	3.555556
9	2012-08-25 09:00:00	2	3.400000

Figure 76: Example of rolling mean and expanding mean features

In our implementation, we created lags for 1, 2, 3, 6, 12, 24 and 36 days, and calculated each variable using the `shift()` function defining the desired number of periods to compare the correlation with the other variables.

We then calculated the weekly rolling mean of sold items, using the `rolling()` function with a 7-day window width. Finally, we calculated the expanding mean of sold items using the `expanding()` function defining the minimum number of observations at 2 days.

Since the max lag of 36 days would introduce a lot of NaN values into the dataset, to finalize the dataframe for modelling we completely removed the data corresponding to the 35 first days.

6. Modelling

We tried out some classical statistical models and some machine learning models to select the one that would produce the most accurate predictions, determined by the lowest calculated RMSE.

For each approach, we plotted the predicted vs actual sales for some randomly selected sample items to visually represent the results it would generate.

6.1. Data splitting

When building a model, it is important for it to be able perform well on new/unknown values, besides the values it knows/had been trained on; in other worlds we need the model to generalize well and not overfit on the training data. Splitting the data into training and validation sets, following a sensible strategy, is the way to achieve a good generalization performance and avoid overfitting.

The training set is the largest subset of the training data. The training set includes both the features and the target variable and is used for training the model with different model parameter settings, learning the relationships between the features and target variable along the way. Each trained model is consequently challenged with the validation set to evaluate its performance (Xu & Goodacre, 2018).

The validation is a smaller subset of the training data, including features and target variables for which the classifications are not previously known to the model. The predictions made on the validation set allow for an assessment of the accuracy of the model. These predictions are assessed against the actual values for multiple model-parameter sets, and the error calculated is used to select the optimal model-parameter set with the lowest validation error (Xu & Goodacre, 2018).

The test data are not taught/shown to the model at all, hence when it is requested to predict the target variable by only receiving the features, its performance is evaluated separately on the fully trained and configured model.

The existing data splitting methods can be roughly categorized into three different types:

- Cross-validation splits the data into several folds, the model is trained on all folds except one, which is used for testing; the process is repeated, changing the testing fold every time until all folds are used for testing, and the final performance metric is the average of the scores obtained in every repeat (e.g., K-fold cross-validation, Blocked cross-validation)

- Random sampling of a proportion of samples and their use as a validation set, training the model with the rest; the process is usually repeated many times and the final model performance is the average performance of all the iterations (e.g., Bootstrap)
- Systematic sampling of some of the most representative samples form the dataset, based on the data distribution, using the remaining samples for validation (e.g., Kennard-Stone algorithm)

In each of the abovementioned methods, there are one or two parameters that need to be optimized; e.g., the number of folds in cross-validation, the number of iterations in the bootstrap, the selection of the sample number in Kennard-Stone sampling etc.

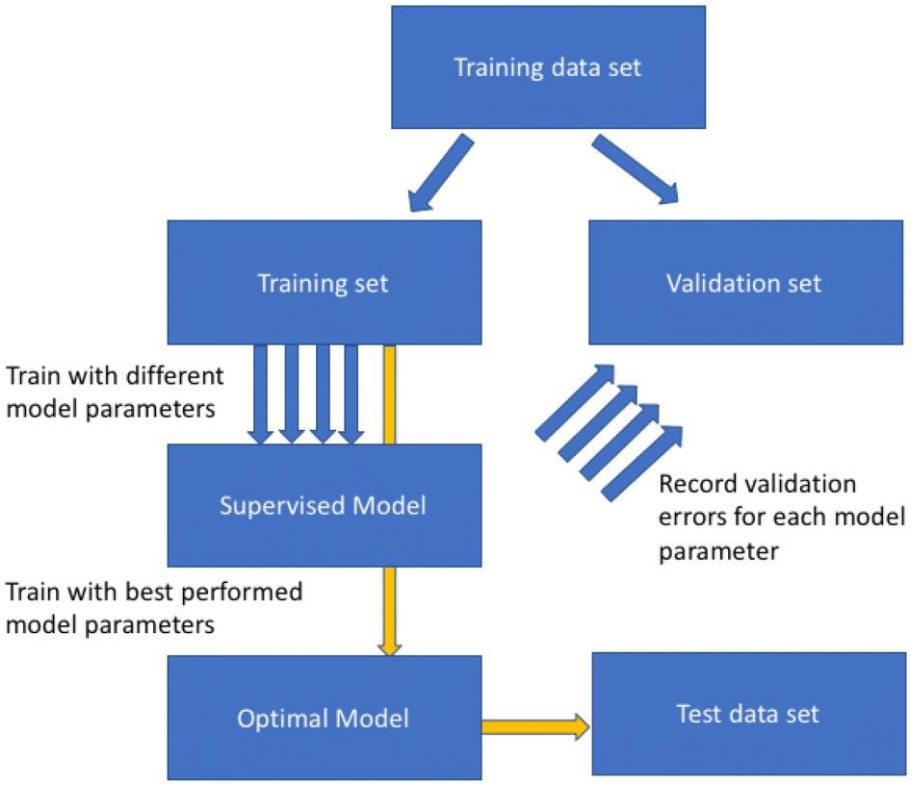


Figure 77: General flowchart used for model selection

In the figure above, the general flowchart used for model selection is illustrated; the blue arrows indicate the validation process while the yellow arrows indicate the final training and test on blind test set process. The data set is separated into a training, a validation and a test subset. The training set is used for training models with different parameters, the validation set is used for validating the error for each model-parameter set and returning the best combination to train the optimal model and use that model on the test subset.

Figure 78: Training and testing data in time series

Some final points to consider (Hyndman & Athanasopoulos, 2018):

- The test data set is typically about 20% of the total sample, although this value depends on the size of the sample and the forecast horizon required (the test data should ideally be at least as large)
- A perfect fit can always be obtained by using a model with enough parameters
- A model which fits the training data well will not necessarily forecast well
- Over-fitting a model is equally bad to failing to identify a pattern in the data

6.2. Statistical models

Linear models for forecasting have existed since the 1920s, when Yule started the notion of stochasticity in time series by assuming that every time series can be regarded as the realization of a stochastic process. This idea formed the bases of a number of time series methods developed since then, following the concept of autoregressive (AR) models and moving average (MA) models and their numerous combinations such as ARARMA, ARIMA, MARMA, SARIMAX, and countless more (Majid, 2018).

In the 1950s and 1960s, exponential smoothing methods were introduced in forecasting. They have been further studied and modified in the course of time, with several different methods existing, including simple exponential smoothing method with no trend and no seasonal component, Holt's linear method with additive trend component and no seasonal component, Holt–Winter's additive and multiplicative methods with additive trend component and additive or multiplicative seasonal component respectively, and so on (Majid, 2018; Makridakis, Spiliotis, & Assimakopoulos, 2018).

In our implementation, we are going to study the simple exponential smoothing model, the Holt linear model and the SARIMAX model, and compared them against a baseline model whose performance will be used as benchmark for efficiency.

6.2.1. Baseline

The first forecast model we are trying out is based on the very simple naïve approach. It simply forecasts the next day's sales as the current day's sales as follows:

$$\hat{y}_{t+1} = y_t$$

- $\hat{y}_{t+1}$ refers to the predicted value for the next day's sales (aka the forecast)
- y_t refers to today's sales

This method works remarkably well for many economic and financial time series.

Seasonal naïve approach is a similar method useful for highly seasonal data. In this case, we set each forecast to be equal to the last observed value from the same season (e.g., the same month of the previous year, or the same day of the previous week, etc). The forecast equals to:

$$\hat{y}_{T+h|T} = y_{T+h-m(k+1)}$$

- m refers to the seasonal period
- k is the integer part of $(h-1)/m$ (i.e., the number of complete years in the forecast period prior to time $T+h$)

For example, for monthly data the forecast for all future February values would be equal to the last observed February value, for quarterly data the forecast of all future Q2 values would be equal to the last observed Q2 value, etc (Pawar, 2020).

Figure 79: Seasonal naïve approach sample results

In our implementation, we used the seasonal naïve method with a yearly seasonality, i.e., assigning each forecasted day the sales value from the same day in the previous year. We then plotted the results for some randomly selected items to see the predicted (red line) vs the actual (blue line) values (Figure 79).

Finally, we calculated the root mean squared error (RMSE) for the validation dataset using the `mean_squared_error` function specifying the `squared` parameter as `False` so that the RMSE is returned. The RMSE for this seasonal naïve approach was 3.22, and it will be used as the performance baseline/benchmark for our forecasting, i.e., a point of reference for the performance of the other models we built.

6.2.2. Exponential smoothing

Exponential smoothing produces forecasts that are weighted averages of past observations, with the weights decaying exponentially as we move further backwards in time. In other words, the more recent the observation the higher the associated weight.

The model can be summarized as follows:

$$\hat{y}_{t+1} = a \cdot y_t + (1 - a) \cdot \hat{y}_t$$

- $\hat{y}_{t+1}$ refers to the calculated forecast as the weighted average of all the observations in the series $y_1, \ldots, y_t$
- a refers to the smoothing parameter which controls the rate of the weights' decay, giving different weightage to different time steps, which ensures that more recent sales data are given more importance compared to older ones

In our implementation, we used the `ExponentialSmoothing` class (from `statsmodels.tsa.holtwinters`) to create the model, specifying 7 as the number of periods in a complete seasonal cycle (corresponding to daily data with a weekly cycle) and both trend and season being of additive type (meaning that their variations are relatively constant, following a linear curve).

We used a sample of 1,000 items to fit and train the model and produced the forecasts on those. In Figure 80 we can see the plotted results for some of these items, actual value is the blue and predicted value is the red line, and we can see that Exponential Smoothing predicts the mean sales with accuracy; however, it generates a rather flattened result due to the low weightage given to older time steps.

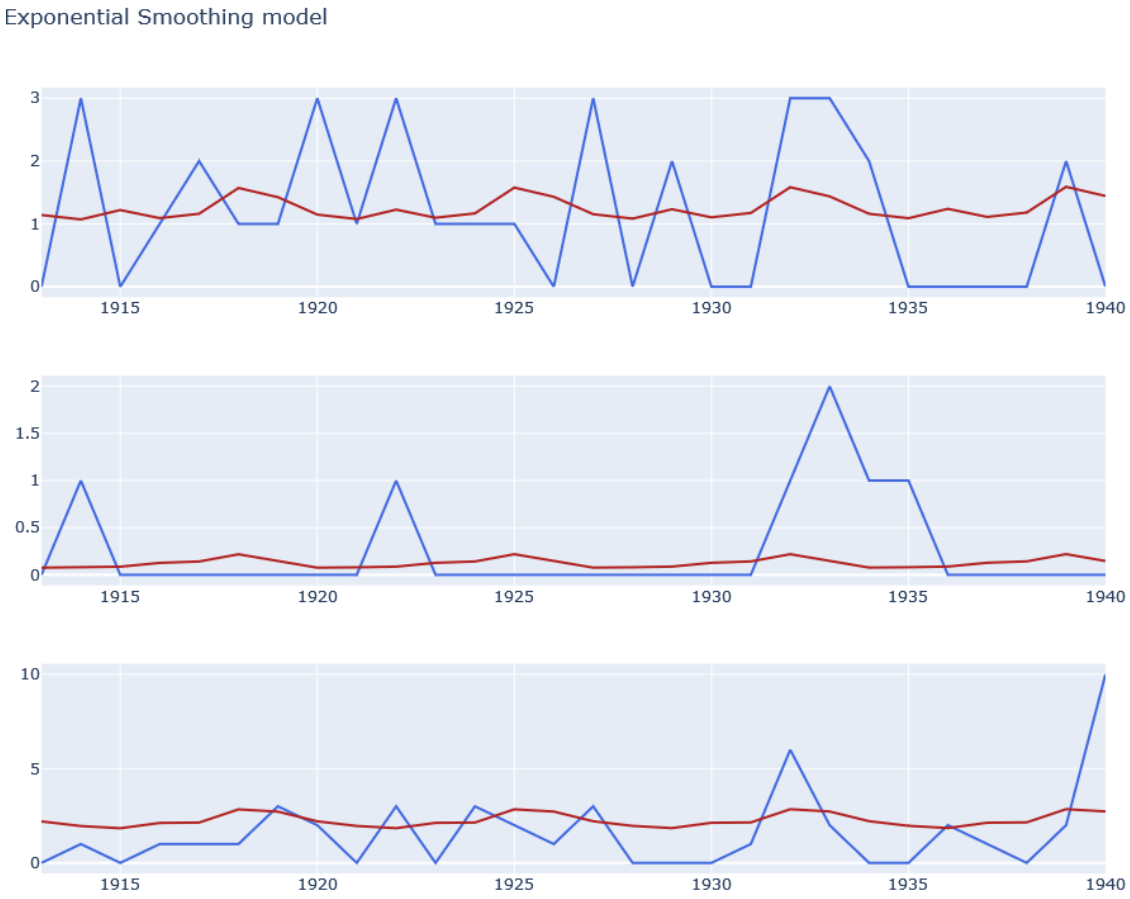


Figure 80: Exponential Smoothing model sample results

Finally, we calculated the root mean squared error (RMSE) for the validation dataset which was 1.95 for this Exponential Smoothing model, better than the baseline approach by 40%.

6.2.3. Holt linear

Holt linear, which is also called double exponential smoothing or Holt's linear trend, attempts to capture the high-level trends in time series data by adding a second exponential model resulting in forecasts with a linear upward or downward trend. The method can be summarized as follows (Hyndman & Athanasopoulos, 2018):

- Forecast equation $\hat{y}_{t+h} = l_t + h \cdot b_t$
- Level equation $l_t = a \cdot y_t + (1 - a) \cdot (l_{t-1} + b_{t-1})$
- Trend equation $b_t = \beta \cdot (l_t - l_{t-1}) + (1 - \beta) \cdot b_{t-1}$

In the above equation, the values l_t and b_t represent the level and trend values respectively at time t. Meanwhile, a and β are constants which can be configured; a is

the smoothing parameter for the level, and β is the smoothing parameter for the trend or slope; they both take values between 0 and 1.

In our implementation, we used the `Holt` class for Holt's Exponential Smoothing (from `statsmodels.tsa.holtwinters`) to create the model, and fitted it with the `fit` function – specifying the smoothing level (α value) at 0.3 and the smoothing slope (β value) at 0.01.

As illustrated by the sample items in Figure 81, our Holt linear model was able to predict high-level trends in the sales creating an upward or downward line, but it did not seem to capture the short-term volatility in the daily sales.

Figure 81: Holt linear model sample results

Finally, we calculated the root mean squared error (RMSE) for the validation dataset which was 2.34 for our Holt Linear model, 27% better than the baseline approach.

6.2.4. SARIMAX

The important assumption of time series forecasting is that the current demand is a function of the past sales. However, as we saw in the EDA section, the past sales are

influenced by many external factors such as trend, seasonality, price, promotions, events, etc. The traditional SARIMA (**S**easonal **A**uto**r**egressive **I**ntegrated **M**oving **A**verage) forecasting model takes into consideration trend and seasonality when describing the correlations in the time series, but it does not incorporate the influence of the other external factors. To overcome this disadvantage, the SARIMAX (SARIMA e**X**ogenous) model is introduced improving the forecast accuracy.

Improving the forecast accuracy is important to reduce understocking and overstocking. Understocking leads to products being out of stock, as well as lower customer confidence and deterioration of market image which are difficult to quantify. Overstocking leads to insufficient shelf space, shrinkage, and food waste, especially in perishable foods (Arunraj, Ahrens, & Fernandes, 2016).

To determine the requirement of differencing, the stationarity of the time series must be checked using the Augmented Dickey-Fuller test (aka ADF or adfuller test). Stationary time series are those whose properties do not depend on the time at which the series is observed, having no predictable patterns in the long term. Differencing is a way to make a non-stationary time series stationary by computing the differences between consecutive observations. Differencing can help stabilise the mean of a time series by removing changes in the level of a time series, and therefore eliminating (or reducing) trend and seasonality (Hyndman & Athanasopoulos, 2018; Verma, 2021).

The null hypothesis of the adfuller test is that there is a unit root, with the alternative that there is no unit root. If the test statistics is more than 5% of the critical value and the p-value is larger than 0.05 that means that the moving average is not constant over time and the null hypothesis of the adfuller test cannot be rejected. If the test statistic is less than 1% of the critical value and the p-value is 0.01, we can say that the time series is stationary with 99% confidence.

We performed the adfuller test on the time-series as a sum of all items sold per day, and for some randomly selected item.

From the results of the adfuller test on the daily_sales dataframe (Figure 82), we can see that the p-value is higher than 5% which means that the evidence of the null hypothesis is low; hence the time series is non-stationary.

```
# Performing the ADF test on the daily_sales dataset (sum of all items sold per day)

from statsmodels.tsa.stattools import adfuller
adf_sum = adfuller(daily_sales, autolag = 'AIC')
print("1. ADF : ",adf_sum[0])
print("2. P-Value : ", adf_sum[1])
print("3. Num Of Lags : ", adf_sum[2])
print("4. Num Of Observations Used For ADF Regression and Critical Values Calculation :", adf_sum[3])
print("5. Critical Values :")
for key, val in dftest[4].items():
    print("\t",key, ": ", val)
```

```
1. ADF :  -1.8124676284926875
2. P-Value :  0.3742896656883241
3. Num Of Lags :  26
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 1942
5. Critical Values :
         1% :  -3.433721763505903
         5% :  -2.8630294391572706
         10% :  -2.567562917840866
```

Figure 82: Adfuller test on daily_sales (sum of all items sold per day)

The same results are observed from the adfuller test on the time series of the items sold per category (Figure 83), where we can also see that the p-value is higher than 5%, indicating that the three time-series are non-stationary.

```
# Performing the ADF test on the three categories time series (sum of items sold per category per day)

adf_foods = adfuller(df_categories.FOODS, autolag='AIC')
adf_hobbies = adfuller(df_categories.HOBBIES, autolag='AIC')
adf_household = adfuller(df_categories.HOUSEHOLD, autolag='AIC')

print("p-value of FOODS time serie is: {}".format(float(adf_foods[1])))
print("p-value of HOBBIES time serie is: {}".format(float(adf_hobbies[1])))
print("p-value of HOUSEHOLD time serie is: {}".format(float(adf_household[1])))
```

```
p-value of FOODS time serie is: 0.9680368207260177
p-value of HOBBIES time serie is: 0.2639990369374694
p-value of HOUSEHOLD time serie is: 0.3742896656883241
```

Figure 83: Adfuller test on items sold per category

Moving on, in our implementation, we fitted the model into the time series using the `SARIMAX` class (from `statsmodels.tsa.statespace`) and specifying the seasonal order. The fitting was completed in approximately 30 minutes. Then we used the `forecast` function to predict the values for days 1914-1941 with the fitted model, and then plotted some randomly selected items to see the results of the predicted vs the actual values.

As we can see in Figure 84, where the blue line represents the actual value and the red line the predicted value, the SARIMAX model was able to find both low-level and high-level trends and it was able to predict a rather accurate periodic function for each of the examined items.

Figure 84: SARIMAX model sample results

Finally, we calculated the root mean squared error (RMSE) for the validation dataset which was 2.48 for our SARIMAX model, 23% better compared to the baseline approach.

6.3. Machine learning models

Machine learning models have established themselves as serious contenders to classical statistical models in the field of forecasting and they can be exploited to improve time series predictions (Ahmed, Atiya, El Gayar, & El-Shishiny, 2010).

Both the ML methods and the statistical ones aim at improving the accuracy of the forecasts by minimizing some loss function, such as the mean of the squared errors. Their main difference is that while in statistical methods the minimization is achieved with linear processes, in ML methods non-linear algorithms are utilized (Makridakis, Spiliotis, & Assimakopoulos, 2018).

Before moving forward, we prepared the datasets that would be used for testing (isolated the id, d, sold values on days 1914-1941) and validation (isolated the id, d, sold values on days 1942-1969) and for storing the predictions (sold values) during modelling. We also

prepared the data for plotting the sample results, isolating the actual data to be shown alongside the predicted values for the sample items.

6.3.1. XGBoost

Gradient boosting refers to a class of ensemble machine learning algorithms that can be used for classification or regression predictive modelling problems. Ensembles are constructed by successively adding decision trees, while fitting to correct the prediction errors made in prior iterations. Any arbitrary differentiable loss function and gradient descent optimization algorithm can be used for fitting, giving the method the name "gradient boosting" since the loss gradient is minimized as the model is being fit (Brownlee, 2021).

XGBoost (e**X**treme **G**radient **Boost**ing) is a highly scalable, flexible and versatile model; it was engineered to optimally exploit resources and to overcome the limitations of the previous gradient boosting methods with the main difference being its use of a new regularization technique to control overfitting which makes it faster and more robust during the model tuning (Al Daoud, 2019).

The decision trees in XGBoost are grown level-wise, checking all of the previous leaves for each new leaf, as illustrated in the figure below.

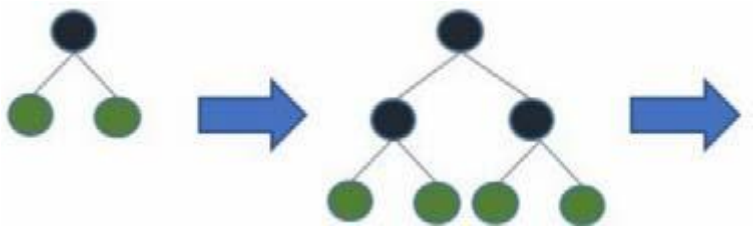


Figure 85: XGBoost Level-wise tree growth

XGBoost requires that the time series dataset is transformed into a supervised learning problem and a specialized technique called walk-forward validation is used for evaluating the model; evaluating the model using k-fold cross validation which randomizes the dataset would not work because the model must always be trained on the past to predict the future.

Formula for computation of predictions:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in F$$

Where estimation $\hat{y}_i$ is the prediction, x_i is a vector of features, $f_k(x_i)$ are the values computed for each tree, and K is the total number of trees.

In our implementation, we used a separate model for each of the 10 stores. For each store, we first split the data into train, validation and test subsets:

- The train subset included values for days 0-1913 (all features included in `X_train` and sales assigned to `y_train`)
- The validation subset included values for days 1914-1941 (all features included in `X_val` and sales assigned to `y_val`)
- The test subset included only the features for 1942-1969 (included in `X_test`)

The regression model was created using the `XGBRegressor` class from the `xgboost` library, with the following hyperparameters configurations:

- `eta` which controls the learning rate used to weight each model, with typical final values: 0.01, 0.2, 0.3 (too high and it will be difficult to converge, too low and it will take more boosting rounds) was set to 0.3
- `min_child_weight` which corresponds to a minimum number of instances needed in each child/node, with typical values such as 1, 5, 6, 10 (a smaller weight leads to a more complex tree with child nodes with fewer samples, a larger weight leads to a more conservative algorithm), was set to 1
- `max_depth` which corresponds to the maximum tree depth, with typical values between 3-10 (the deeper the tree, the more complex the model but also more prone to overfitting), was set to 6

The model was fitted on the training data for days 0-1913 and evaluated on the validation data for days 1914-1941, with RMSE used for the evaluation of the model's performance and `early_stopping_rounds` set to 20 so that the model training stops at the 20th iteration even if the validation error keeps decreasing.

Eventually, the predictions from each store were combined and saved within a unified dataframe.

Our XGBoost model was quite successful in predicting the future values with the sample results, as showcased in Figure 86, being a bit less accurate for higher values.

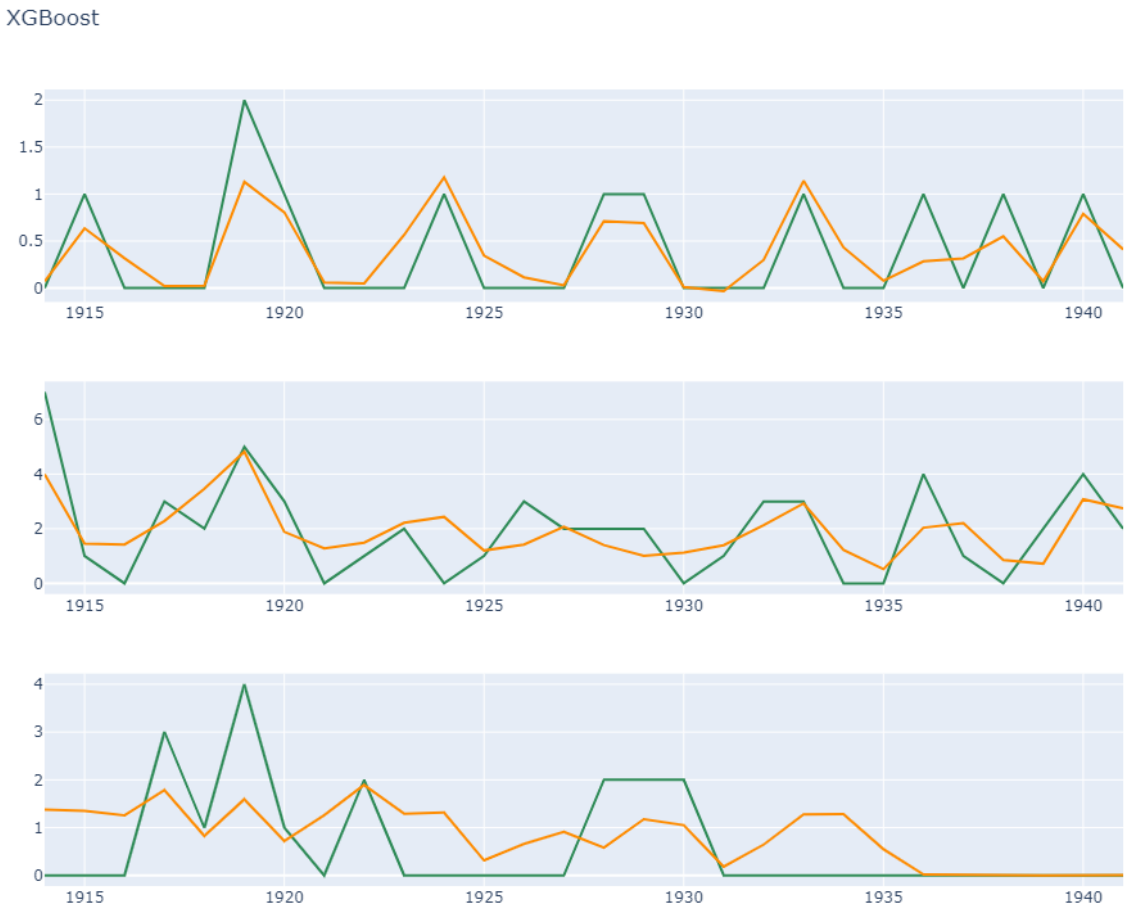


Figure 86: XGBoost model sample results

Finally, we calculated the root mean squared error (RMSE) for the validation dataset which was 1.68 for our XGBoost model, 48% better compared to the baseline approach.

6.3.1. LightGBM

LightGBM is short for Light **G**radient **B**oosted **M**achine, a library developed by a Microsoft team in April 2017 with the goal of reducing the implementation time. The primary benefit of the LightGBM is a more efficient implementation of the training gradient boosting algorithm that make the process significantly faster (Al Daoud, 2019).

The main difference with XGBoost is that the decision trees in LightGBM are grown leaf-wise, instead of checking all of the previous leaves for each new leaf, and all attributes are sorted and grouped as bins. This implementation is called histogram implementation.

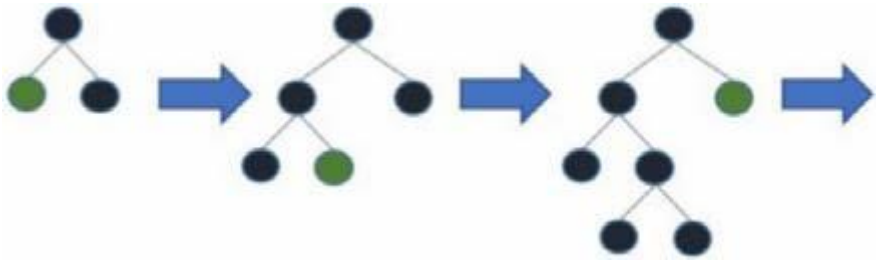


Figure 87: LightGBM Leaf-wise tree growth

In our implementation, similarly to the XGBoost model, we created a separate model for each of the 10 stores. For each store, we first split the data into train, validation and test subsets:

- The train subset included values for days 0-1913 (all features included in `X_train` and sales assigned to `y_train`)
- The validation subset included values for days 1914-1941 (all features included in `X_val` and sales assigned to `y_val`)
- The test subset included only the features for 1942-1969 (included in `X_test`)

Our model was created using the `lgbm.LGBMRegressor` class, with the following hyperparameters configurations:

- `n_estimators` referring to the number of boosted trees to fit, was set to 1000
- `learning_rate` was set to 0.3
- `subsample` which refers to the number of samples for constructing bins, was set to 0.8
- `colsample_bytree` which refers to the subsample ratio of columns when constructing each tree, was set to 0.8
- `max_depth` which refers to the maximum tree depth for base learners (if <=0 means there is no limit), was set to 8
- `num_leaves` which refers to the maximum tree leaves, was set to 50
- `min_child_weight` which refers to the minimum sum of instance weight needed in a child/leaf, was set to 300

The model was fitted on the training data for days 0-1913 and evaluated on the validation data for days 1914-1941, with RMSE used for the evaluation of the model's performance and `early_stopping_rounds` set to 20 so that the model training stops at the 20th iteration even if the validation error keeps decreasing.

Eventually, the predictions from each store were combined and saved within a unified dataframe.

The results of our LightGBM model, illustrated in Figure 88 for some sample items, indicate that it has high accuracy in predicting future values, capturing both the high and low-level trends.

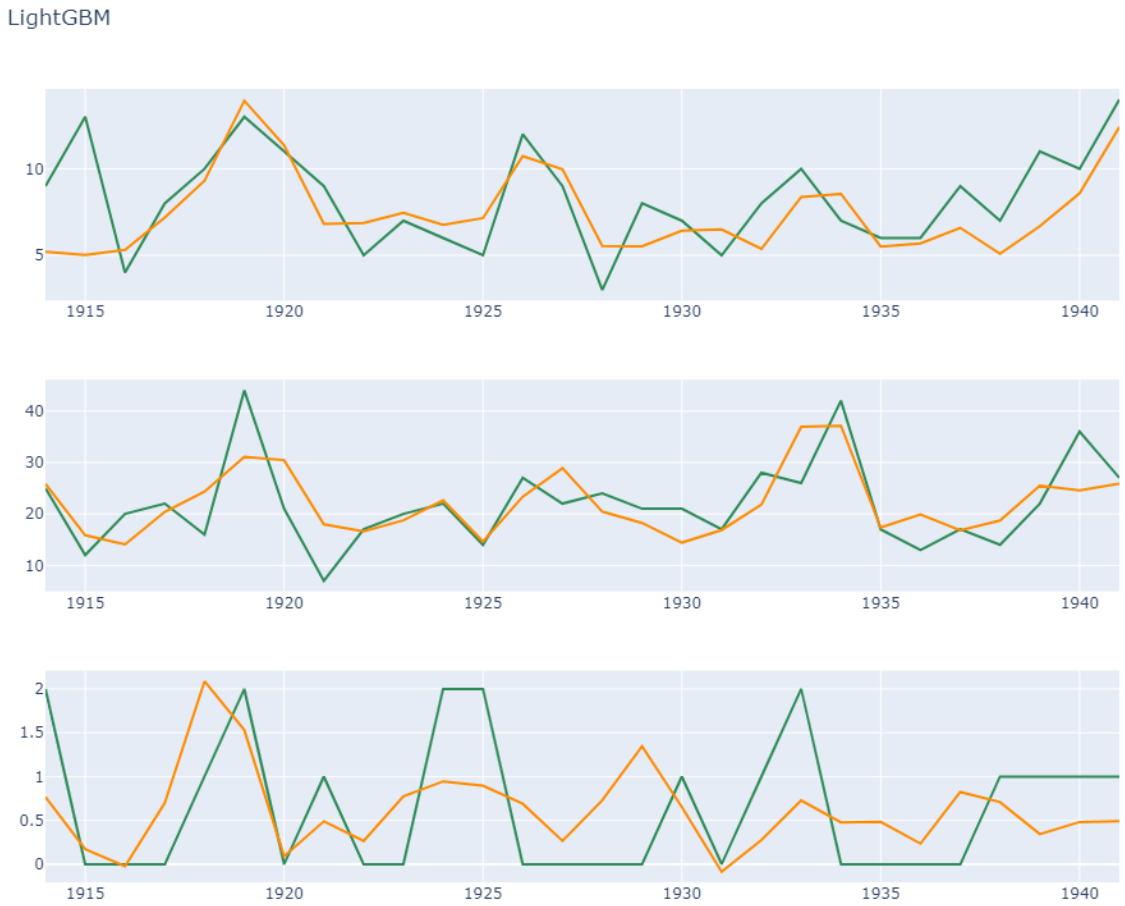


Figure 88: LightGBM model sample results

Finally, we calculated the root mean squared error (RMSE) for the validation dataset which was 1.55 for our LightGBM model, 52% better compared to the baseline approach making it the best performing model amongst the examined ones.

6.4. Evaluation of models

We have used RMSE (**R**oot **M**ean **S**quare **E**rror) as the performance measure for the methods used to predict the future daily sales of the items. RMSE is the standard deviation of the residuals (i.e., prediction errors); residuals measure the distance of the data points from the regression line and RMSE indicates how spread out these residuals are. RMSE

as a performance metric verifying experimental result is commonly used in climatology, forecasting, and regression analysis.

The metric was computed for each time series and then averaged across all time-series, the results for each model are summarized in Figure 89, showing that all examined models outperformed the baseline approach with LightGBM being the winner solution.

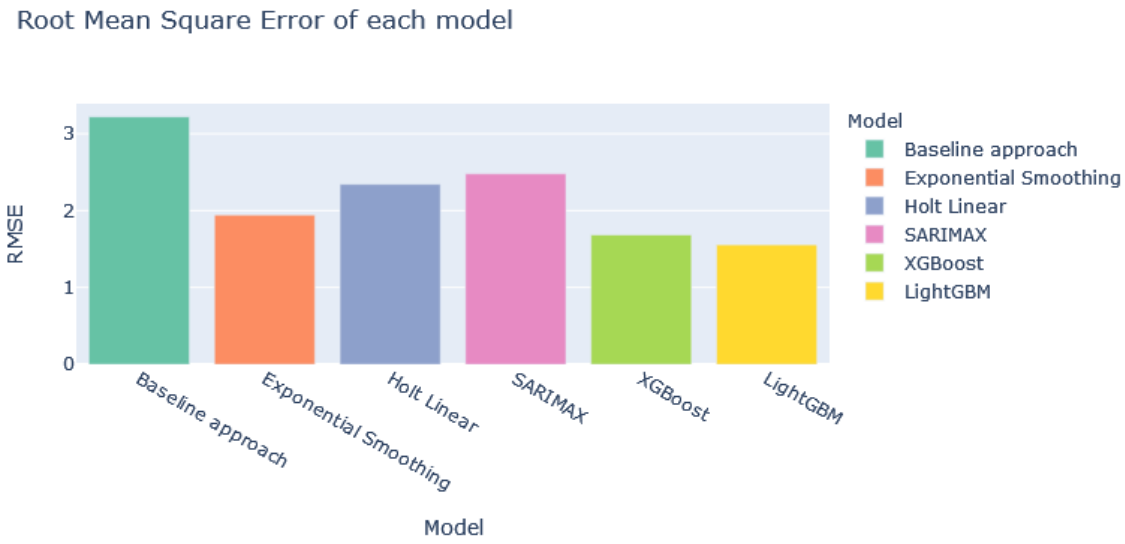


Figure 89: RMSE of each model

LightGBM was also the method featured in all but one of the top 5 solutions in the competition results. Some possible reasons for the superior performance of LightGBM (Kenworthy, 2021):

- LightGBM does not build its trees level-by-level, but leaf-by-leaf. This can lead to overfitting when the data is sparse. We can use a parameter (maximum leaf depth) to limit this.
- LightGBM determines the split point for tree nodes by using histogram algorithms instead of pre-sorted feature values. A search for the splits is efficient as the number of bins is less than the number of values. XGBoost uses a histogram algorithm too, but with LightGBM this process is optimized during training by weighting the model instances in favour of those that are under trained.
- If features are mutually exclusive (they are neither simultaneously trivial or non-trivial) they can be combined in a single feature. LightGBM does this to reduce the number of overall features which also aids training.

7. Summary

This research aimed to review the M5 competition and by implementing exploratory data analysis, engineering features derived from the data, and eventually building models for time series forecasting with both statistical and machine learning methods.

In order to understand the context of the M competitions, previous iterations were examined to gain some insights into the historic precedence of time series forecasting. So far, the M5 competition had the highest combination of both time series and explanatory features, and it was also the first one that was held on the Kaggle platform resulting in an unprecedented volume of participants, when comparing to the previous iterations. It was also the first iteration to declare a pure Machine learning solution as the winner, while the previous were dominated by either classical statistical or hybrid solutions.

As there were numerous kernels in Kaggle presenting various solutions and analysis approaches, we examined some of those that we singled out due to their exquisite appearance and comprehensive presentation of the analytical approach or solution, acknowledging their influence and guidance in our implementation; specifically, 6 notebooks were selected, most of which started with an EDA, included some feature engineering approach, and offered a solution to our time series forecasting problem. We discussed the methodology followed in each case and highlighted some of the most innovative plots.

We conducted an extensive Exploratory Data Analysis, after first introducing the main definitions for the concept, and presenting several suggested approaches for the process and some of the most commonly used plot types. Our missing and null values analysis showed that the sales and prices dataframes had many null values due to many products being introduced in later years and hence having zero sales in the beginning of the examined period, while the calendar dataframe had many missing values due to the majority of the days not having any events.

From the decomposition analysis, using both additive and multiplicative models, we gained some insights on the level, trend, seasonality and noise/residual of the time-series; we established that sales have been on an overall increasing trend throughout the years, with strong seasonality being present alongside a high level of randomness (i.e., residual noise).

We inspected the time series on several levels of aggregation, and from different points of view, observing some differences in the trends and behaviour amongst categories,

stores, states etc with some exhibiting more clear patterns throughout the examined period. We also analysed the different explanatory variables, gaining some indicators of the impact of the different events and SNAP days onto the performance, mostly affecting it positively. From our exploration of the product prices, we deducted that most products have relatively stable prices throughout the examined years with many items not exhibiting any price change at all, or a very low one, especially in the Foods product category.

Feature Engineering followed EDA, an essential step before modelling. As the first step, we managed to significantly reduce the memory usage by applying some transformation via the established downcasting process to the datatype of our data, bringing it down by as much as almost 80%. We label encoded all categorical values, in order to bring the data into an ML-friendly format, and introduced some mean encoding features based on some selected groups of variables, all with the purpose of increasing the quality of the forecast. Finally, by introducing lag and window features (rolling and expanding mean), we managed to shape our time series into a format suitable for machine learning modelling.

All of the statistical and machine learning models that we implemented performed better than the benchmark of the seasonal Naïve baseline approach we calculated. The simple Exponential smoothing was the best performing amongst the statistical models, outperforming the more complex Holt linear and SARIMAX methods. At the same time, the two gradient boosting models we implemented, performed better than all statistical ones, with LightGBM taking the lead mostly due to its ability to avoid overfitting and simplicity compared to XGBoost.

References

Ahmed, N. K., Atiya, A. F., El Gayar, N., & El-Shishiny, H. (2010). An Empirical Comparison of Machine Learning Models for Time Series Forecasting. *Econometric Reviews*, 594-621. doi:https://doi.org/10.1080/07474938.2010.481556

Al Daoud, E. (2019). Comparison between XGBoost, LightGBM and CatBoost Using a Home Credit Dataset. *International Journal of Computer and Information Engineering, 145*, 6-10.

Arunraj, N. S., Ahrens, D., & Fernandes, M. (2016). Application of SARIMAX Model to Forecast Daily Sales in Food Retail Industry. *International Journal of Operations Research and Information Systems*, 1-21.

Bezerra, A., Silva, I., Guedes, L. A., Silva, D., Leitão, G., & Saito, K. (2019). Extracting Value from Industrial Alarms and Events: A Data-Driven Approach Based on Exploratory Data Analysis. *Sensors, 19*(12). doi:https://doi.org/10.3390/s19122772

Brittain, J., Cendon, M., Nizzi, J., & Pleis, J. (2018). Data Scientist's Analysis Toolbox: Comparison of Python, R, and SAS Performance. *SMU Data Science Review*, Vol. 1 : No. 2 , Article 7.

Brownlee, J. (2020). *Introduction to Time Series Forecasting with Python*. Machine Learning Mastery: eBook.

Brownlee, J. (2021, March). *XGBoost for Regression*. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/xgboost-for-regression/

Chen, J. (2021, August 24). *Logarithmic Price Scale*. Retrieved from Investopedia: https://www.investopedia.com/terms/l/logarithmicscale.asp

Chug, A. (2021, September). *Label Encoding of datasets in Python*. Retrieved from Geeks for Geeks: https://www.geeksforgeeks.org/ml-label-encoding-of-datasets-in-python/

Cox, V. (2017). *Translating Statistics to Make Decisions: A Guide for the Non-Statistician.* Salisbury, United Kingdom: Apress.

Ghosh, A., Nashaat, M., Miller, J., Quader, S., & Marston, C. (2018). A comprehensive review of tools for exploratory analysis of tabular industrial datasets. *Visual Informatics, Volume 2*(Issue 4), 235-253. doi:https://doi.org/10.1016/j.visinf.2018.12.004

Greate Learning Team. (2021, December). *Label Encoding in Python Explained*. Retrieved from Great Learning: https://www.mygreatlearning.com/blog/label-encoding-in-python

Hyndman, R. J., & Athanasopoulos, G. (2018). *Forecasting: principles and practice.* Melbourne, Australia: OTexts. Retrieved from https://otexts.com/fpp2/

Jain, A. (2016, March). *XGBoost Parameter Tuning*. Retrieved from Analytics Vidhya: https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

Juggins, S., & Telford, R. (2012). Exploratory Data Analysis and Data Display. In H. J. Birks, A. F. Lotter, S. Juggins, & J. P. Smol, *Tracking Environmental Change Using Lake Sediments* (pp. 123-141). Dordrecht: Springer.

Kenworthy, A. (2021, March). *A review of the M5 time-series forecasting competition*. Retrieved from Balluff blog - Innovating Automation: https://www.innovating-automation.blog/m5-forecasting-competition/

Kolassa, S. (2016). Evaluating predictive count data distributions in retail sales forecasting. *International Journal of Forecasting*, 788-803.

Laerd Statistics team. (n.d.). *Types of Variable*. Retrieved from Laerd Statistics: https://statistics.laerd.com/statistical-guides/types-of-variable.php

Majid, R. (2018). Advances in Statistical Forecasting Methods: An Overview. *Economic Affairs, Vol. 63*(No. 4), 815-831. doi:https://doi.org/10.30954/0424-2513.4.2018.5

Makridakis Competitions. (n.d.). *History of Competitions - MOFC*. Retrieved from MOFC Web site: https://mofc.unic.ac.cy/history-of-competitions/

Makridakis, S. (2020, 03 10). *The M5 Competition - MOFC.* Retrieved from MOFC - Forecact | Compete | Excel: https://mofc.unic.ac.cy/m5-competition/

Makridakis, S., & Hibon, M. (2000). The M3-Competition: results, conclusions and implications. *International Journal of Forecasting*, 451-476.

Makridakis, S., Andersen, A., Carbone, R., Fildes, R., Hibon, M., Lewandowski, R., . . . Winkler, R. (1982). The accuracy of extrapolation (time series) methods: Results of a forecasting competition. *Journal of Forecasting*, 111-153.

Makridakis, S., Chatfield, C., Hibon, M., Lawrence, M., Mills, T., Ord, K., & Simmons, L. F. (1993). The M2-competition: A real-time judgmentally based forecasting study. *International Journal of Forecasting*, 5-22.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018). Statistical and Machine Learning forecasting methods: Concerns and ways forward. *PLOS ONE*. doi:https://doi.org/10.1371/journal.pone.0194889

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2020). The M4 Competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, 54-74.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2020, 10). The M5 Accuracy competition: Results, findings and conclusions. *ResearchGate (preprint)*. Retrieved from https://www.researchgate.net/publication/344487258_The_M5_Accuracy_competition_Results_findings_and_conclusions

Milo, T., & Somech, A. (2020). Automating Exploratory Data Analysis via Machine Learning: An Overview. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (pp. 2617–2622). New York, NY, USA: Association for Computing Machinery. doi:https://doi.org/10.1145/3318464.3383126

Monteiro, M. J. (2018, October). *Why you should try Mean Encoding*. Retrieved from Towards Data Science: https://towardsdatascience.com/why-you-should-try-mean-encoding-17057262cd0

Morgenthaler, S. (2009). Exploratory data analysis. *WIREs Comp Stat*, 33-44. doi:https://doi.org/10.1002/wics.2

Morgenthaler, S. (2009). Exploratory data analysis. *WIREs Comp Stat*, 33-44. doi:https://doi.org/10.1002/wics.2

Notebook by Anirban Sen. (2020, May). *M5 Forecasting Exhaustive EDA Beginner*. Retrieved from Kaggle: https://www.kaggle.com/anirbansen3027/m5-forecasting-exhaustive-eda-beginner

Notebook by Anshul Sharma. (2020, September). *Time Series Forecasting-EDA, FE & Modelling*. Retrieved from Kaggle: https://www.kaggle.com/anshuls235/time-series-forecasting-eda-fe-modelling

Notebook by Heads or Tails. (2020, October). *Back to (predict) the future - Interactive M5 EDA*. Retrieved from Kaggle: https://www.kaggle.com/headsortails/back-to-predict-the-future-interactive-m5-eda

Notebook by Jestelrod. (2020, May). *M5: EDA (Plotly) + LSTM Neural Network Vs. XGBoost*. Retrieved from Kaggle: https://www.kaggle.com/jestelrod/m5-eda-plotly-lstm-neural-network-vs-xgboost

Notebook by Konstantin Yakovlev. (2020, April). *M5 - Custom features*. Retrieved from Kaggle: https://www.kaggle.com/kyakovlev/m5-custom-features

Notebook by Konstantin Yakovlev. (2020, April). *M5 - Lags features*. Retrieved from Kaggle: https://www.kaggle.com/kyakovlev/m5-lags-features

Notebook by Konstantin Yakovlev. (2020, April). *M5 - Three shades of Dark: Darker magic*. Retrieved from Kaggle: https://www.kaggle.com/kyakovlev/m5-three-shades-of-dark-darker-magic

Notebook by Marisaka Mozz. (2020, October). *M5 Exponential Smoothing*. Retrieved from Kaggle: https://www.kaggle.com/marisakamozz/m5-exponential-smoothing

Notebook by nokin. (2020, May). *M5: Analysis by departments and stores (EDA)*. Retrieved from Kaggle: https://www.kaggle.com/one1111/m5-analysis-by-departments-and-stores-eda

Notebook by Rob Mulla. (2020, March). *M5 Forecasting - Starter Data Exploration*. Retrieved from Kaggle: https://www.kaggle.com/robikscube/m5-forecasting-starter-data-exploration

Notebook by Tarun Paparaju. (2020, March). *M5 Competition : EDA + Models*. Retrieved from Kaggle: https://www.kaggle.com/tarunpaparaju/m5-competition-eda-models/notebook

NumPy v1.22 Manual. (n.d.). Retrieved September 2021, from NumPy Web site: https://numpy.org/doc/stable/reference/generated/numpy.log1p.html

Paruchuri, V. (2020, October 21). *R vs Python for Data Analysis — An Objective Comparison*. Retrieved from Dataquest: https://www.dataquest.io/blog/python-vs-r/

Pawar, S. (2020, April). *Time series Forecasting in Python & R, Part 2 (Forecasting)*. Retrieved from Sandeep Pawar's blog: https://pawarbi.github.io/blog/forecasting/r/python/rpy2/altair/fbprophet/ensemble_forecast/uncertainty/simulation/2020/04/21/timeseries-part2.html

Rahmany, M., Mohd Zin, A., & Sundararajan, E. A. (2020). Comparing Tools Provided by Python and R for Exploratory Data Analysis. *International Journal Information System and Computer*, 131-142.

Ribecca, S. (n.d.). Retrieved September 2021, from The Data Visualisation Catalogue: https://datavizcatalogue.com/

Singh, A. (2019, December). *Feature Engineering Techniques For Time Series*. Retrieved from Analytics Vidhya:

https://www.analyticsvidhya.com/blog/2019/12/6-powerful-feature-engineering-techniques-time-series/

Tufféry, S. (2011). *Data Mining and Statistics for Decision Making.* Chichester, UK: Wiley.

UNSW online. (2020, January). *Types of data & the scales of measurement*. Retrieved from UNSW online: https://studyonline.unsw.edu.au/blog/types-of-data

Verma, Y. (2021, July). *Complete Guide To SARIMAX in Python for Time Series Modeling.* Retrieved from Analytics India Magazine: https://analyticsindiamag.com/complete-guide-to-sarimax-in-python-for-time-series-modeling/

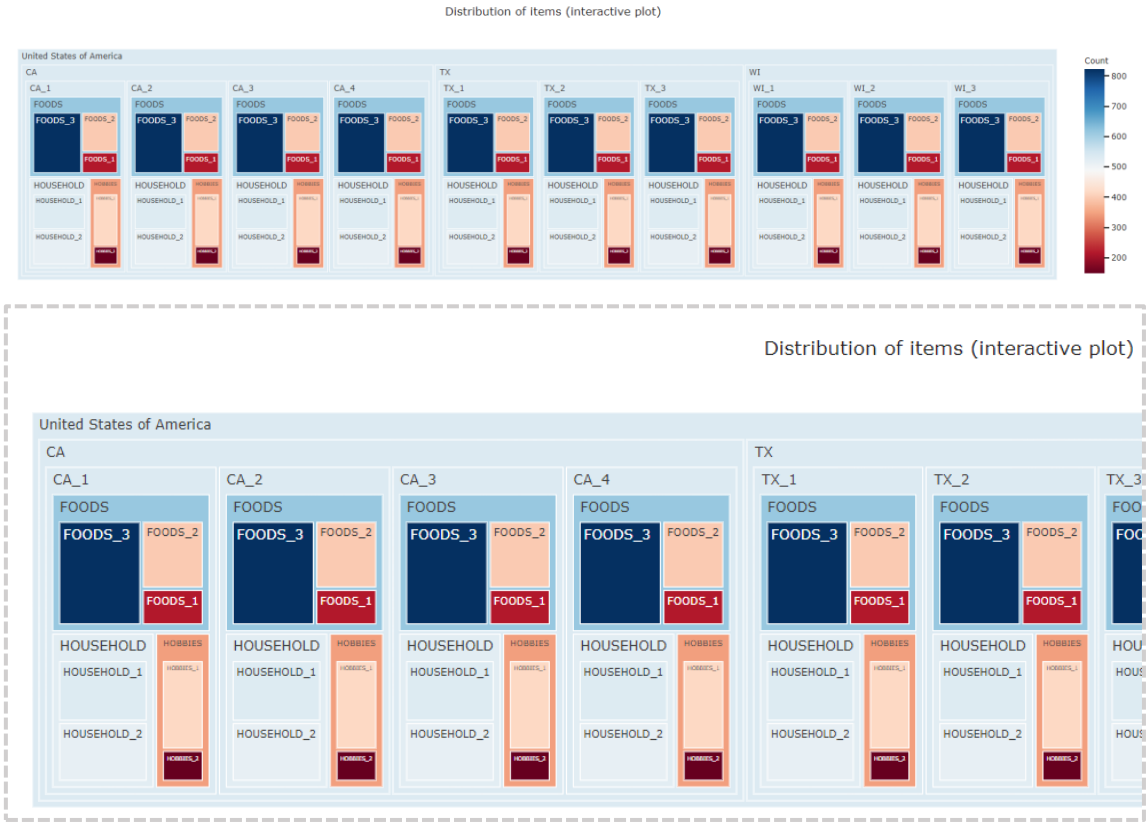
What is SNAP and How to Apply. (n.d.). Retrieved September 2021, from Feeding America Web site: https://www.feedingamerica.org/our-work/hunger-relief-programs/snap

Xu, Y., & Goodacre, R. (2018). On Splitting Training and Validation Set: A Comparative Study of Cross-Validation, Bootstrap and Systematic Sampling for Estimating the Generalization Performance of Supervised Learning. *Journal of Analysis and Testing*, 249–262. doi:https://doi.org/10.1007/s41664-018-0068-2

Young, F. W., Faldowski, R. A., & McFarlane, M. M. (1993). Multivariate statistical visualization. *Handbook of Statistics, vol. 9 Computational Statistics*, pp. 959-998.

Appendix

A1. An interactive plot of items distribution



A2. Code in Python – Jupyter notebook

The export to PDF from Jupyter notebook resulted in 77 pages which are appended here.

Time Series Forecasting: EDA and Modeling

Contents of the notebook

Analysis Section

Solution Section

1 Acknowledgments

- Back to (predict) the future - Interactive M5 EDA
- M5 Competition : EDA + Models
- M5: EDA (Plotly) + LSTM Neural Network Vs. XGBoost
- M5 Forecasting Exhaustive EDA Beginner
- M5 Forecasting - Starter Data Exploration
- Time Series Forecasting-EDA, FE & Modelling
- Time series analysis in Python
- M5: Exponential Smoothing
- Time Series Tutorial

2 Preparations

Go to contents

2.1 Import libraries

In [2]:
```
# Importing all the necessary modules and libraries
import os
import gc
import time
import math
import datetime
from math import log, floor
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
import random
from random import randrange
import psutil
import pickle
import joblib

import numpy as np
import pandas as pd
from pathlib import Path
from tqdm.notebook import tqdm as tqdm
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
#import pmdarima as pm

import seaborn as sns
from matplotlib import colors
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
import matplotlib.ticker as ticker

import plotly.express as px
import plotly.graph_objects as go
#import plotly.figure_factory as ff
from plotly.subplots import make_subplots
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot

import cufflinks as cf
cf.go_offline()
cf.set_config_file(offline=False, world_readable=True)

import pywt

import statsmodels
import statsmodels.api as sm
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.api import ExponentialSmoothing, Holt

import scipy
import lightgbm as lgbm
from xgboost import XGBRegressor

import warnings
warnings.filterwarnings("ignore")

from palettable.cartocolors.qualitative import Safe_3, Vivid_3

print('Libraries imported')
```

```
Libraries imported
```

2.2 Helper functions

Any helper functions that are used.

In [9]:
```
# Helper function to display decomposed plot results in subplots

def plotdecomp(results, axes ):
    results.observed.plot(ax=axes[0], legend=False)
    axes[0].set_ylabel('Observed')
    results.trend.plot(ax=axes[1], legend=False)
    axes[1].set_ylabel('Trend')
    results.seasonal.plot(ax=axes[2], legend=False)
    axes[2].set_ylabel('Seasonal')
    results.resid.plot(ax=axes[3], legend=False)
    axes[3].set_ylabel('Residual')
```

In [10]:
```
# Helper function to plot the share of zero values in each of the six years alongsid
# Inactive products: items with only zero values in the examined year

def hist_percentage_zero_values(df_sales):

    sales_train_id_stacked = df_sales.set_index(['id']).drop(['item_id','store_id',
                                                              'dept_id', 'cat_id', '
    sales_train_id_stacked = sales_train_id_stacked.merge(df_calendar[["date","year"
                                                          right_on="d",left_index=Tr

    percent_of_zeros = np.abs(sales_train_id_stacked.clip(0,1).groupby(level="year")

    names = [2011,2012,2013,2014,2015,2016]
    fig, ax = plt.subplots(6, 2,sharex="col",gridspec_kw={'width_ratios': [2, 0.5]},
    for i,t in enumerate(names):
        zero_tmp = percent_of_zeros.loc[t,:]
        ax_dist = ax[i,0]
        ax[i, 0].set_yscale('log')
        ax[i, 0].hist(zero_tmp,bins=60,color="lightsteelblue")
        ax[i, 0].set_ylabel(str(t))
        vals = ax[i, 0].get_xticks()
        ax[i, 0].set_xticklabels(['{:,.0%}'.format(x) for x in vals])


        zero_share = pd.Series(index=["Inactive products","Active products"])
        zero_share["Inactive products"] = zero_tmp.loc[zero_tmp==1].count()/zero_tmp
        zero_share["Active products"] = 1 - zero_share["Inactive products"]
        ax[i, 1].pie(zero_share, autopct='%1.0f%%', pctdistance=1.5,textprops={'font
                     colors=["lightsteelblue","cornflowerblue"])

    plt.title("Histogram of percentage of zero values",y=7.3,x=-4)
    plt.legend(labels=zero_share.index,bbox_to_anchor=(0.9,0.04), loc="lower right",
                   bbox_transform=fig.transFigure, ncol=1, fontsize=11)

    plt.tight_layout()
    return plt.show()
```

In [9]:
```
# Downcast function to reduce the memory usage

def downcast(df):
    cols = df.dtypes.index.tolist()
    types = df.dtypes.values.tolist()
    for i,t in enumerate(types):
        if 'int' in str(t):
            if df[cols[i]].min() > np.iinfo(np.int8).min and df[cols[i]].max() < np.
                df[cols[i]] = df[cols[i]].astype(np.int8)
```

```
            elif df[cols[i]].min() > np.iinfo(np.int16).min and df[cols[i]].max() <
                df[cols[i]] = df[cols[i]].astype(np.int16)
            elif df[cols[i]].min() > np.iinfo(np.int32).min and df[cols[i]].max() <
                df[cols[i]] = df[cols[i]].astype(np.int32)
            else:
                df[cols[i]] = df[cols[i]].astype(np.int64)
        elif 'float' in str(t):
            if df[cols[i]].min() > np.finfo(np.float16).min and df[cols[i]].max() <
                df[cols[i]] = df[cols[i]].astype(np.float16)
            elif df[cols[i]].min() > np.finfo(np.float32).min and df[cols[i]].max()
                df[cols[i]] = df[cols[i]].astype(np.float32)
            else:
                df[cols[i]] = df[cols[i]].astype(np.float64)
        elif t == np.object:
            if cols[i] == 'date':
                df[cols[i]] = pd.to_datetime(df[cols[i]], format='%Y-%m-%d')
            else:
                df[cols[i]] = df[cols[i]].astype('category')
    return df
```

2.3 Load data

Our dataset consists of five files:

1. `sales_train_validation.csv` - Contains the daily sold units for `[d_1 - d_1913]` (2011-01-29 until 2016-05-22)
2. `sales_train_evaluation.csv` - Contains the sales for `[d_1 - d_1941]` (2011-01-29 until 2016-06-19)
3. `calendar.csv` - Contains information like day-of-the week, month, year, SNAP food stamps allowed at this date or not
4. `sell_prices.csv` - Contains information about the price of the products sold per store and date (weekly average)
5. `submission.csv` - Demonstrates the correct format for submission to the competition

Regarding the split of the datasets:

- Training set: `[d_1 - d_1913]`
- Validation set: `[d_1914 - d_1941]`
- Evaluation set: `[d_1942 - d_1969]` (forecast target)

In [7]:

```
# Load the data

# when files in local directory (in the same PC folder as the code)
INPUT_DIR = ''

# when using Google Colab notebooks (with Gdrive mounted)
# mount google drive
# from google.colab import drive
# drive.mount('/content/drive')
# change the directory from local to the one on google drive
# INPUT_DIR = '/content/drive/My Drive/m5-forecasting-accuracy/'

df_sales = pd.read_csv(f'{INPUT_DIR}sales_train_evaluation.csv')
df_calendar = pd.read_csv(f'{INPUT_DIR}calendar.csv')
df_prices = pd.read_csv(f'{INPUT_DIR}sell_prices.csv')
# sample_submission = pd.read_csv(f'{INPUT_DIR}sample_submission.csv')
print('Data loaded')
```

```
Data loaded
```

3 Data inspection

Go to contents

As a first step we will inspect the datasets by checking the top and/or bottom rows, some general statistics regarding the included values, the datatypes etc.

3.1 Sales datasets

In [13]:
```
df_sales.shape
```

Out[13]:
```
(30490, 1919)
```

In [14]:
```
df_sales_eval.shape
```

Out[14]:
```
(30490, 1947)
```

In [15]:
```
df_sales.head()
```

Out[15]:

	id	item_id	dept_id	cat_id	store_id	state_id	d_1	d_2
0	HOBBIES_1_001_CA_1_validation	HOBBIES_1_001	HOBBIES_1	HOBBIES	CA_1	CA	0	0
1	HOBBIES_1_002_CA_1_validation	HOBBIES_1_002	HOBBIES_1	HOBBIES	CA_1	CA	0	0
2	HOBBIES_1_003_CA_1_validation	HOBBIES_1_003	HOBBIES_1	HOBBIES	CA_1	CA	0	0
3	HOBBIES_1_004_CA_1_validation	HOBBIES_1_004	HOBBIES_1	HOBBIES	CA_1	CA	0	0
4	HOBBIES_1_005_CA_1_validation	HOBBIES_1_005	HOBBIES_1	HOBBIES	CA_1	CA	0	0

5 rows × 1919 columns

In [16]:
```
df_sales_eval.head()
```

Out[16]:

	id	item_id	dept_id	cat_id	store_id	state_id	d_1	d_2
0	HOBBIES_1_001_CA_1_evaluation	HOBBIES_1_001	HOBBIES_1	HOBBIES	CA_1	CA	0	0
1	HOBBIES_1_002_CA_1_evaluation	HOBBIES_1_002	HOBBIES_1	HOBBIES	CA_1	CA	0	0
2	HOBBIES_1_003_CA_1_evaluation	HOBBIES_1_003	HOBBIES_1	HOBBIES	CA_1	CA	0	0
3	HOBBIES_1_004_CA_1_evaluation	HOBBIES_1_004	HOBBIES_1	HOBBIES	CA_1	CA	0	0
4	HOBBIES_1_005_CA_1_evaluation	HOBBIES_1_005	HOBBIES_1	HOBBIES	CA_1	CA	0	0

5 rows × 1947 columns

In [17]:
```

```

In [ ]:
```
df_sales.tail()
```

Out[17]:

	id	item_id	dept_id	cat_id	store_id	state_id	d_1	d_2	c
30485	FOODS_3_823_WI_3_validation	FOODS_3_823	FOODS_3	FOODS	WI_3	WI	0	0	
30486	FOODS_3_824_WI_3_validation	FOODS_3_824	FOODS_3	FOODS	WI_3	WI	0	0	
30487	FOODS_3_825_WI_3_validation	FOODS_3_825	FOODS_3	FOODS	WI_3	WI	0	6	
30488	FOODS_3_826_WI_3_validation	FOODS_3_826	FOODS_3	FOODS	WI_3	WI	0	0	
30489	FOODS_3_827_WI_3_validation	FOODS_3_827	FOODS_3	FOODS	WI_3	WI	0	0	

5 rows × 1919 columns

In [18]:
```
df_sales.info
```

Out[18]:
```
<bound method DataFrame.info of                                     id        item_id
dept_id   cat_id  \
0      HOBBIES_1_001_CA_1_validation  HOBBIES_1_001  HOBBIES_1  HOBBIES
1      HOBBIES_1_002_CA_1_validation  HOBBIES_1_002  HOBBIES_1  HOBBIES
2      HOBBIES_1_003_CA_1_validation  HOBBIES_1_003  HOBBIES_1  HOBBIES
3      HOBBIES_1_004_CA_1_validation  HOBBIES_1_004  HOBBIES_1  HOBBIES
4      HOBBIES_1_005_CA_1_validation  HOBBIES_1_005  HOBBIES_1  HOBBIES
...                              ...            ...        ...      ...
30485    FOODS_3_823_WI_3_validation    FOODS_3_823    FOODS_3    FOODS
30486    FOODS_3_824_WI_3_validation    FOODS_3_824    FOODS_3    FOODS
30487    FOODS_3_825_WI_3_validation    FOODS_3_825    FOODS_3    FOODS
30488    FOODS_3_826_WI_3_validation    FOODS_3_826    FOODS_3    FOODS
30489    FOODS_3_827_WI_3_validation    FOODS_3_827    FOODS_3    FOODS

      store_id state_id  d_1  d_2  d_3  d_4  ...  d_1904  d_1905  d_1906  \
0         CA_1       CA    0    0    0    0  ...       1       3       0
1         CA_1       CA    0    0    0    0  ...       0       0       0
2         CA_1       CA    0    0    0    0  ...       2       1       2
3         CA_1       CA    0    0    0    0  ...       1       0       5
4         CA_1       CA    0    0    0    0  ...       2       1       1
...        ...      ...  ...  ...  ...  ...  ...     ...     ...     ...
30485     WI_3       WI    0    0    2    2  ...       2       0       0
30486     WI_3       WI    0    0    0    0  ...       0       0       0
30487     WI_3       WI    0    6    0    2  ...       2       1       0
30488     WI_3       WI    0    0    0    0  ...       0       0       1
30489     WI_3       WI    0    0    0    0  ...       0       0       0

       d_1907  d_1908  d_1909  d_1910  d_1911  d_1912  d_1913
0           1       1       1       3       0       1       1
1           0       0       1       0       0       0       0
2           1       1       1       0       1       1       1
3           4       1       0       1       3       7       2
4           0       1       1       2       2       2       4
...       ...     ...     ...     ...     ...     ...     ...
30485       0       0       0       1       0       0       1
30486       0       0       0       0       0       1       0
30487       2       0       1       0       0       1       0
30488       0       0       1       0       3       1       3
30489       0       0       0       0       0       0       0

[30490 rows x 1919 columns]>
```

In [19]:
```
df_sales.describe()
```

Out[19]:

	d_1	d_2	d_3	d_4	d_5	d_6	
count	30490.000000	30490.000000	30490.000000	30490.000000	30490.000000	30490.000000	30490.0(
mean	1.070220	1.041292	0.780026	0.833454	0.627944	0.958052	0.9
std	5.126689	5.365468	3.667454	4.415141	3.379344	4.785947	5.0
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0(
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0(
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0(
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0(
max	360.000000	436.000000	207.000000	323.000000	296.000000	314.000000	316.0(

8 rows × 1913 columns

In [20]:
```
df_sales.dtypes
```

Out[20]:
```
id          object
item_id     object
dept_id     object
cat_id      object
store_id    object
             ...
d_1909       int64
d_1910       int64
d_1911       int64
d_1912       int64
d_1913       int64
Length: 1919, dtype: object
```

Observations & Findings:

- The sales dataframes contain daily sold units per product and store (i.e. not the revenue but the quantity):
  - 1 row for each item in each store (30490 rows for all combinations of 3049 items and 10 stores)
  - 1 column for each day from 2011-01-29 until 2016-05-22 (+ dates until 2016-06-19 in the evaluation dataset)
  - additional columns for item ID, department, category, store, and state
- Each row has a combination ID defined as follows:
  - CategoryName_DepartmentNumber_ItemNumber_StateName_StoreNumber_validation
- The columns corresponding to each date start with the prefix *d_* followed by a serial number (1 to 1913 or 1941)
- Many items have several days with zero sales (i.e. zero values for the date columns)
  - We can e.g. see that 75% of the items under the first 15 days have no sales value

3.2 Sell prices dataset

In [21]:
```
df_prices.shape
```

Out[21]: (6841121, 4)

In [22]:
```
df_prices.head()
```

Out[22]:

	store_id	item_id	wm_yr_wk	sell_price
0	CA_1	HOBBIES_1_001	11325	9.58
1	CA_1	HOBBIES_1_001	11326	9.58
2	CA_1	HOBBIES_1_001	11327	8.26
3	CA_1	HOBBIES_1_001	11328	8.26
4	CA_1	HOBBIES_1_001	11329	8.26

In [23]:
```
df_prices.dtypes
```

Out[23]:
```
store_id       object
item_id        object
wm_yr_wk        int64
sell_price    float64
dtype: object
```

In [24]:
```
df_prices.describe().apply(lambda s: s.apply('{0:.2f}'.format))
#the format is applied to suppress the scientific notation in the output from .descr
```

Out[24]:

	wm_yr_wk	sell_price
count	6841121.00	6841121.00
mean	11382.94	4.41
std	148.61	3.41
min	11101.00	0.01
25%	11247.00	2.18
50%	11411.00	3.47
75%	11517.00	5.84
max	11621.00	107.32

Observations & Findings:

- The sell prices dataframe contains information about the price of each product sold per store and date
- There are 6,841,121 distinct combinations of items in stores
- The sales price of each item is provided as a weekly average for the specific product in the specific store
  - Prices range from 0.01 to 107.32 dollars
  - Most of the prices (75%) are around 6 dollars
- We can interlink the df_prices and df_sales dataframes using the *store_id* and *item_id* fields

3.3 Calendar dataset

In [25]:
```
df_calendar.shape
```

Out[25]:
```
(1969, 14)
```

In [26]:
```
df_calendar.head()
```

Out[26]:

	date	wm_yr_wk	weekday	wday	month	year	d	event_name_1	event_type_1	event_nam
0	2011-01-29	11101	Saturday	1	1	2011	d_1	NaN	NaN	N
1	2011-01-30	11101	Sunday	2	1	2011	d_2	NaN	NaN	N
2	2011-01-31	11101	Monday	3	1	2011	d_3	NaN	NaN	N
3	2011-02-01	11101	Tuesday	4	2	2011	d_4	NaN	NaN	N
4	2011-02-02	11101	Wednesday	5	2	2011	d_5	NaN	NaN	N

In [27]:
```
df_calendar.tail()
```

Out[27]:

	date	wm_yr_wk	weekday	wday	month	year	d	event_name_1	event_type_1	eve
1964	2016-06-15	11620	Wednesday	5	6	2016	d_1965	NaN	NaN	
1965	2016-06-16	11620	Thursday	6	6	2016	d_1966	NaN	NaN	
1966	2016-06-17	11620	Friday	7	6	2016	d_1967	NaN	NaN	
1967	2016-06-18	11621	Saturday	1	6	2016	d_1968	NaN	NaN	
1968	2016-06-19	11621	Sunday	2	6	2016	d_1969	NBAFinalsEnd	Sporting	I

In [28]:
```
df_calendar.dtypes
```

Out[28]:
```
date            object
wm_yr_wk         int64
weekday         object
wday             int64
month            int64
year             int64
d               object
event_name_1    object
event_type_1    object
event_name_2    object
event_type_2    object
snap_CA          int64
snap_TX          int64
snap_WI          int64
dtype: object
```

In [29]:
```
df_calendar.describe(include = 'all')
```

Out[29]:

	date	wm_yr_wk	weekday	wday	month	year	d	event_name_1
count	1969	1969.000000	1969	1969.000000	1969.000000	1969.000000	1969	162
unique	1969	NaN	7	NaN	NaN	NaN	1969	30
top	2011-01-29	NaN	Saturday	NaN	NaN	NaN	d_1	SuperBowl
freq	1	NaN	282	NaN	NaN	NaN	1	6
mean	NaN	11347.086338	NaN	3.997461	6.325546	2013.288471	NaN	NaN
std	NaN	155.277043	NaN	2.001141	3.416864	1.580198	NaN	NaN
min	NaN	11101.000000	NaN	1.000000	1.000000	2011.000000	NaN	NaN
25%	NaN	11219.000000	NaN	2.000000	3.000000	2012.000000	NaN	NaN
50%	NaN	11337.000000	NaN	4.000000	6.000000	2013.000000	NaN	NaN
75%	NaN	11502.000000	NaN	6.000000	9.000000	2015.000000	NaN	NaN
max	NaN	11621.000000	NaN	7.000000	12.000000	2016.000000	NaN	NaN

Observations & Findings:

- The calendar dataframe contains information on the dates in scope (in the *yyyy/dd/mm* format)
  - It includes related features like day-of-the week, month, year, and binary flags (1 or 0) for whether the stores in each state allowed purchases with SNAP food stamps at this date (1) or not (0)
- *wm_yr_wk* can interconnect df_calendar to df_prices
- We have information on 1969 days (from 2011-01-29 until 2016-06-19)
  - This covers the training (days 1-1913), validation (days 1-1941) and evaluation (days 1942-1969) time periods
- New weeks start on Saturday and there are 282 Saturdays in the data (i.e. 282 weeks overall)
- There are overall 162 days that have at least one event
  - There are 4 distinct types of events
  - Religious is the most common event type occuring 55 times
  - Only 5 days have a second event (e.g. Easter and OrthodoxEaster on same day)
  - There are overall 30 unique event names

3.4 Missing values & zero values

In [30]:
```
# Count and print the missing values in each dataframe
print(df_sales.isna().sum().sum())
print(df_prices.isna().sum().sum())
print(df_calendar.isna().sum().sum())
```

```
0
0
7542
```

In [31]:
```
# Check which columns include the missing values in the df_calendar dataframe
print(df_calendar.isna().sum())
```

```
date              0
wm_yr_wk          0
weekday           0
wday              0
month             0
year              0
d                 0
event_name_1    1807
event_type_1    1807
event_name_2    1964
event_type_2    1964
snap_CA           0
snap_TX           0
snap_WI           0
dtype: int64
```

In [32]:
```
# Count zero values for each dataframe
zeros_sales = ((df_sales == 0).sum(axis=0)/len(df_sales.index)).to_frame('%zeros')
zeros_prices = ((df_prices == 0).sum(axis=0)/len(df_prices)).to_frame('%zeros')
zeros_calendar = ((df_calendar == 0).sum(axis=0)/len(df_calendar)).to_frame('%zeros'
```

In [33]:
```
print(zeros_sales)
```

```
            %zeros
id        0.000000
item_id   0.000000
dept_id   0.000000
cat_id    0.000000
store_id  0.000000
...            ...
d_1909    0.590849
d_1910    0.597704
d_1911    0.561069
d_1912    0.515513
d_1913    0.511742

[1919 rows x 1 columns]
```

In [34]:
```
print(zeros_prices)
```

```
            %zeros
store_id       0.0
item_id        0.0
wm_yr_wk       0.0
sell_price     0.0
```

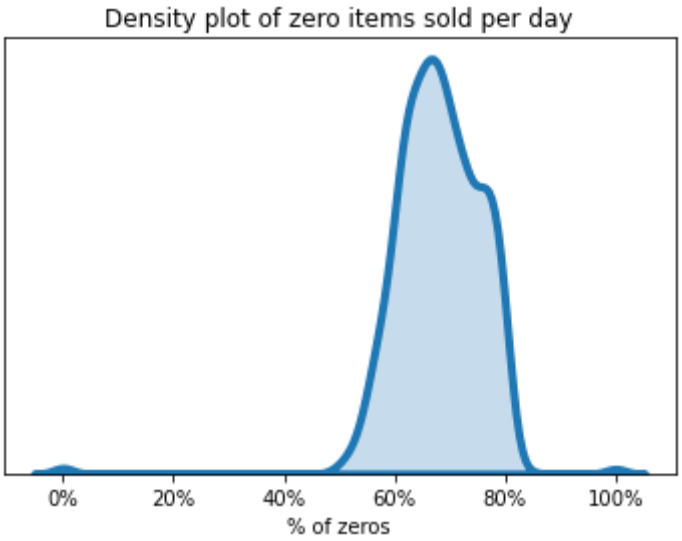
In [35]:
```
print(zeros_calendar)
```

```
               %zeros
date         0.000000
wm_yr_wk     0.000000
weekday      0.000000
wday         0.000000
month        0.000000
year         0.000000
d            0.000000
```

```
event_name_1  0.000000
event_type_1  0.000000
event_name_2  0.000000
event_type_2  0.000000
snap_CA       0.669883
snap_TX       0.669883
snap_WI       0.669883
```

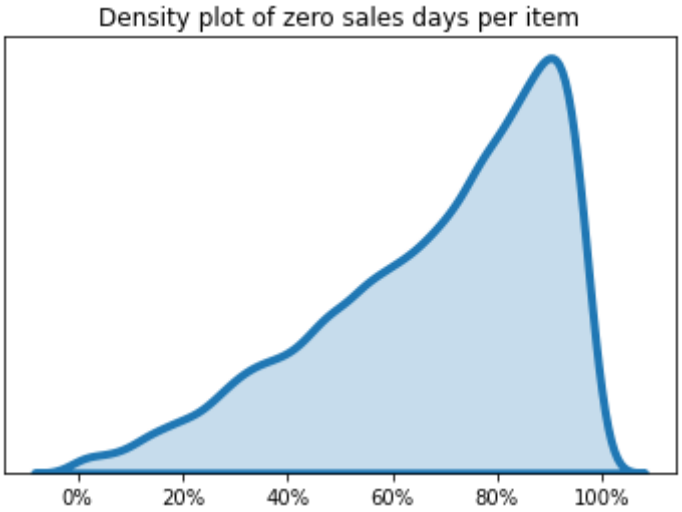
In [36]:

```
# Density distributions to get more insights on the zero values for df_sales datafra

# Plot the percentage of zero sales days per item
zeros_sales[zeros_sales.index.str.contains('d_')]  # keep only the rows for days

ax = sns.distplot(zeros_sales['%zeros'], hist=False, kde=True,
             kde_kws={'linewidth': 4, 'shade': True})

plt.title('Density plot of zero items sold per day')
plt.xlabel('% of zeros')
vals = ax.get_xticks()
ax.set_xticklabels(['{:,.0%}'.format(x) for x in vals])
plt.ylabel('')
plt.yticks([])
plt.show()
```



In [37]:

```
# Plot the percentage of zero sales days per item
zeros_days = ((df_sales == 0).sum(axis=1)/1914).to_frame('%zeros')

ax = sns.distplot(zeros_days['%zeros'], hist=False, kde=True,
             kde_kws={'linewidth': 4, 'shade': True})

plt.title('Density plot of zero sales days per item')
plt.xlabel('')
vals = ax.get_xticks()
ax.set_xticklabels(['{:,.0%}'.format(x) for x in vals])
plt.ylabel('')
plt.yticks([])
plt.show()
```



Observations & Findings:

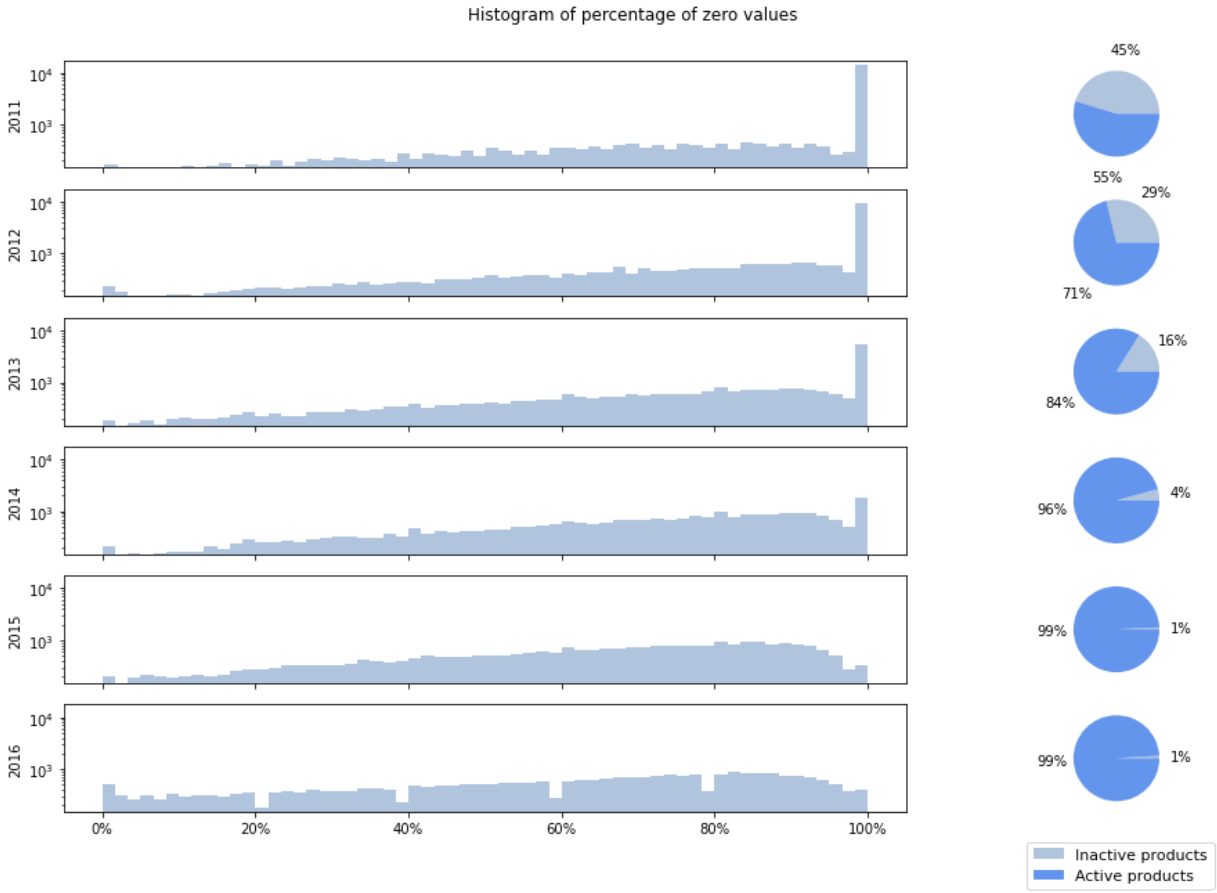
Missing values analysis (for columns with nominal data)

- Neither df_sales nor df_prices data have any missing values overall
- The df_calendar dataframe has 7542 missing values
- Out of the 1969 distinct days that are included in the df_calendar dataframe:
  - only 162 days (8.2% of all) have an event (1807 days without an event)
  - only 5 days have a second event (that's 0.2% of all)

Zero values analysis (for columns with numerical data)

- The df_prices dataframe has no zero values overall (i.e. all items had a specified price for every day)
- The df_sales and df_calendar dataframes have several columns with zero values
- For df_calendar we understand that each state has 650 SNAP days
  - Explanation: there are 67% zero values in each snap-state related column, i.e. 1319 days are not SNAP days
- For df_sales we have some observations from the density plots
  - Density plot of zero items sold per day: We see that on most days 60-80% of the items weren't sold
  - Density plot of zero sales day per item: We see that few items are sold for more than half of the days - the position of the peak actually shows that most items don't have a sale for about 90% of the days

In [38]:
```
# Plot the share of zero values in each of the six years alongside the percentage of

hist_percentage_zero_values(df_sales)
```



Observations & Findings:

The percentage of inactive products is higher in the first examined years. This is explained by the fact that some of the products found in the later years would not have had any sales in the first years since they were probably newer products introduced later to the stores.

Interactive visualization showing the distribution of the items across different aggregation levels (source).

In [39]:
```
group = df_sales.groupby(['state_id','store_id','cat_id','dept_id'],as_index=False)[
group['USA'] = 'United States of America'
group.rename(columns={'state_id':'State','store_id':'Store','cat_id':'Category',
                      'dept_id':'Department','item_id':'Count'},
             inplace=True)
fig = px.treemap(group,
                 path=['USA', 'State', 'Store', 'Category', 'Department'],
                 values='Count',
                 color='Count',
                 color_continuous_scale=px.colors.sequential.RdBu,
                 #px.colors.qualitative.Set1
                 title='Distribution of items (interactive plot)')
fig.update_layout(template='seaborn')
fig.show()
```

United States of America

CA

4 Time series analysis: Sales

Go to contents

4.1 Decomposition components

Any given timeseries is characterized by some systematic and non-systematic components:

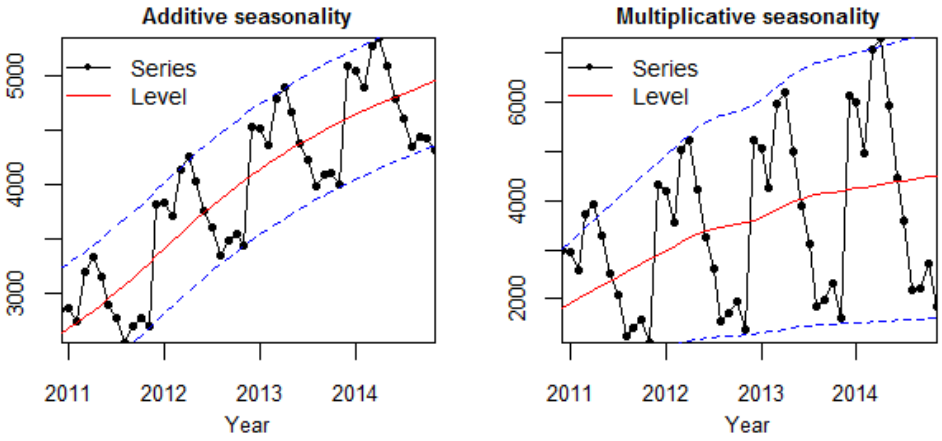
- Systematic: Components of the time series that have consistency or recurrence and can be described and modeled
  - Level: The average value in the series
  - Trend: The increasing or decreasing value in the series
  - Seasonality: The repeating short-term cycle in the series
- Non-Systematic: Components of the time series that cannot be directly modeled
  - Noise: The random variation in the series

A timeseries is considered to be an aggregate or combination of these four components:

- All series have a level and noise
- The trend and seasonality components are optional
- The components can combine either additively or multiplicatively
  - An additive model is linear where changes over time are consistently made by the same amount
  - A multiplicative model is nonlinear, such as quadratic or exponential, and changes increase or decrease over time

Whether our model is *additive* or *multiplicative* depends on if the amplitude of the data's seasonality is level (mean) dependent. If the seasonality's amplitude is independent of the level

then we should use the additive model, and if the seasonality's amplitude is dependent on the level then we should use the multiplicative model.

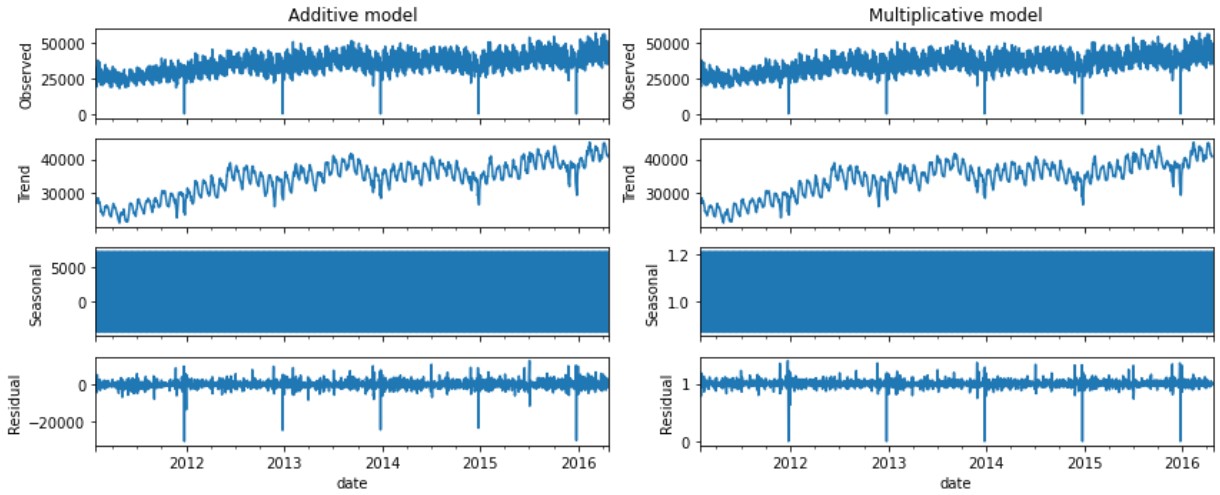


In [4]:
```
# First create some usefull lists
item_ids = sorted(list(set(df_sales['id'])))
state_ids = ['CA', 'TX', 'WI']
store_ids = df_sales['store_id'].unique()
categ_ids = ['FOODS','HOBBIES','HOUSEHOLD']
dept_ids = df_sales['dept_id'].unique()
date_col = [c for c in df_sales.columns if c.startswith('d_')] # all the columns tha
state_names = ['California', 'Texas', 'Wisconsin']
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov
```

In [41]:
```
# Calculate the daily sales (sum of all items sold per day)
daily_sales = pd.DataFrame(df_sales[date_col].sum(axis =0),columns=["sales"])

# Convert the index to a datetime column
base = datetime.datetime(2011,1,29) # set the start date
daily_sales['date'] = [base + datetime.timedelta(days=x) for x in range(len(date_col
daily_sales.set_index('date', drop=True, inplace=True)
daily_sales.sort_index(inplace=True)
```

In [42]:
```
# Decomposition of the timeseries using an additive and a multiplicative model
decomp_results_add = seasonal_decompose(daily_sales, model='additive')
decomp_results_mul = seasonal_decompose(daily_sales, model='multiplicative')
```

In [43]:
```
# Plot decomposition results for additive vs multiplicative models
fig, axes = plt.subplots(ncols=2, nrows=4, sharex=True, figsize=(12,5))

plotdecomp(decomp_results_add, axes[:,0])
plotdecomp(decomp_results_mul, axes[:,1])

axes[0,0].set_title('Additive model')
axes[0,1].set_title('Multiplicative model')

plt.tight_layout()
plt.show()
```

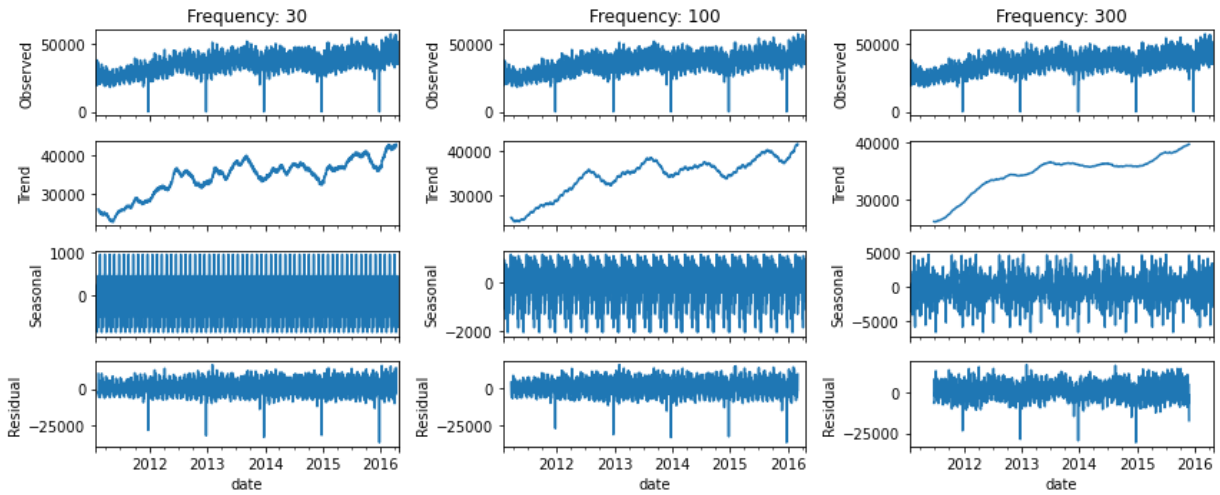


In [44]:

```
# Decomposition of the timeseries using different frequency levels
decomp_results_30freq = seasonal_decompose(daily_sales, model='additive',freq=30)
decomp_results_100freq = seasonal_decompose(daily_sales, model='additive',freq=100)
decomp_results_300freq = seasonal_decompose(daily_sales, model='additive',freq=300)
```

In [45]:

```
# Plot different frequency decomposition results in one figure for comparative analy
fig, axes = plt.subplots(ncols=3, nrows=4, sharex=True, figsize=(12,5))

plotdecomp(decomp_results_30freq, axes[:,0])
plotdecomp(decomp_results_100freq, axes[:,1])
plotdecomp(decomp_results_300freq, axes[:,2])

axes[0,0].set_title('Frequency: 30')
axes[0,1].set_title('Frequency: 100')
axes[0,2].set_title('Frequency: 300')


plt.tight_layout()
plt.show()
```



Observations & Findings:

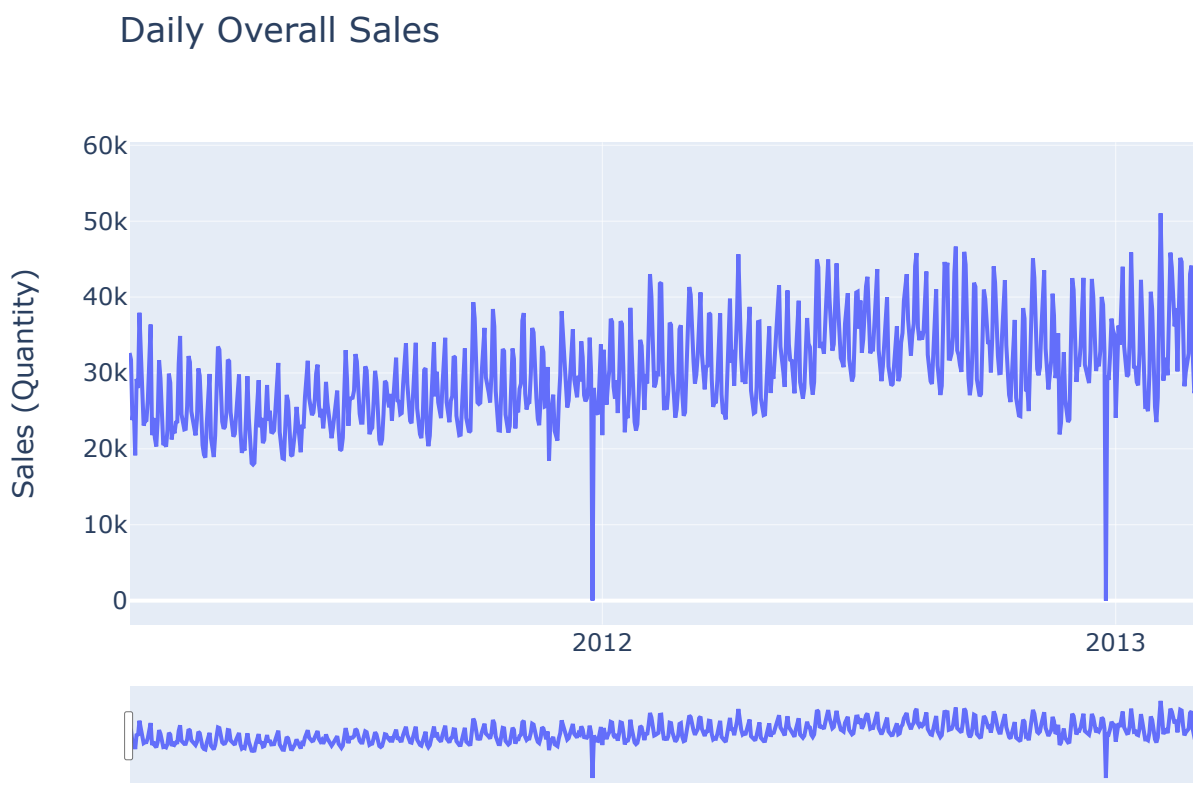
- The first graph shows the actual time series
  - Sales seem to have an increasing trend over the years with the average clearly moving upwards in time
  - There seems to be some periodical drops in sales which we need to further investigate

- The second graph shows the trend, confirming how it is increasing over the years
- The third graph shows there is strong seasonality (clear for all examined frequency levels)
- The last graph shows that there is a lot of residual noise (randomness) in our timeseries

4.2 Aggregated sales

Using plots on different aggregation levels to analyze our sales data.

In [46]:
```
# Plot the daily sales
fig = go.Figure()

fig.add_trace(go.Scatter(x=daily_sales.index, y=daily_sales['sales'], showlegend=Fal
                         mode='lines',
                         marker=dict(color=np.random.rand(3,3))))

fig.update_layout(height = 500, width = 1500,
                  title_text = "Daily Overall Sales",
                  xaxis_rangeslider_visible = True,
                  xaxis_title = "Days",
                  yaxis_title = "Sales (Quantity)")

fig.show()
```



Observations & Findings:

- By zooming in on the periodical drops in sales we see that these were all recorded on Christmas day - the only day when all stores were closed

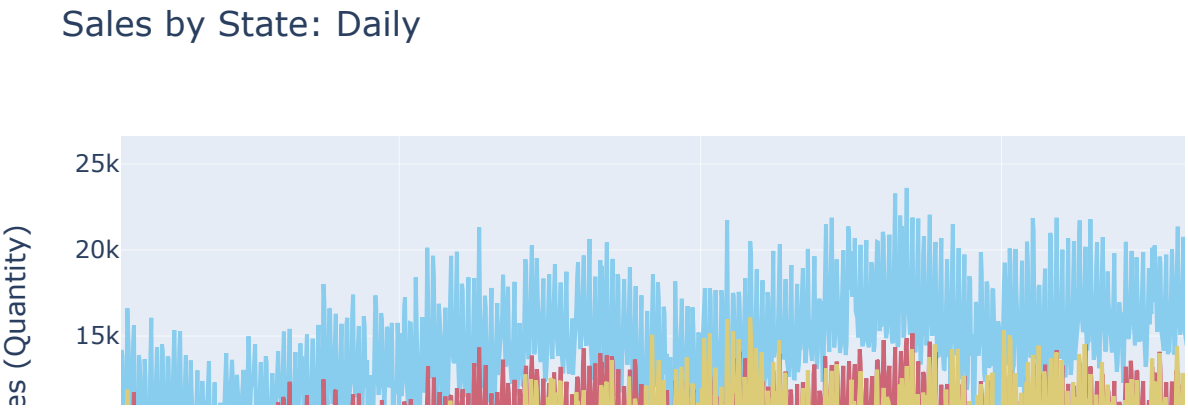
4.2.1 Sales per State

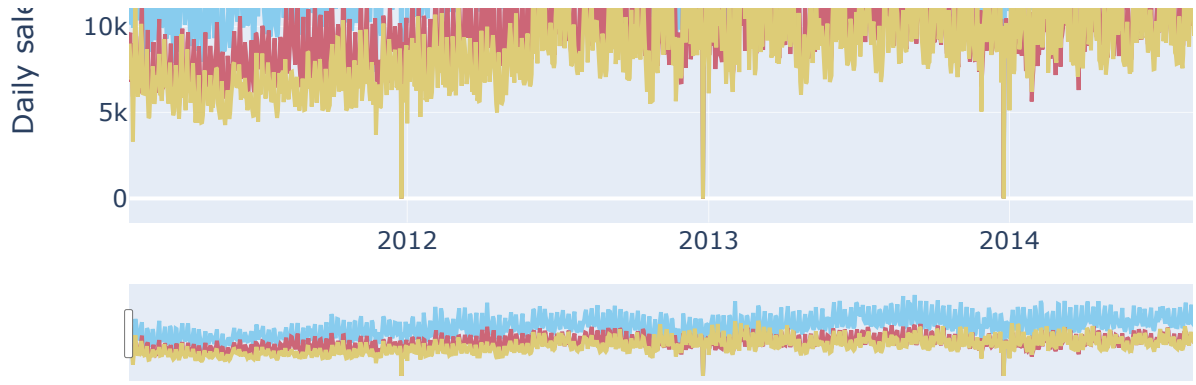
Analyze sales per state on several aggregate levels.

In [47]:
```
# Separate the (daily) sales by state
# separate the daily sales by state from the df_sales dataframe
df_states = df_sales.loc[:,"state_id":df_sales.columns[-1]].groupby("state_id").sum(
# transpose and configure the index
df_states = df_states.transpose().reset_index().rename(columns={'index':'d'})
# merge with the related df_calendar rows
df_states = pd.merge(df_states, df_calendar.loc[0:len(df_states)], on='d').fillna(0)
# set the date column as datetime type
df_states['date'] = pd.to_datetime(df_states['date'])
```

In [48]:
```
# Several aggregation levels for the sales by state
# Weekly
df_states_weekly = df_states.groupby(pd.Grouper(key='date',freq='W-SAT')).sum().filt

# Monthly
df_states_monthly = df_states.groupby(pd.Grouper(key='date',freq='MS')).sum().filter

# Yearly
df_states_yearly = df_states.groupby(pd.Grouper(key='date',freq='YS')).sum().filter(

# Overall
df_states_sum = pd.DataFrame(df_states.filter(items=state_ids, axis=1).sum(axis=0),c
```

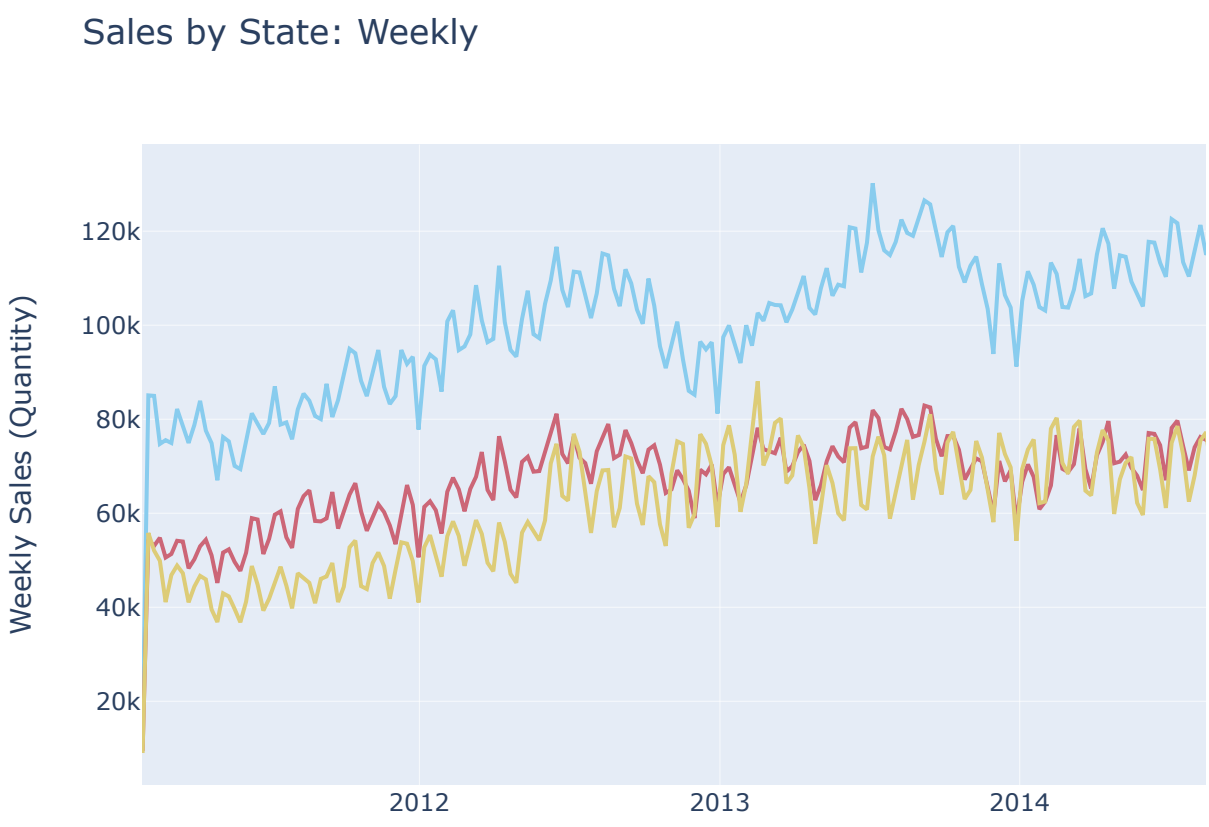
In [49]:
```
# Plot the daily sales by state
fig = go.Figure()

for i in range(len(state_names)):
  fig.add_trace(go.Scatter(x = df_states.date, y = df_states.iloc[:,i+1],
                           name =state_names[i],
                           line_color = px.colors.qualitative.Safe[i]))

fig.update_layout(height = 500, width = 1000,
                  title_text = "Sales by State: Daily",
                  xaxis_rangeslider_visible = True,
                  yaxis_title = "Daily sales (Quantity)")

fig.show()
```





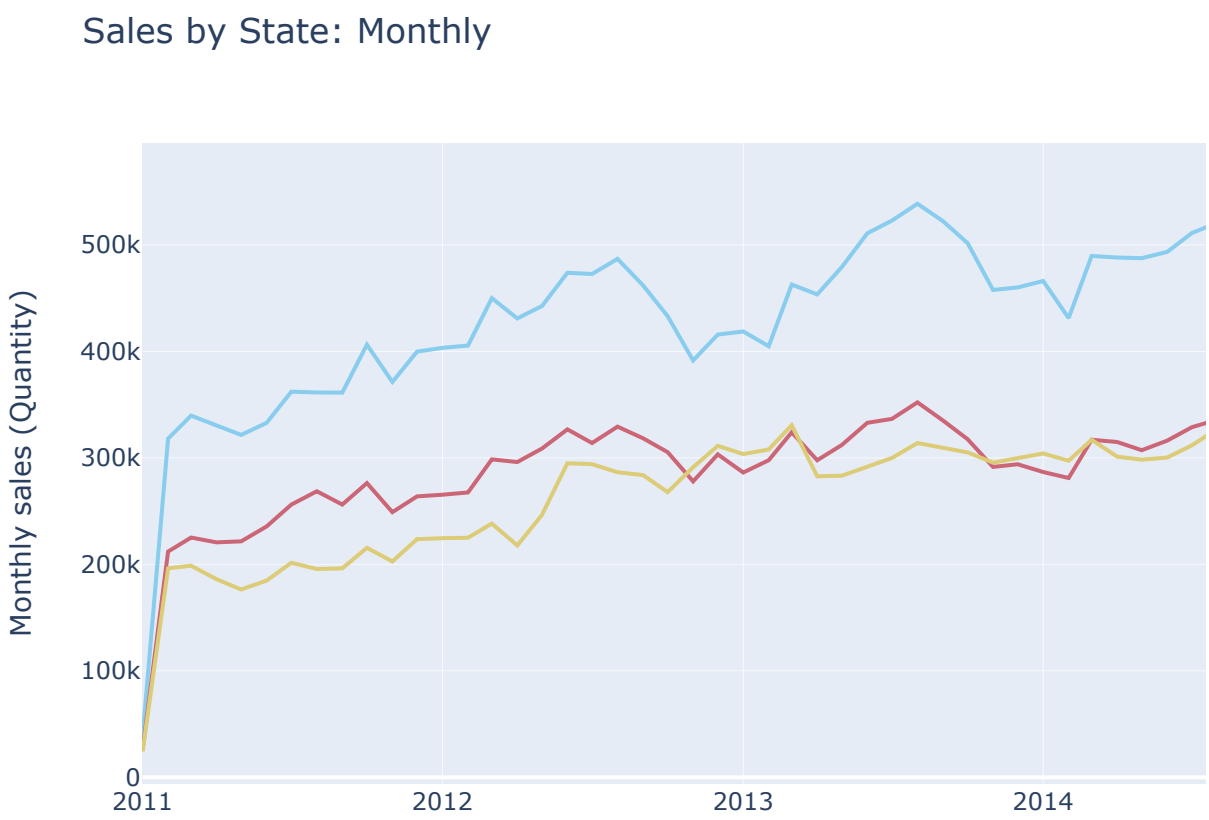
In [50]:

```
# Plot the weekly sales by state
fig = go.Figure()

for i in range(len(state_names)):
  fig.add_trace(go.Scatter(x = df_states_weekly.index, y = df_states_weekly.iloc[:,i
                           name =state_names[i],
                           line_color = px.colors.qualitative.Safe[i]))

fig.update_layout(height = 500, width = 1000,
                  title_text = "Sales by State: Weekly",
                  yaxis_title = "Weekly Sales (Quantity)")

fig.show()
```



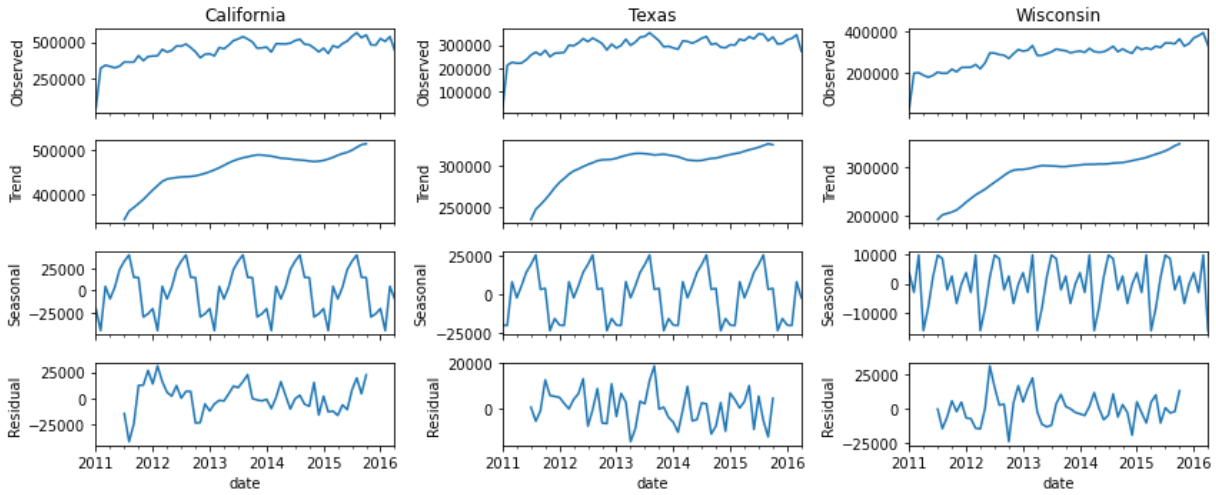
In [51]:

```
# Plot the monthly sales by state
fig = go.Figure()

for i in range(len(state_names)):
  fig.add_trace(go.Scatter(x = df_states_monthly.index, y = df_states_monthly.iloc[:
                           name =state_names[i],
                           line_color = px.colors.qualitative.Safe[i]))

fig.update_layout(height = 500, width = 1000,
                  title_text = "Sales by State: Monthly",
                  yaxis_title = "Monthly sales (Quantity)")

fig.show()
```



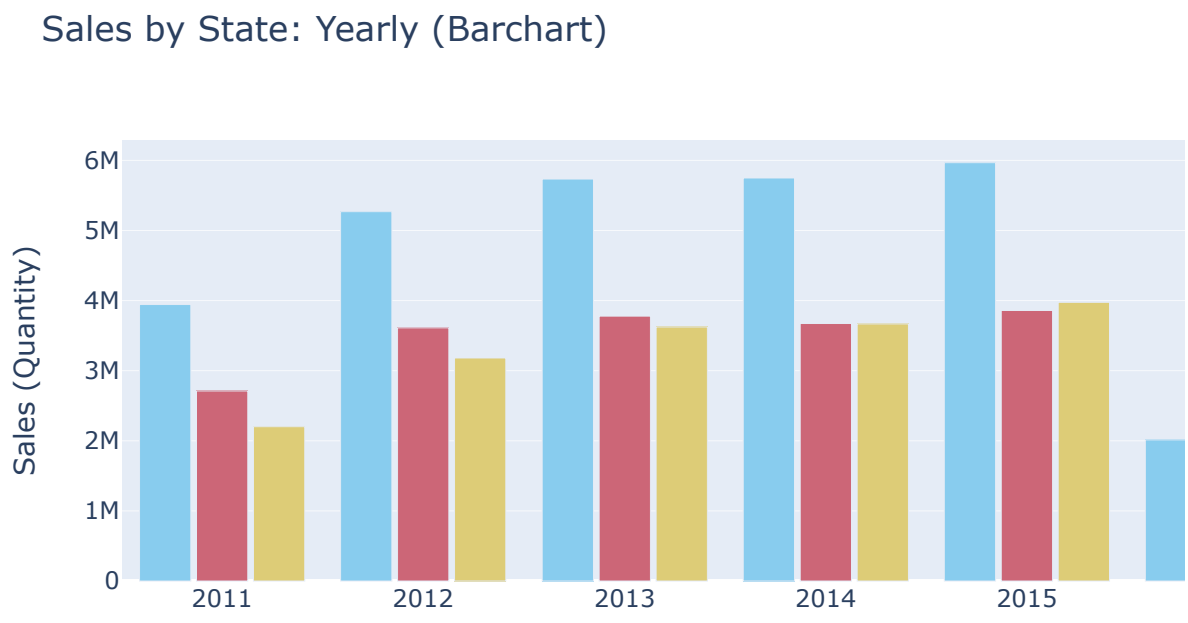
In [52]:

```
# Plot the decomposition components for each state timeseries - monthly aggregation
fig, axes = plt.subplots(ncols=3, nrows=4, sharex=True, figsize=(12,5))

for i in range(len(state_names)):
    plotdecomp(seasonal_decompose(df_states_monthly.iloc[:,i], model='additive'), ax
    axes[0,i].set_title(state_names[i])

plt.tight_layout()

plt.show()
```

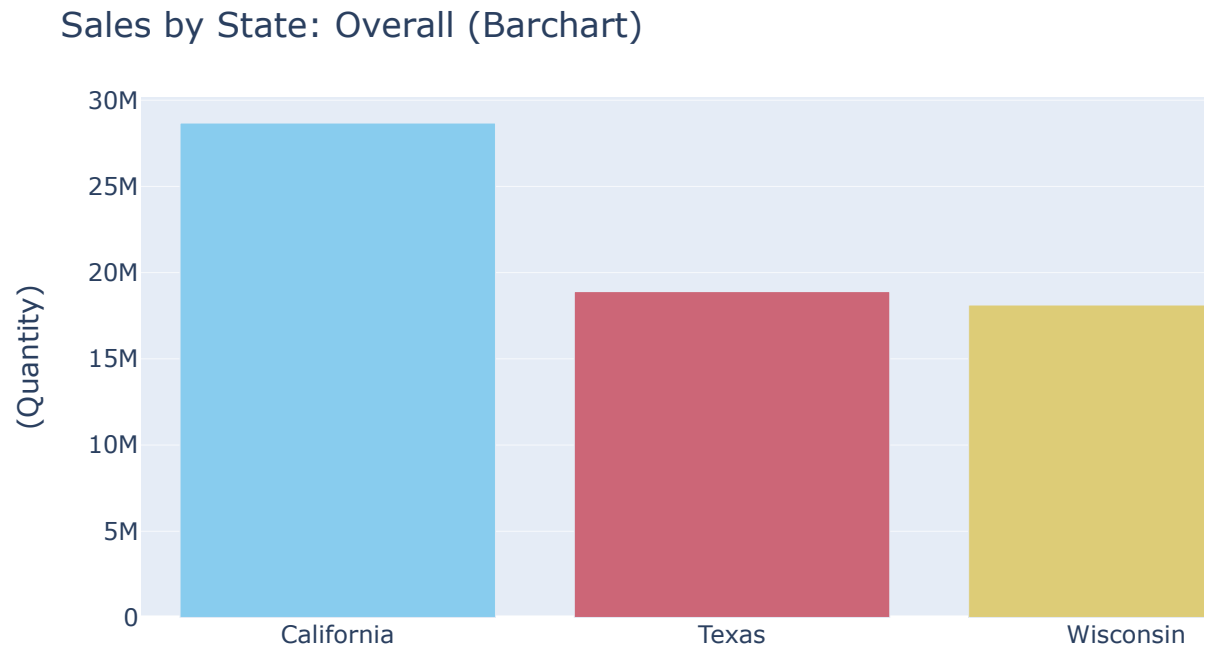


In [53]:

```
# Yearly sales by state - Barchart
fig = go.Figure()

for i in range(len(state_names)):
  fig.add_trace(go.Bar(x = df_states_yearly.index,
                       y = df_states_yearly[state_ids[i]],
                       name = state_names[i],
                       marker_color = px.colors.qualitative.Safe[i]))

fig.update_layout(height = 400, width = 800,
                  title_text = 'Sales by State: Yearly (Barchart)',
                  xaxis_title = "",
                  yaxis_title = "Sales (Quantity)",
                  legend=dict(x = 1.0, y = 1.0,
                              bgcolor='rgba(255, 255, 255, 0)',
                              bordercolor='rgba(255, 255, 255, 0)'),
                  barmode='group',
                  bargap=0.15, # gap between bars of adjacent location coordinates
                  bargroupgap=0.1 # gap between bars of the same location coordinate
                  )

fig.show()
```



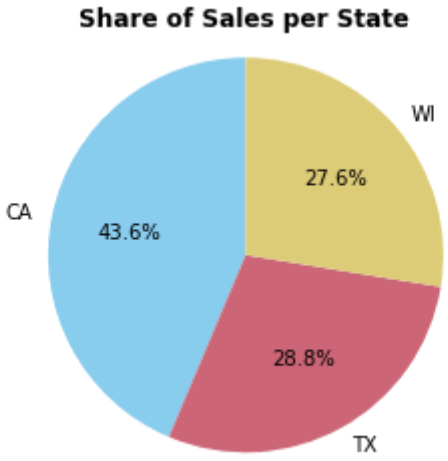
In [54]:
```
# Overall sales by state - Barchart

fig = px.bar(df_states_sum,
             x = state_names,
             y = df_states_sum.iloc[:,0],
             color = state_names,
             labels = {'color':'State','x':'State','y':'Sales'},
             color_discrete_sequence= px.colors.qualitative.Safe)

fig.update_layout(height = 400, width = 800,
                  title_text = 'Sales by State: Overall (Barchart)',
                  xaxis_title = "",
                  yaxis_title = "(Quantity)")

fig.show()
```



In [55]:
```
# Calculate the ratio of sales per state (to overall sales)
df_states_ratio = df_states_sum/df_states_sum.sum()*100
```

In [56]:
```
# Plot the sales per state ratio
fig, ax = plt.subplots()

ax.pie(df_states_ratio.iloc[:,0],
       labels = df_states_ratio.index,
       autopct = '%1.1f%%',
       shadow = False,
       colors = ['#88ccee','#cc6677','#ddcc77'],
       startangle = 90)

ax.axis('equal')
```

```
plt.title("Share of Sales per State", fontweight = "bold")

plt.show()
```



In [57]:

```
# Overall sales per state - Choropleth map
# Plot the overal sales by state in a choropleth map
fig = px.choropleth(locations = state_ids,
                    locationmode = "USA-states",
                    color = df_states_sum.iloc[:,0],
                    scope = "usa",
                    color_continuous_scale = 'teal',
                    labels = {'color':'Sales','locations':'State'})

fig.update_layout(title_text = 'Overall Sales by State (map view)')

fig.show()
```

Overall Sales by State (map view)

Observations & Findings:

From observing the charts:

- California accounts for most of the sales (28.7M items sold overall, 43.6% of the overall sales),
  - Sales for CA have always been the highest throughout the observed period
  - Seasonality impacts CA sales the most - the peaks in August are the most evident at CA in comparison to other states
- Texas is 2nd in overall sales (18.9M, 28.8% share), followed closely by Wisconsin (18.1M items sold overall, 27.6% share)
- Wisconsin have shown the highest increase in sales over the years
  - Sales in WI were initially lower than TX (until 2013), the state slowly approached and was similar to TX until Aug-2015 and eventually surpassed TX towards the end of the reporting period

From the state-specific decomposition components analysis:

- CA and WI show a gradually increasing trend while TX grew up fast in the beginning and then became somewhat stagnant
- CA and TX show a similar seasonality peaking around Jul-Aug and dipping at Dec-Jan
- WI shows a different seasonality peaking at Mar, dipping at Apr and then peaking again at Aug

4.2.2 Sales per Store

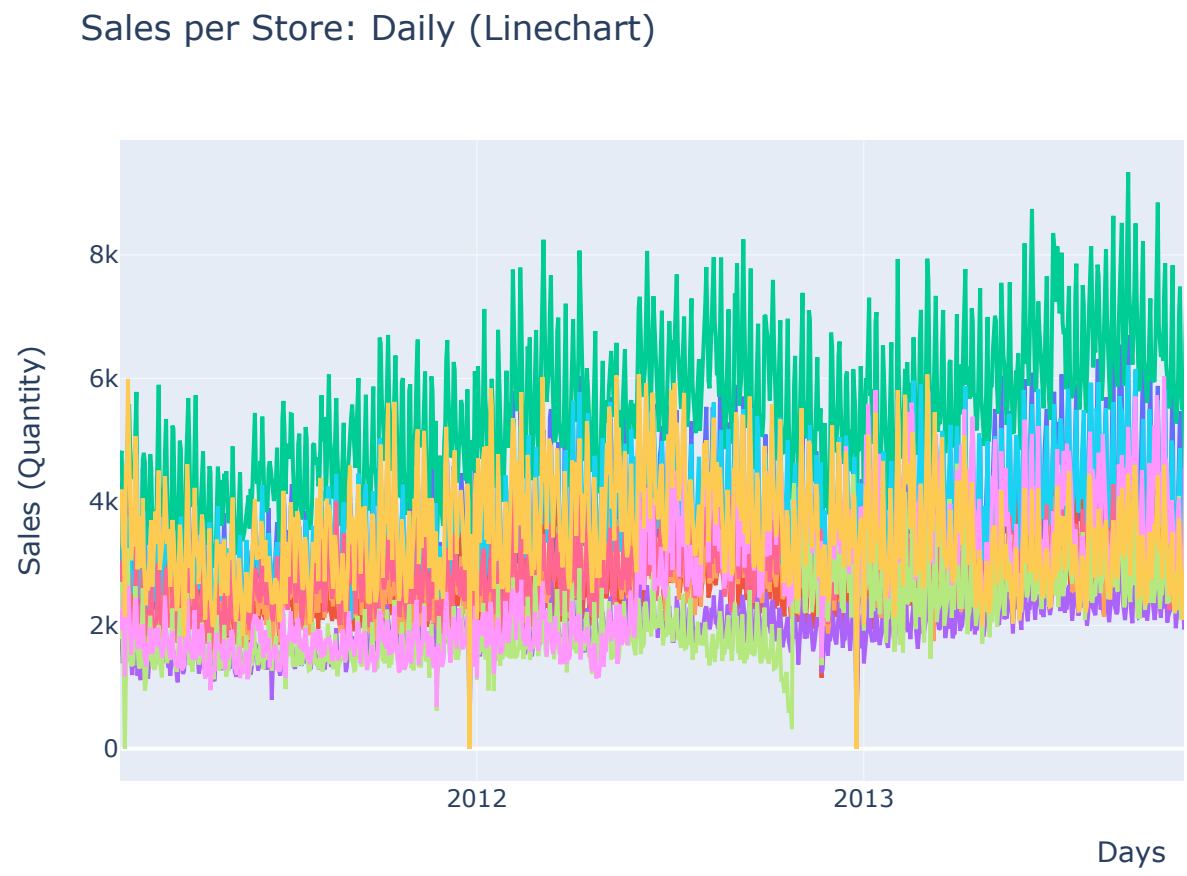
We have a total of 10 stores in the analysed dataframe: 4 are located in California, 3 in Texas and 3 in Wisconsin.

In [58]:
```
# Separate the daily sales by item
# separate the daily sales per item from the df_sales dataframe and transpose the da
df_items = df_sales.set_index('id')[date_col].T
# merge with df_calendar dataframe to get the dates
df_items = df_items.merge(df_calendar.set_index('d')['date'], left_index=True,right_
# set the date column as datetime type
df_items['date'] = pd.to_datetime(df_items['date'])
# set the date column as index for the created dataframe
df_items = df_items.set_index('date')
```

In [59]:
```
# Separate the daily sales by store
# separate the daily sales by store
df_stores = df_sales.groupby("store_id").sum()
# transpose and configure the index
df_stores = df_stores.transpose().reset_index().rename(columns={'index':'d'})
# merge with the related df_calendar rows
df_stores = pd.merge(df_stores, df_calendar.loc[0:len(df_stores)], on='d').fillna(0)
# set the date column as datetime type
df_stores['date'] = pd.to_datetime(df_stores['date'])
```

In [60]:
```
# Create some aggregates for the sales per store
# Sum of sales per store
df_stores_sum = pd.DataFrame(df_stores.filter(items=store_ids, axis=1).sum(axis=0),c

# Average sales per store
df_stores_avg = pd.DataFrame(df_stores.filter(items=store_ids, axis=1).mean(axis=0),
```

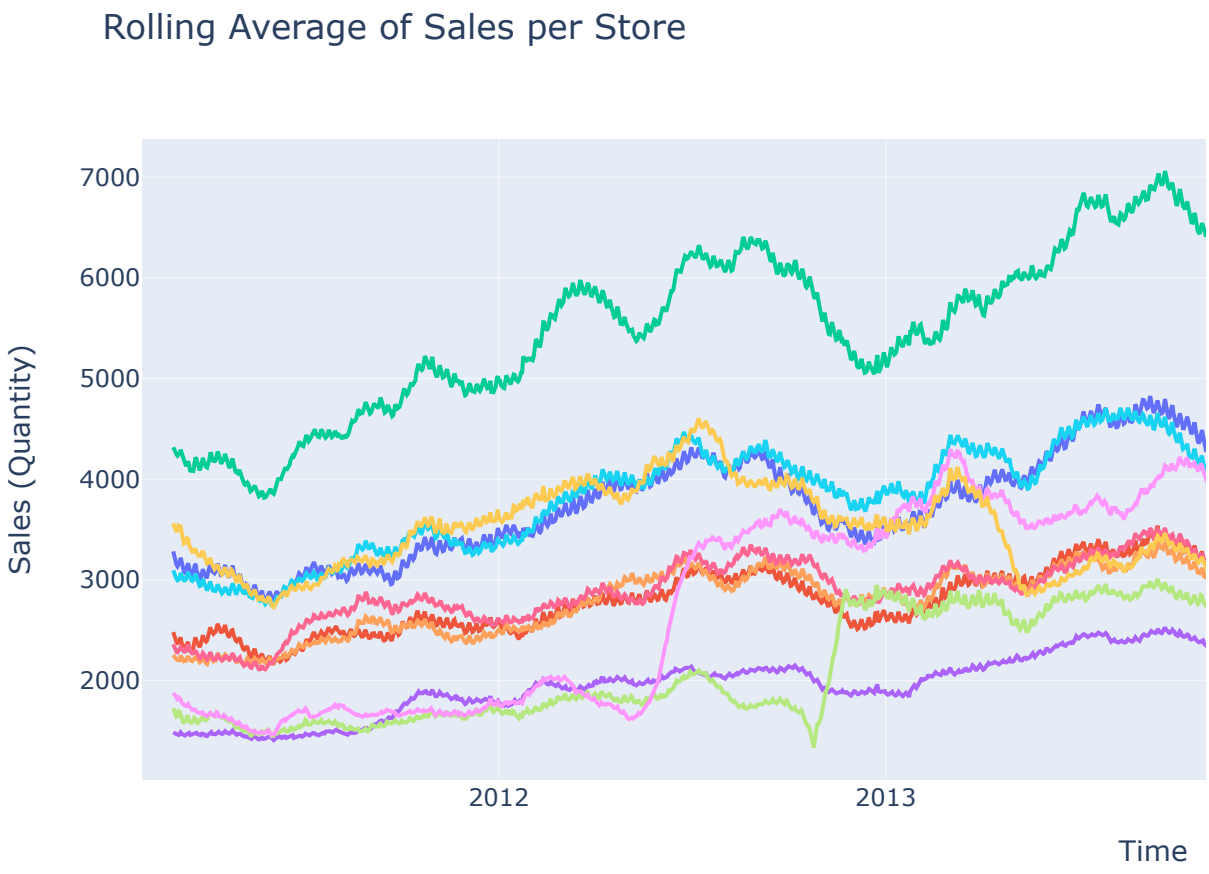
In [61]:
```
# Plot the daily sales for each store - Linechart
fig = go.Figure()

for s in store_ids:
    store_items = [c for c in df_items.columns if s in c]
    plot_data = df_items[store_items].sum(axis=1)
    fig.add_trace(go.Scatter(x = plot_data.index,
                             y = plot_data,
                             name = s))

fig.update_layout(height = 500, width = 1200,
                  title="Sales per Store: Daily (Linechart)",
                  xaxis_title = "Days",
                  yaxis_title = "Sales (Quantity)")

fig.show()
```



The above chart is obviously not very insightful, therefore, in order to have a more helpful view of all 10 stores we will use the 30-days rolling average to plot the sales for each store.

In [62]:
```
# Plot the rolling average for items sold in each store in time - Linechart
```

```
fig = go.Figure()

means = []

for s in store_ids:
    store_items = [c for c in df_items.columns if s in c]
    plot_data = df_items[store_items].sum(axis=1).rolling(30).mean()
    means.append(np.mean(df_items[store_items].sum(axis=1)))
    fig.add_trace(go.Scatter(x = plot_data.index,
                             y = plot_data,
                             name = s))

fig.update_layout(height = 500, width = 1200,
                  title="Rolling Average of Sales per Store",
                  xaxis_title = "Time",
                  yaxis_title = "Sales (Quantity)")

fig.show()
```

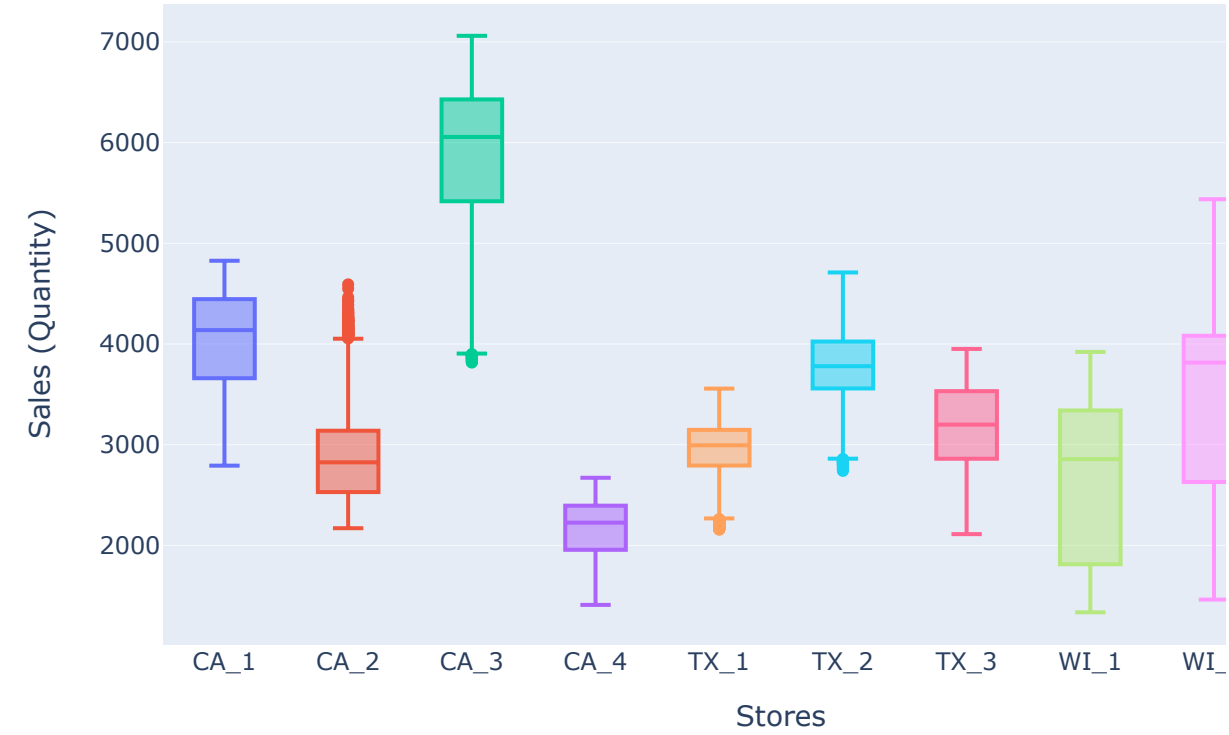


Similarly, when plotting a boxplot of the sales data per store we can use both the actual and the rolling average for our observations.

In [63]:

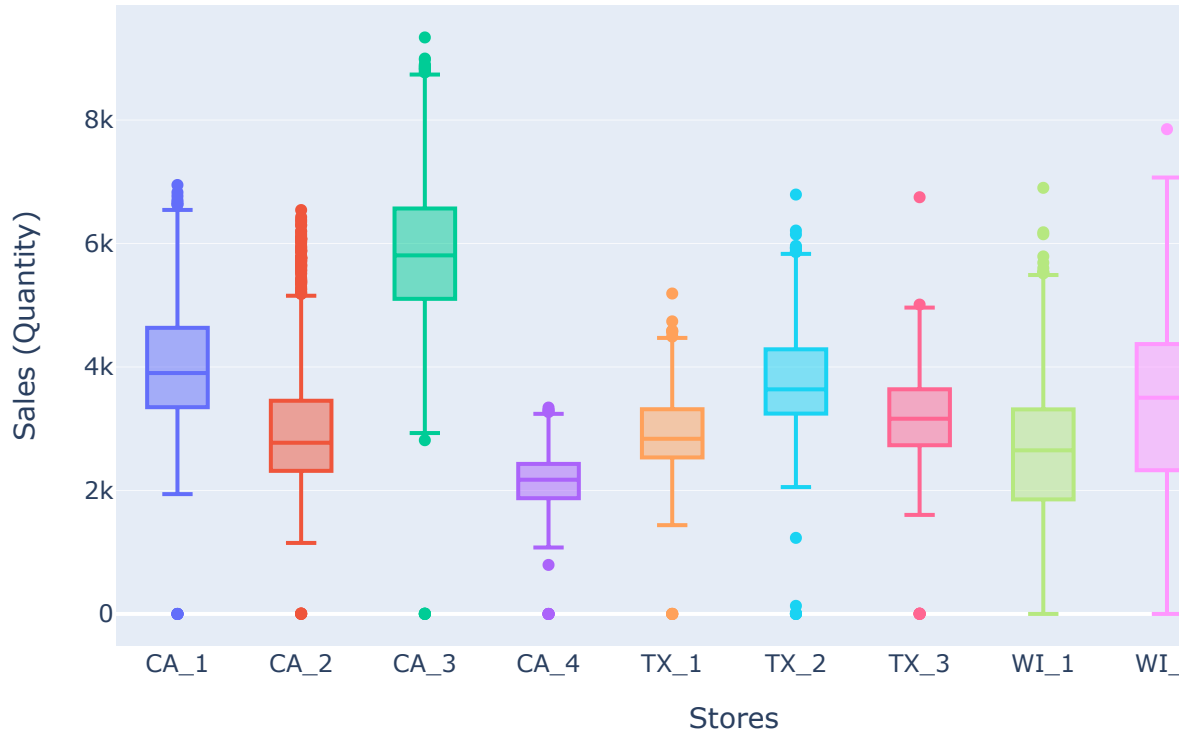
```
# Plot the rolling average for items sold in each store - Boxplot
fig = go.Figure()

for i, s in enumerate(store_ids):
        store_items = [c for c in df_items.columns if s in c]
        plot_data = df_items[store_items].sum(axis=1).rolling(30).mean()
        fig.add_trace(go.Box(x = [s]*len(plot_data),
                             y = plot_data,
```

```
                            name=s))

fig.update_layout(height = 500, width = 800,
                  title="Rolling Average of Sales per Store (boxplot)",
                  xaxis_title = "Stores",
                  yaxis_title = "Sales (Quantity)")

fig.show()
```



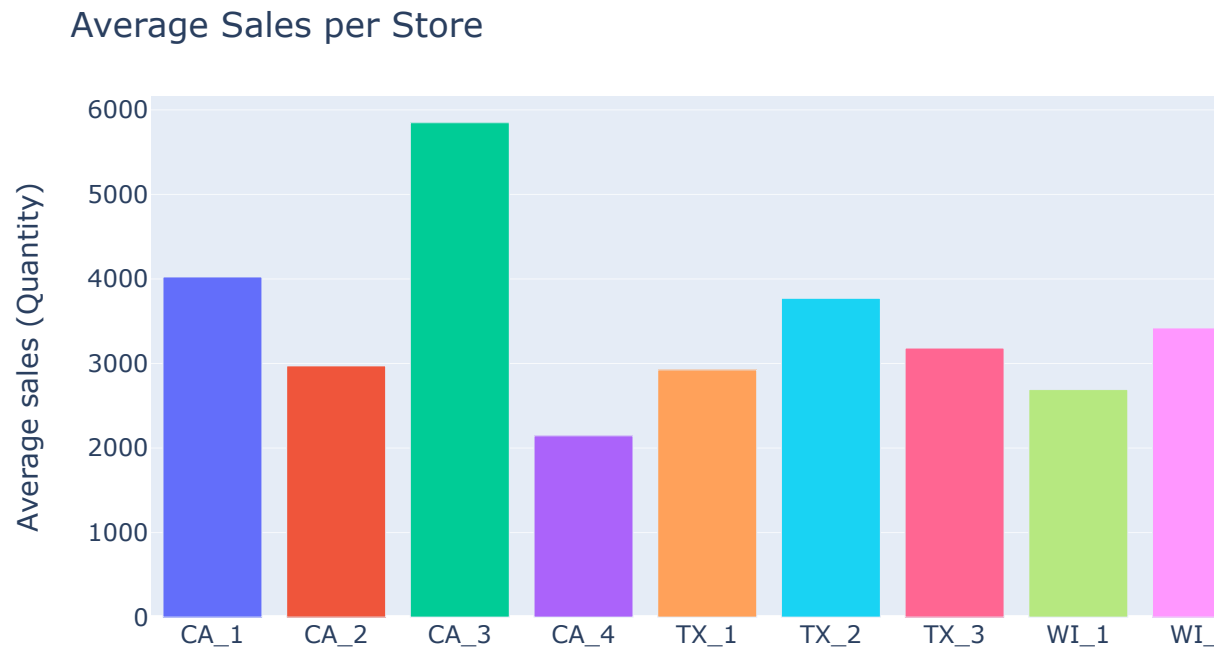
In [64]:

```
# How the boxplot would look like without using a rolling average (including the out
fig = go.Figure()

for i, s in enumerate(store_ids):
        store_items = [c for c in df_items.columns if s in c]
        plot_data = df_items[store_items].sum(axis=1)
        fig.add_trace(go.Box(x = [s]*len(plot_data),
                             y = plot_data,
                             name=s))

fig.update_layout(height = 500, width = 800,
                  title="Sales per Store: Boxplot",
                  xaxis_title = "Stores",
                  yaxis_title = "Sales (Quantity)")

fig.show()
```



In [65]:

```
# Plot the average sales per store - Barchart
fig = px.bar(df_stores_avg,
             x = store_ids,
             y = df_stores_avg.iloc[:,0],
             color = store_ids,
             labels = {'color':'Store','x':'Store','y':'Average Sales'})

fig.update_layout(height = 400, width = 800,
                  title_text = 'Average Sales per Store',
                  xaxis_title = "Stores",
                  yaxis_title = "Average sales (Quantity)")

fig.show()
```



Stores

Observations & Findings:

- Most sales curves exhibit a "linear oscillation" trend: sales grow linearly in the long run despite some short-term fluctuations
- The stores in California seem to have the highest variance in sales, indicating that some of the state's stores grow significantly faster than others
- The stores in Wisconsin and Texas are quite consistent in sales, without much variance, indicating that development was more stable in these states
- The California stores seem to have the highest overall mean sales

4.2.3 Sales per Category

Products sold in each store are separated into 3 categories: Foods, Hobbies, and Household

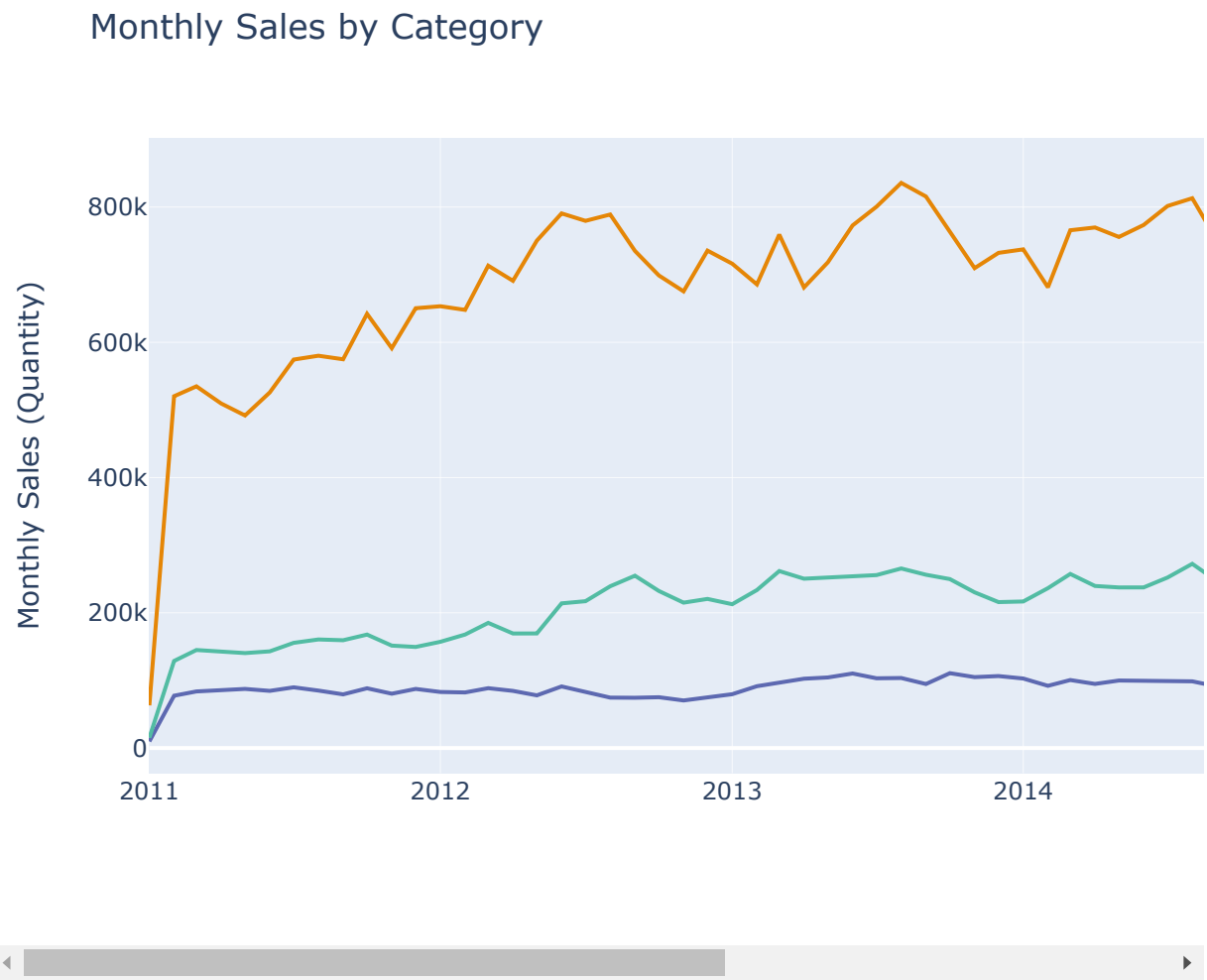
In [66]:
```
# Separate the daily sales by state
# separate the daily sales by state from the df_sales dataset
df_states = df_sales.loc[:,"state_id":df_sales.columns[-1]].groupby("state_id").sum(
# transpose and configure the index
df_states = df_states.transpose().reset_index().rename(columns={'index':'d'})
# merge with the related df_calendar rows
df_states = pd.merge(df_states, df_calendar.loc[0:len(df_states)], on='d').fillna(0)
# set the date column as datetime type
df_states['date'] = pd.to_datetime(df_states['date'])
```

In [67]:
```
# Separate the daily sales by category from the initial sales dataset
df_categories = df_sales.groupby("cat_id")[date_col].sum().T

# Configure the index to include the dates
base = datetime.datetime(2011,1,29) # set the start date
df_categories['date'] = [base + datetime.timedelta(days=x) for x in range(len(date_c
df_categories = df_categories.set_index('date', drop=True).sort_index()
```

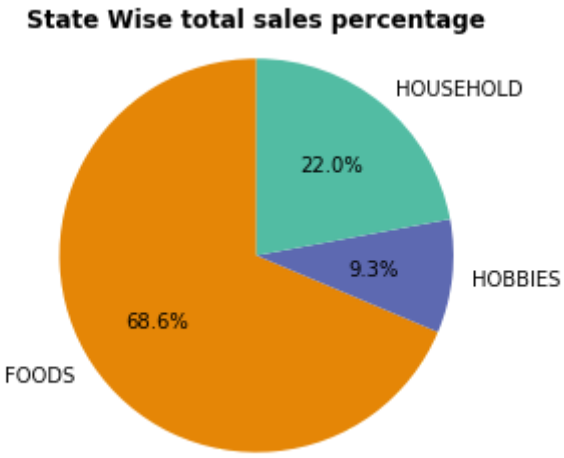
In [68]:
```
# Several aggregation levels for the sales by category
# Monthly
df_categories_monthly = df_categories.groupby(pd.Grouper(freq='MS')).sum().filter(it

# Yearly
df_categories_yearly = df_categories.groupby(pd.Grouper(freq='YS')).sum().filter(ite

# Overall - Sum
df_categories_sum = pd.DataFrame(df_categories.filter(items=categ_ids, axis=1).sum(a

# Overall - Average
df_categories_avg = pd.DataFrame(df_categories.filter(items=categ_ids, axis=1).mean(
```

In [69]:
```
# Calculate the ratio of sales per category (to overall sales)
df_categ_ratio = df_categories_sum/df_categories_sum.sum()*100
```

In [70]:
```
# Plot monthly sales by category
fig = go.Figure()

for i in range(len(categ_ids)):
  fig.add_trace(go.Scatter(x = df_categories_monthly.index, y = df_categories_monthl
                           name =categ_ids[i],
                           line_color = px.colors.qualitative.Vivid[i]))

fig.update_layout(height = 500, width = 1000,
                  title_text = "Monthly Sales by Category",
                  yaxis_title = "Monthly Sales (Quantity)")

fig.show()
```

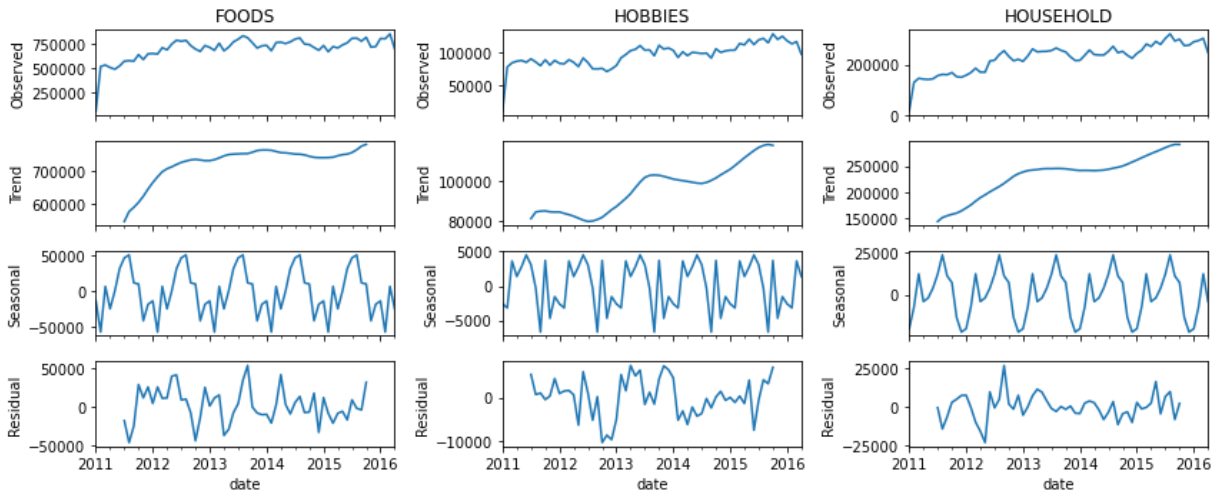


In [71]:
```
# Plot the sales per category ratio
fig, ax = plt.subplots()

ax.pie(df_categ_ratio.iloc[:,0],
       labels = df_categ_ratio.index,
       autopct = '%1.1f%%',
       shadow = False,
       colors = ['#e58606','#5d69b1','#52bca3'],
       startangle = 90)

ax.axis('equal')

plt.title("State Wise total sales percentage", fontweight = "bold")

plt.show()
```



In [72]:

```
# Plot the decomposition components for each category timeseries - monthly aggregati
fig, axes = plt.subplots(ncols=3, nrows=4, sharex=True, figsize=(12,5))

for i in range(len(categ_ids)):
    plotdecomp(seasonal_decompose(df_categories_monthly.iloc[:,i], model='additive')
    axes[0,i].set_title(categ_ids[i])

plt.tight_layout()

plt.show()
```



Observations & Findings:

- Sales of FOODS items have always been much higher than sales for HOUSEHOLD and HOBBIES items
- FOODS sales:
  - 69% of all items sold
  - increased more in the benigging and have remained somewhat stagnant since 2012
  - exhibited a clear yearly seasonality with two spikes that occur in August and March
- HOBBIES sales:
  - 9% of all items sold
  - had a more flat trend with a less clear seasonality
  - the spikes in March are closer to the August spikes in terms of volume
  - sales increased from Aug-2012 to Aug-2013 and then from Aug-2014 to Aug-2015, remaining stagnant in-between

- HOUSEHOLD sales:
  - 22% of all items sold
  - had the most increasing trend over the years
  - exhibited a clear seasonality in March and August

4.2.4 Sales per Category and State

Plot the sales for each of the categories in each of the states.

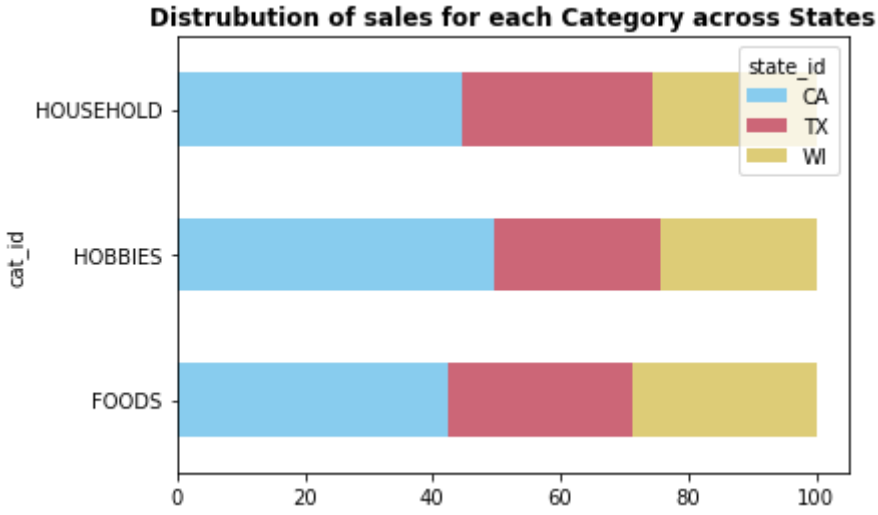
In [73]:
```
# Create a dataframe that includes daily sales for each State & Category combination
# Group the daily sales by state and category
df_state_categ = df_sales.groupby(["state_id",
                                   "cat_id"])[date_col].sum().reset_index().set_inde


# Configure the index to include the dates
df_state_categ['date'] = [base + datetime.timedelta(days=x) for x in range(len(date_
df_state_categ = df_state_categ.set_index('date', drop=True).sort_index()

# Configure the columns to reference the STATE_CATEGORY combination
df_state_categ.columns = [f'{i}_{j}' if j != '' else f'{i}' for i,j in df_state_cate

df_state_categ.head()
```

Out[73]:

date	CA_FOODS	CA_HOBBIES	CA_HOUSEHOLD	TX_FOODS	TX_HOBBIES	TX_HOUSEHOLD	WI_FO
2011-01-29	10101	1802	2292	6853	879	1706	
2011-01-30	9862	1561	2382	7030	870	1730	
2011-01-31	6944	1472	1692	5124	526	1128	
2011-02-01	7864	1405	1778	5470	809	1102	
2011-02-02	7178	1181	1566	4602	501	809	

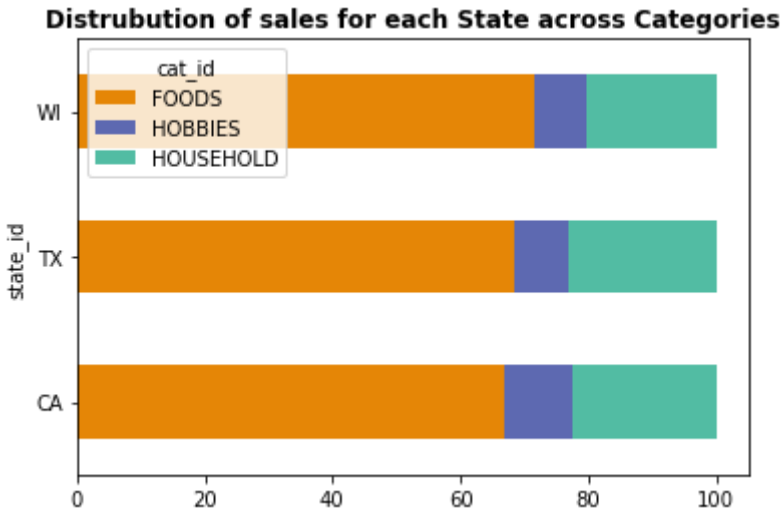
In [74]:
```
# Calculate the total sales for each Category across States
df_categ_state_sum = df_sales.groupby(['cat_id',
                                       'state_id']).sum()[date_col].sum(axis=1)
                                        \.groupby(level=0).apply(lambda x: 100 * x /
```

In [75]:
```
# Plot the total sales for each Category across States
df_categ_state_sum.plot(kind = 'barh',
                        stacked = True,
                        colormap = Safe_3.mpl_colormap)

plt.title("Distrubution of sales for each Category across States", fontweight = "bol

plt.show()
```



In [76]:
```
# Calculate the total sales for each State across Categories
df_state_categ_sum = df_sales.groupby(['state_id','cat_id'])
                    \.sum()[date_col].sum(axis=1).groupby(level=0).apply(lambda x: 1
```

In [77]:
```
# Plot the total sales for each State across Categories
df_state_categ_sum.plot(kind = 'barh',
                        stacked = True,
                        colormap = Vivid_3.mpl_colormap)

plt.title("Distrubution of sales for each State across Categories", fontweight = "bo

plt.show()
```



In [78]:
```
# Monthly aggregation for sales by State & Category
df_state_categ_monthly = df_state_categ.groupby(pd.Grouper(freq='MS')).sum()
```

In [79]:
```
# Plot the monthly sales by State & Category
cf.Figure(cf.subplots([df_state_categ_monthly[[col for col in df_state_categ_monthly
                       \.figure(colors=px.colors.qualitative.Safe),
                       df_state_categ_monthly[[col for col in df_state_categ_monthly
                       \.figure(colors=px.colors.qualitative.Safe),
                       df_state_categ_monthly[[col for col in df_state_categ_monthly
                       \.figure(colors=px.colors.qualitative.Safe)],
                      shape = (1,3),
                      subplot_titles = (categ_ids),
```

```
            )
        ).iplot()
```

Observations & Findings:

From a State perspective

- California
  - spent 67% on FOODS, 11% on HOBBIES and 22% on HOUSEHOLD items
  - accounts for more than 40% of the items sold in the FOODS and HOUSEHOLD categories and about 50% of items sold in HOBBIES category
- Texas
- spent 69% on food, 8% in hobbies and 23% in household
  - has a 30% share of the items sold in FOODS and HOUSEHOLD categories and about 25% of items sold in HOBBIES category
- Winscoin
  - spent 72% on FOODS, 8% on HOBBIES and 20% on HOUSEHOLD items
  - has a 25% share of the items sold in HOBBIES and HOUSEHOLD categories and about 30% of the items sold in FOODS category

From a Category perspective

- FOODS

    - Sales of FOODS items in California have had a high yearly seasonality with peaks in August but overall sales have not increased significantly after 2012
    - Sales of FOODS items in Texas have had a similar trend as California (although at a lower scale) with high monthly seasonality peaking in August and dipping around January-February with the overall trend remaining the same after 2012
    - Sales of FOODS items in Wisconsin have not had any clear seasonality while increasing significantly over the years, especially comparing 2011 to 2016
    - Wisconsin was essentially the only state showing an increase in FOODS items sold over the years
- HOBBIES
    - Looking at all states, sales of HOBBIES items did not seem to have any visible seasonalities
    - In California, sales of HOBBIES items appear to increase every second year and relatively decrease in-between; this is especially noticable at the drop starting in Aug-12 which was followed by an increase starting Jan-13
        - Overall, sales of HOBBIES items in California have had an increasing trend
    - The sales of HOBBIES items in Texas and Wisconsin also showed a similar increasing trend as California, with Texas performing better than Wisconsin
- HOUSEHOLD
    - HOUSEHOLD sales have had the most consistenly and clearly increasing trend amongst the three categories, in all three states
    - California had the highest increase over the years and a clear yearly seasonality peaking in August and dipping around December-January
    - Texas had overall better sales than Wisconsin throughout the years but both show similarly increasing trend. They also both have a weak seasonality with relatively visible peaks in August and March

4.2.5 Sales per Department

Each of the categories is further separated into 2 or 3 departments:

- FOOD is separated into 3 departments
- HOBBIES is separated into 2 departments
- HOUSEHOLD is separated into 2 departments

In [80]:
```
# Create a dataframe that includes daily sales for each Department
# Group the daily sales by department
df_dept = df_sales.groupby("dept_id")[date_col].sum().reset_index().set_index("dept_

# Configure the index to include the dates
df_dept['date'] = [base + datetime.timedelta(days=x) for x in range(len(date_col))]
df_dept = df_dept.set_index('date', drop=True).sort_index()
```

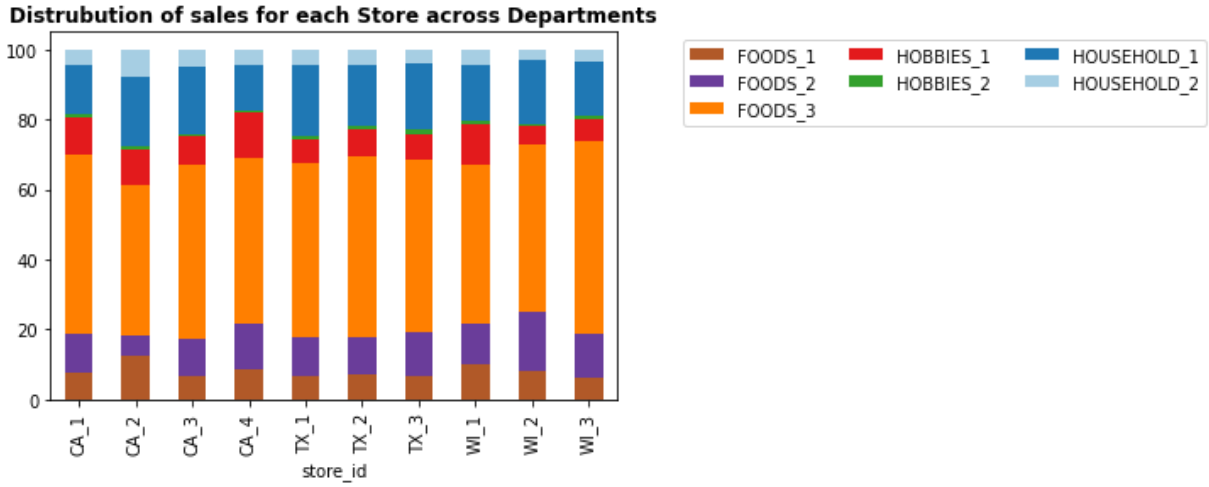
In [81]:
```
# Monthly aggregation for sales by Department
df_dept_monthly = df_dept.groupby(pd.Grouper(freq='MS')).sum()
```

In [82]:
```
# Plot the monthly sales by Department
```

```
fig = cf.Figure(cf.subplots([df_dept_monthly[[col for col in df_dept_monthly.columns
                             \.figure(colors=px.colors.qualitative.Set1),
                             df_dept_monthly[[col for col in df_dept_monthly.columns
                             \.figure(colors=px.colors.qualitative.Set2),
                             df_dept_monthly[[col for col in df_dept_monthly.columns
                             \.figure(colors=px.colors.qualitative.D3)],
                            shape = (1,3),
                            subplot_titles = (categ_ids)
                            )
                ).iplot()
```

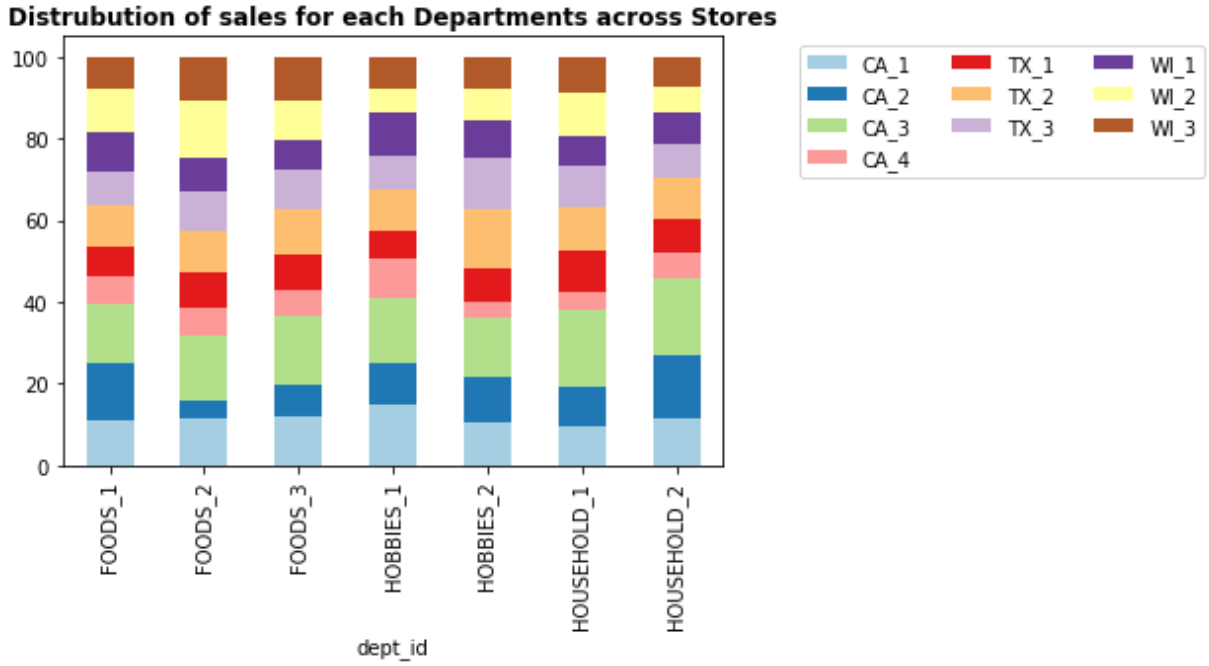
In [83]:
```
# Calculate the total sales for each Store across Departments
df_store_dept_sum = df_sales.groupby(['store_id','dept_id']).sum()[date_col].sum(axi
                                    \.groupby(level=0).apply(lambda x: 100 * x / flo
```

In [84]:
```
# Plot the total sales for each Store across Departments
df_store_dept_sum.plot(kind = 'bar',
                       stacked = True,
                       colormap = 'Paired_r')

plt.title("Distrubution of sales for each Store across Departments", fontweight = "b

plt.legend(loc='upper left', bbox_to_anchor=(1.1, 1), ncol=3)

plt.show()
```



In [85]:
```
# Calculate the total sales for each Departments across Stores
df_dept_store_sum = df_sales.groupby(['dept_id','store_id']).sum()[date_col]
                                    \.sum(axis=1).groupby(level=0).apply(lambda x: 1
```

In [86]:
```
# Plot the total sales for each Departments across Stores
df_dept_store_sum.plot(kind = 'bar',
                       stacked = True,
                       colormap = 'Paired')

plt.title("Distrubution of sales for each Departments across Stores", fontweight = "

plt.legend(loc='upper left', bbox_to_anchor=(1.1, 1), ncol=3)

plt.show()
```



Observations & Findings:

From a department perspective

- FOODS_1 remained relatively stagnant over the years, especially after 2012

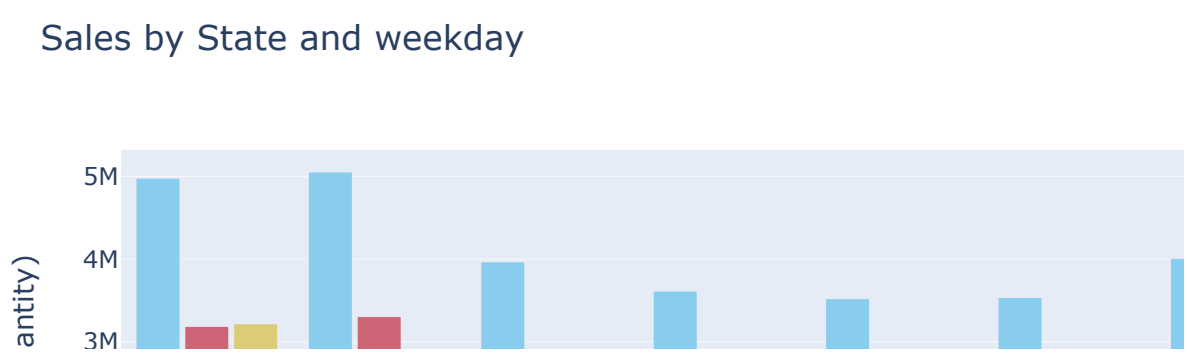
- FOODS_2 increased especially over the years 2015-2016, going from a higher value of 130k to more than 170k
- FOODS_3 was the department with the most items sold
  - it also had a high seasonality ranging from 500k to 600k throughout the year, with peaks around the summer months
- HOUSEHOLD_1 had the highest increase in sales over the years in comparison to all other departmenrs, doubling from about 100k in 2011 to more than 200k in 2016
- HOUSEHOLD_2 has been steadily increasing with a very clear seasonality of peaking around the summer months
- HOBBIES_1 included most of the items sold under HOBBIES category, while HOBBIES_2 was at much lower levels
- HOBBIES_2 had a high seasonality with peaks in October of each year

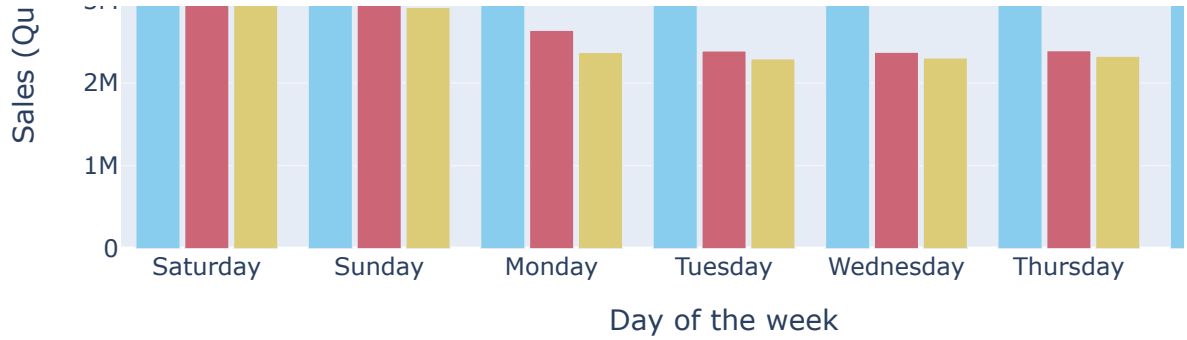
4.3 Seasonalities study

4.3.1 Day of Week Seasonality

In [87]:
```
# Sales by state and weekday
df_states_weekday = df_states.groupby("weekday").sum().loc[:,state_ids]
                    \.reindex(['Saturday', 'Sunday', 'Monday', 'Tuesday', 'Wednesday
```

In [88]:
```
# Sales by state and weekday - Barchart
fig = go.Figure()

for i in range(len(state_names)):
  fig.add_trace(go.Bar(x = df_states_weekday.index,
                       y = df_states_weekday[state_ids[i]],
                       name = state_names[i],
                       marker_color = px.colors.qualitative.Safe[i]))

fig.update_layout(height = 400, width = 800,
                  title_text = 'Sales by State and weekday',
                  xaxis_title = "Day of the week",
                  yaxis_title = "Sales (Quantity)",
                  legend=dict(x = 1.0, y = 1.0,
                              bgcolor='rgba(255, 255, 255, 0)',
                              bordercolor='rgba(255, 255, 255, 0)'),
                  barmode='group',
                  bargap=0.15, # gap between bars of adjacent location coordinates
                  bargroupgap=0.1 # gap between bars of the same location coordinate
                  )

fig.show()
```



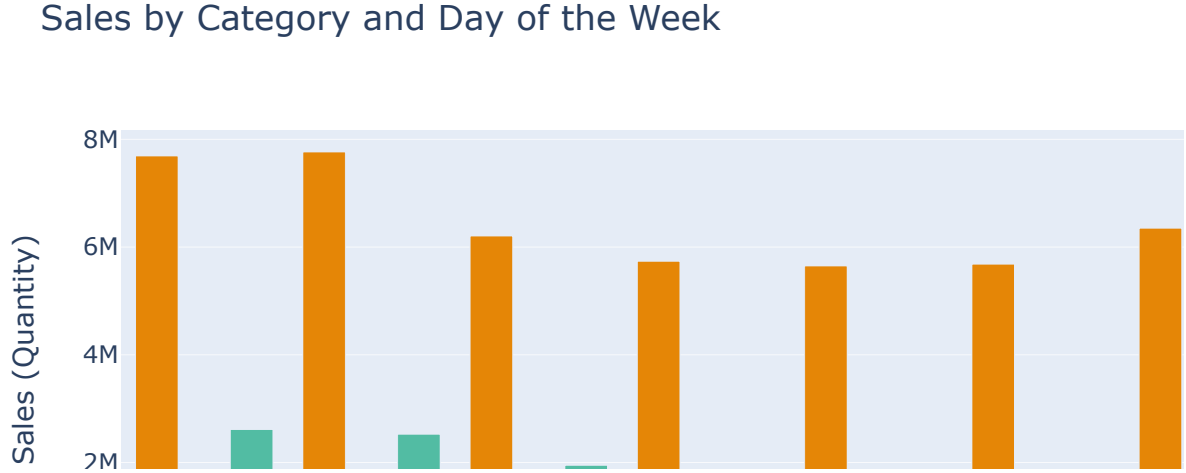


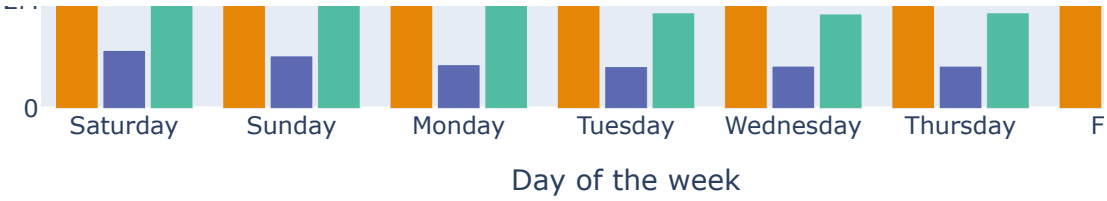
In [89]:

```
# Sales by category and weekday
#setup the df_calendar dataframe in order to merge with df_categories
df_calendar['date'] = pd.to_datetime(df_calendar['date'])

df_categories = pd.merge(df_categories,df_calendar.loc[0:len(df_categories)], on='da

df_categories_weekday = df_categories.groupby("weekday").sum().loc[:,categ_ids]
                        \.reindex(['Saturday', 'Sunday', 'Monday', 'Tuesday', 'Wedne
```

In [90]:

```
# Sales by category and weekday - Barchart
fig = go.Figure()

for i in range(len(categ_ids)):
  fig.add_trace(go.Bar(x = df_categories_weekday.index,
                       y = df_categories_weekday[categ_ids[i]],
                       name = categ_ids[i],
                       marker_color = px.colors.qualitative.Vivid[i]))

fig.update_layout(height = 400, width = 800,
                  title_text = 'Sales by Category and Day of the Week',
                  xaxis_title = "Day of the week",
                  yaxis_title = "Sales (Quantity)",
                  legend=dict(x = 1.0, y = 1.0,
                              bgcolor='rgba(255, 255, 255, 0)',
                              bordercolor='rgba(255, 255, 255, 0)'),
                  barmode='group',
                  bargap=0.15, # gap between bars of adjacent location coordinates
                  bargroupgap=0.1 # gap between bars of the same location coordinate
                  )
fig.show()
```





Observations & Findings:

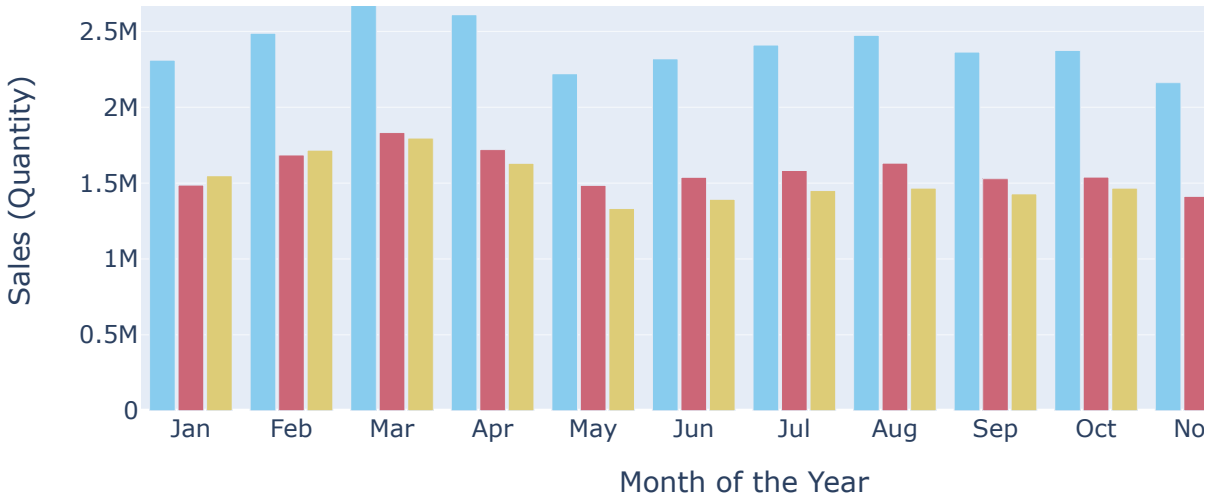
- Sales by State: There is a clear weekly seasonality with sales being higher during and closer to the weekend (Friday through Monday) and lower in the middle of the week (Tuesday through Thursday)
- Sales by Category: The weekly seasonality is also reflected here, with sales for all three categories being higher around the weekend (Friday through Monday) and lower in the middle of the week (Tuesday through Thursday)
  - FOODS items have more sales on Sundays (7.8M), followed by Saturdays (7.7M)
  - HOBBIES and HOUSEHOLD items have more sales on Saturdays compared to Sundays, however the differences between the two days are not that significant

4.3.2 Month of Year Seasonality

In [91]:
```
# Sales by state and months
df_states_months = df_states.groupby("month").sum().loc[:,state_ids]
                                    \.rename(index=dict(zip([1,2,3,4,5,6,7,8,9,10,11
```

In [92]:
```
# Sales by state and month - Barchart
fig = go.Figure()

for i in range(len(state_ids)):
  fig.add_trace(go.Bar(x = df_states_months.index,
                       y = df_states_months[state_ids[i]],
                       name = state_names[i],
                       marker_color = px.colors.qualitative.Safe[i]))

fig.update_layout(height = 400, width = 800,
                  title_text = 'Sales by State and Month of the Year',
                  xaxis_title = "Month of the Year",
                  yaxis_title = "Sales (Quantity)",
                  legend=dict(x = 1.0, y = 1.0,
                              bgcolor='rgba(255, 255, 255, 0)',
                              bordercolor='rgba(255, 255, 255, 0)'),
                  barmode='group',
                  bargap=0.15, # gap between bars of adjacent location coordinates
                  bargroupgap=0.1 # gap between bars of the same location coordinate
                  )
fig.show()
```

Sales by State and Month of the Year



Observations & Findings:

- In California and Texas, sales were higher during the months of March and April and lower on the months towards the end of the year (November-December)
- In the case of Wisconsi, May has been the month with the lowest sales throughout the year

5 Explanatory Variables: Events and Prices

Go to contents

5.1 Events

In [93]:
```
# Show all event names per event type
# Events are captured in two columns so we will gather all of them and eliminate dup
events_1 = df_calendar[['event_type_1','event_name_1',]] # events that are included
events_2 = df_calendar[['event_type_2','event_name_2',]] # events that are included
events_2.columns = ["event_type_1","event_name_1"]
events = pd.concat([events_1,events_2],ignore_index = True)
events = events.dropna().drop_duplicates()
events_dict = {k: g["event_name_1"].tolist() for k,g in events.groupby("event_type_1
events_df = pd.DataFrame(dict([ (k,pd.Series(v)) for k,v in events_dict.items()]))
events_df
```

Out[93]:

	Cultural	National	Religious	Sporting
0	ValentinesDay	PresidentsDay	LentStart	SuperBowl
1	StPatricksDay	MemorialDay	LentWeek2	NBAFinalsStart
2	Cinco De Mayo	IndependenceDay	Purim End	NBAFinalsEnd
3	Mother's day	LaborDay	OrthodoxEaster	NaN
4	Father's day	ColumbusDay	Pesach End	NaN
5	Halloween	VeteransDay	Ramadan starts	NaN
6	Easter	Thanksgiving	Eid al-Fitr	NaN

	Cultural	National	Religious	Sporting
7	NaN	Christmas	EidAlAdha	NaN
8	NaN	NewYear	Chanukah End	NaN
9	NaN	MartinLutherKingDay	OrthodoxChristmas	NaN

In [94]:
```
# Days when a store had been closed (aka had almost zero sales)
df_closed_stores = df_stores.loc[df_stores['WI_1']<10,['date','event_name_1']] # we
df_closed_stores
```

Out[94]:

	date	event_name_1
4	2011-02-02	0
330	2011-12-25	Christmas
696	2012-12-25	Christmas
1061	2013-12-25	Christmas
1426	2014-12-25	Christmas
1791	2015-12-25	Christmas

Observations & Findings:

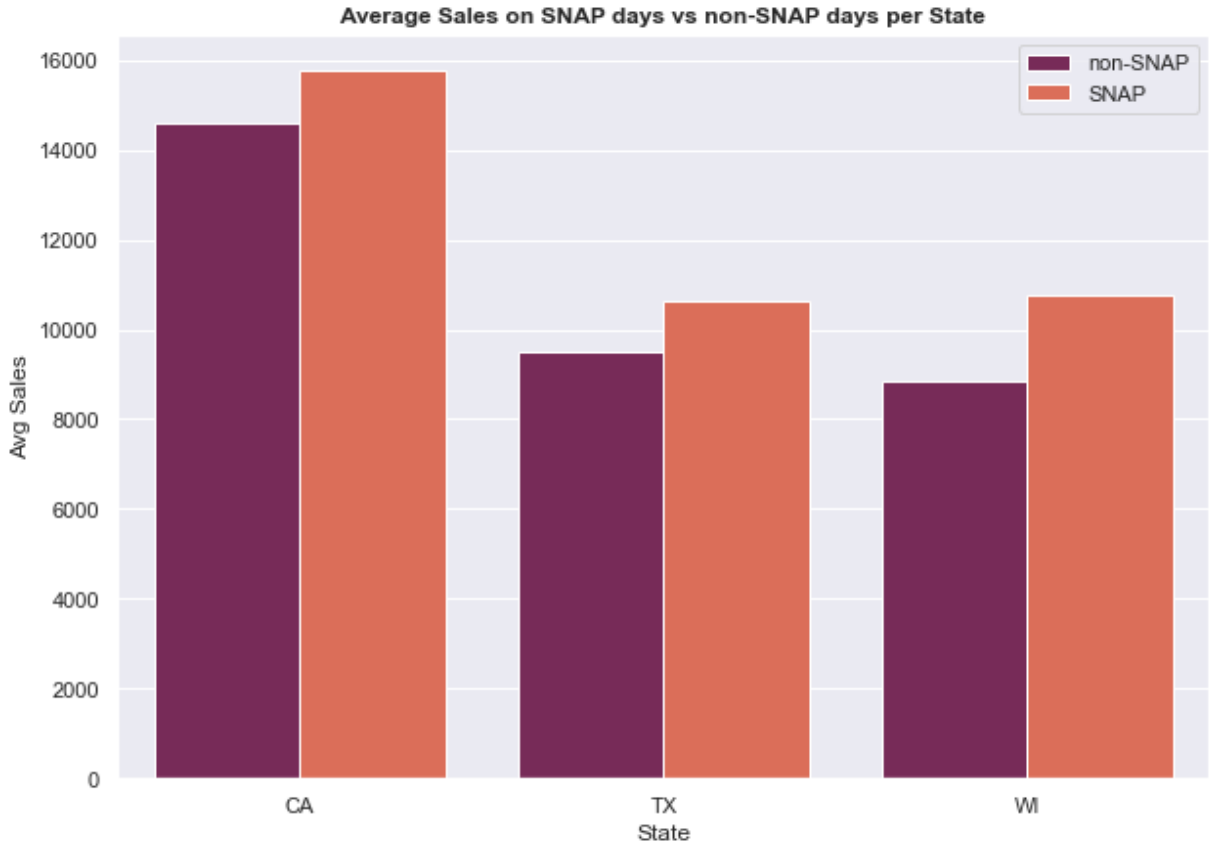
- There are 4 different event types and each of them includes several different event names (ranging from 3 to 9 per event type). In total, there are 27 different event names.
- Christmas day was the only day throughout the year that all Walmart stores were closed.
- Additionaly, WI_1 store was also closed on Feb 2nd, 2011 which was due to a strong blizzard that occured on that day shutting down southern Wisconsin (Groundhog Day Blizzard, 2011).

5.1.1 Supplemental Nutrition Assistance Program (SNAP)

In [95]:
```
# Impact of SNAP days on average overall sales by state
snap_ca_average = df_states.groupby("snap_CA")["CA"].mean().reset_index().rename(col
snap_tx_average = df_states.groupby("snap_TX")["TX"].mean().reset_index().rename(col
snap_wi_average = df_states.groupby("snap_WI")["WI"].mean().reset_index().rename(col

# Merge into one dataframe the SNAP average sales per state
df_snap_impact = snap_ca_average.merge(snap_tx_average,on='SNAP').merge(snap_wi_aver

# Unpivot the dataframe with SNAP as the identifier variable and State, Avg Sales as
df_snap_impact = pd.melt(df_snap_impact,
                         id_vars = ['SNAP'],
                         value_vars = ['CA','TX','WI'],
                         var_name = 'State',
                         value_name = 'Avg Sales').replace({'SNAP': {0: 'non-SNAP',
```

In [96]:
```
# Average sales of SNAP vs non-SNAP days per state - Barchart

sns.set(rc={'figure.figsize':(10,7)})

chart = sns.barplot(x = "State", y = 'Avg Sales', hue = 'SNAP', data = df_snap_impac
```

```
plt.title("Average Sales on SNAP days vs non-SNAP days per State", fontweight="bold"

leg = plt.legend()
```



In [97]:

```
p1 = (snap_ca_average.loc[1,state_ids[0]]/snap_ca_average.loc[0,state_ids[0]]-1)
print("Difference between SNAP vs non-SNAP days in average sales in California: {:.1

p2 = (snap_tx_average.loc[1,state_ids[1]]/snap_tx_average.loc[0,state_ids[1]]-1)
print("Difference between SNAP vs non-SNAP days in average sales in Texas: {:.1%}".f

p3 = (snap_wi_average.loc[1,state_ids[2]]/snap_wi_average.loc[0,state_ids[2]]-1)
print("Difference between SNAP vs non-SNAP days in average sales in Wisconsin: {:.1%
```

```
Difference between SNAP vs non-SNAP days in average sales in California: 8.0%
Difference between SNAP vs non-SNAP days in average sales in Texas: 11.5%
Difference between SNAP vs non-SNAP days in average sales in Wisconsin: 21.8%
```

Observations & Findings:

Households can use SNAP to buy nutritious foods such as breads and cereals, fruits and vegetables, meat and fish and dairy products. SNAP benefits cannot be used to buy any kind of alcohol or tobacco products or any nonfood items like household supplies and vitamins and medicines (further info found here).

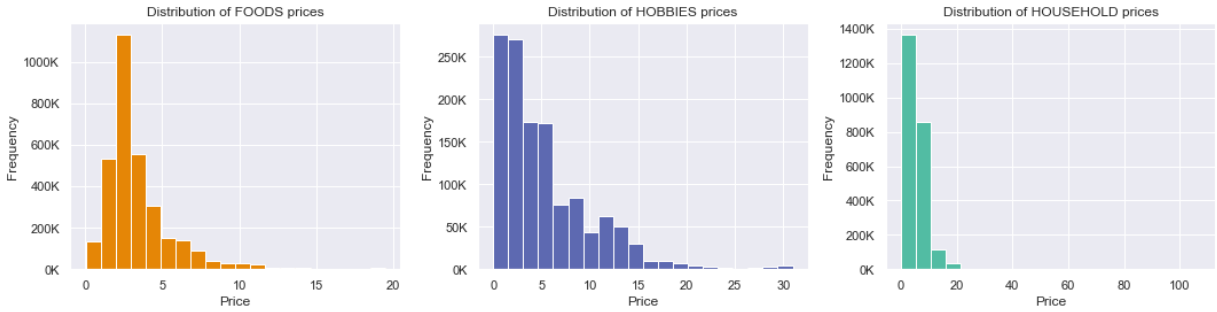
- An assumption from the description is that SNAP days whould mostly affect sales of items in FOOD category.
- All states had higher average sales on SNAP days versus non-SNAP days
  - California average sales were 8.0% more on SNAP vs non-SNAP days
  - Texas average sales were 11.5% more on SNAP vs non-SNAP days
  - Wisconsin average were are 21.8% more on SNAP vs non-SNAP days

5.2 Item Prices

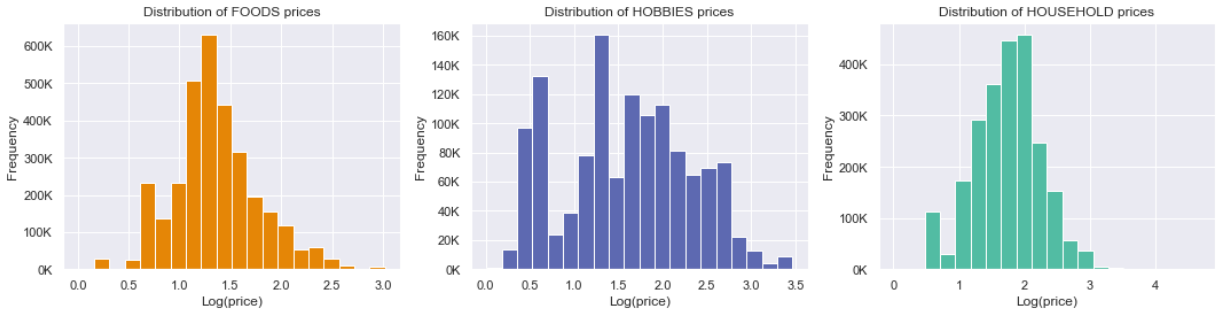
Explore the impact of item prices on each of the item categories.

In [98]:
```
# Add Category column to the df_prices dataset (derived from the item_id)
df_prices["category"] = df_prices["item_id"].str.split("_",expand = True)[0]
```

In [99]:
```
# Distribution of item prices by category (normal scale)
categ_colors = ['#e58606','#5d69b1','#52bca3']

fig, axs = plt.subplots(1, 3, figsize=(15, 4))
i = 0
for cat, d in df_prices.groupby('category'):
    ax = d['sell_price'].plot(kind = 'hist',
                              bins = 20,
                              title = f'Distribution of {cat} prices',
                              ax = axs[i],
                              color = categ_colors[i])
    ax.yaxis.set_major_formatter(ticker.FuncFormatter(lambda y, pos: '%.0fK' % (y *
    ax.set_xlabel('Price')
    i += 1

plt.tight_layout()
```



In [100…
```
# Distribution of item prices by category (log scale)
categ_colors = ['#e58606','#5d69b1','#52bca3']

fig, axs = plt.subplots(1, 3, figsize=(15, 4))
i = 0
for cat, d in df_prices.groupby('category'):
    ax = d['sell_price'].apply(np.log1p).plot(kind = 'hist',
                                              bins = 20,
                                              title = f'Distribution of {cat} prices
                                              ax = axs[i],
                                              color = categ_colors[i])
    ax.yaxis.set_major_formatter(ticker.FuncFormatter(lambda y, pos: '%.0fK' % (y *
    ax.set_xlabel('Log(price)')
    i += 1

plt.tight_layout()
```



In [101…

```
# Create a dataframe including the maximum and minimum prices for each item in each
df_item_store_prices = df_prices.groupby(["item_id","store_id"]).agg({"sell_price":[
df_item_store_prices.columns = [f'{i}_{j}' if j != '' else f'{i}' for i,j in df_item
df_item_store_prices["price_change"] = df_item_store_prices["sell_price_max"] - df_i

# Create a sorted dataframe based on the price change over the years for each item i
df_item_store_prices_sorted = df_item_store_prices.sort_values(["price_change","item
df_item_store_prices_sorted["category"] = df_item_store_prices_sorted["item_id"].str
```

In [102…

```
# Show the top 10 items with biggest price change over the years
df_item_store_prices_sorted.head(10)
```

Out[102…

	item_id	store_id	sell_price_max	sell_price_min	price_change	category
0	HOUSEHOLD_2_406	WI_3	107.32	3.26	104.06	HOUSEHOLD
1	HOUSEHOLD_2_406	WI_2	61.46	12.46	49.00	HOUSEHOLD
2	HOUSEHOLD_2_466	TX_1	52.62	6.46	46.16	HOUSEHOLD
3	HOUSEHOLD_2_178	TX_1	44.36	3.00	41.36	HOUSEHOLD
4	HOUSEHOLD_2_250	WI_2	34.18	3.36	30.82	HOUSEHOLD
5	HOUSEHOLD_2_406	WI_1	35.88	9.97	25.91	HOUSEHOLD
6	HOUSEHOLD_2_250	WI_1	30.32	4.97	25.35	HOUSEHOLD
7	HOUSEHOLD_1_469	WI_3	19.97	1.00	18.97	HOUSEHOLD
8	HOUSEHOLD_2_514	TX_2	19.54	1.00	18.54	HOUSEHOLD
9	HOUSEHOLD_1_342	WI_3	17.97	1.00	16.97	HOUSEHOLD

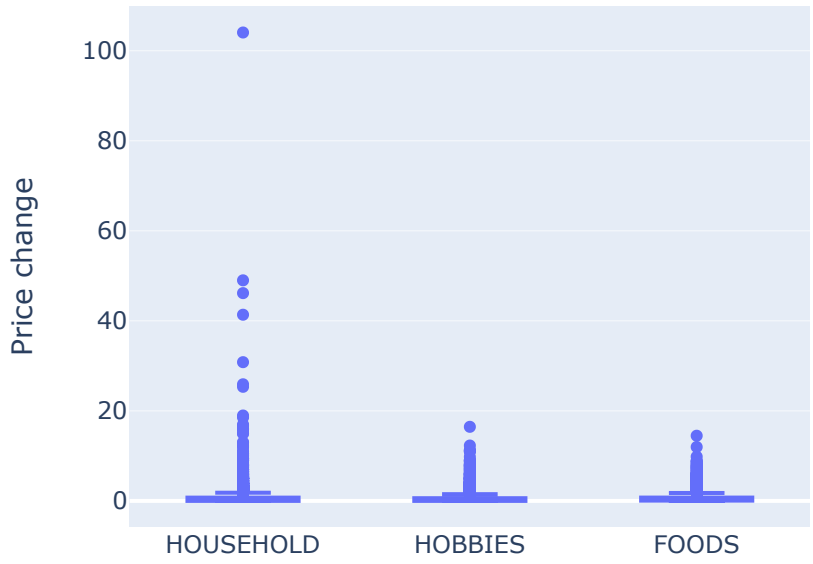
In [103…

```
# Check how many items had no price change throughout the years
steady_price = (df_item_store_prices['price_change'] == 0.0).sum(axis=0)/len(df_item
print("{:.0%} of all items exhibited no price change throught the years".format(stea
```

```
27% of all items exhibited no price change throught the years
```

In [104…

```
# Plot of price changes over the years - boxplot
fig = px.box(df_item_store_prices_sorted,
             x = "category",
             y = "price_change")

fig.update_layout(height = 400, width = 500,
                  title="Max price change across all categories",
                  xaxis_title = "",
                  yaxis_title = "Price change")

fig.show()
```



Observations & Findings:

A logarithmic price scale is prefered for the analysis so that the smaller prices are represented equally on the chart. This type of scale is generally used for long-term perspective analysis of price changes (source). Furthermore, we use log1p here (natural algorithm+1) since we are looking at very low prices (source)

Price ranges

- Most of the prices for FOODS items lie between $1-10
  - The peaks at $3 corresponds to almost 2 million of FOODS items that have around this price
- HOBBIES items have a wide range of prices
  - About 900K of the HOBBIES items are below $5
- HOUSEHOLDS items are generaly more expensive than FOODS items (peak closer to $10)

Price changes

- Most products do not change their price throughout the years (majority of price changes concentrated at the base of the box plot)
  - more specifically, 27% of all items exhibited no price change throught the years
- The changes that do happen are mostly restricted to below $15
- Household items have the highest price changes over the years

6 Feature Engineering

Go to contents

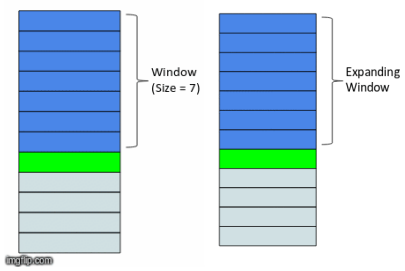
The purpose of feature engineering here is to convert our time series problem into a supervised learning problem in order to build regression models.

(Output for models) Variable to be predicted: items sold each day

(Input for models) Features to be used to make predictions (constructed with feature engineering):

- Date-related features: components of the time step itself (day of week, month etc)
- Lag Features: values at prior time steps; the value at time t is affected by past values aka lags (t-1, t-2 etc)
  - The lag value we chose will depend on the correlation of individual values with past values (weekly/monthly etc trend)
  - Then e.g. a linear regression model will assign appropriate weights (of coefficients) to the created lag features
  - Ways to determine the lag at which correlation is significant: ACF (autocorrelation function), PACF (partial autocorrelation function), etc
- Window Features: a summary of values over a fixed window of prior time steps
  - Rolling window features (e.g. rolling mean, rolling sum, weighted average etc)
    - Rolling window method calculates statistical values based on past values with a different window for every data point
  - Expanding window features
    - Expanding window method takes all the past values into account since the window size increases for every new values
- Domain specific features
  - Provided a good understanding of the problem statement, clarity of the end objective and knowledge of the available data
  - Take into consideration state, store, category, events etc

Rolling Window vs Expanding Window



6.1 Reducing the data size

Numerical Columns: Use a subtype of int and float which is less memory consuming. Also, use an unsigned subtype if there is no negative value.
Categorical Columns: Force pandas to use a virtual mapping table for categorical columns with low cardinality where all unique values are mapped via an integer instead of a pointer using the category datatype.

In [105…

```
# Calculate the memory usage of the dataframes before downcasting
sales_bd = np.round(df_sales.memory_usage().sum()/(1024*1024),1)
```

```
calendar_bd = np.round(df_calendar.memory_usage().sum()/(1024*1024),1)
prices_bd = np.round(df_prices.memory_usage().sum()/(1024*1024),1)
```

In [10]:

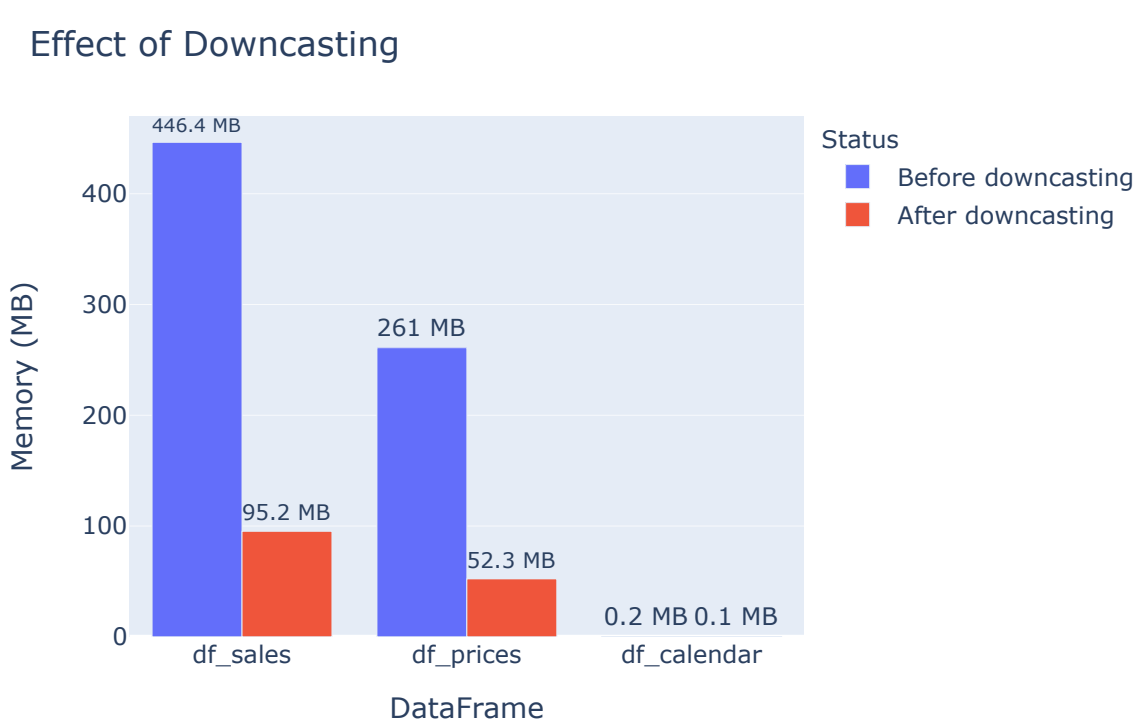
```
# Apply the downcasting function to the dataframes
df_sales = downcast(df_sales)
df_calendar = downcast(df_calendar)
df_prices = downcast(df_prices)
```

In [107…

```
# Calculate the memory usage of the dataframes after downcasting
sales_ad = np.round(df_sales.memory_usage().sum()/(1024*1024),1)
calendar_ad = np.round(df_calendar.memory_usage().sum()/(1024*1024),1)
prices_ad = np.round(df_prices.memory_usage().sum()/(1024*1024),1)
```

In [108…

```
# Plot the memory usage of the dataframes before vs after downcasting
dic = {'DataFrame':['df_sales','df_calendar','df_prices'],
       'Before downcasting':[sales_bd,calendar_bd,prices_bd],
       'After downcasting':[sales_ad,calendar_ad,prices_ad]}

memory = pd.DataFrame(dic)
memory = pd.melt(memory, id_vars='DataFrame', var_name='Status', value_name='Memory
memory.sort_values('Memory (MB)',ascending=False, inplace=True)

fig = px.bar(memory, x='DataFrame', y='Memory (MB)', color='Status', barmode='group'
fig.update_traces(texttemplate='%{text} MB', textposition='outside')
fig.update_layout(height=400, width=600, title='Effect of Downcasting')
fig.show()
```



Observations & Findings:

After downcasting, the size of all three dataframes significantly decreased:

- df_sales decreased by -79%

- df_prices decreased by -78%
- df_calendar decreased by -50%

In [11]:
```
# Save the downcasted dataframes as binary files on the hard drive

df_sales.to_pickle('df_sales.pkl')
df_calendar.to_pickle('df_calendar.pkl')
df_prices.to_pickle('df_prices.pkl')
```

In [109…
```
# Remove unwanted data (used during EDA) to free memory for further processing

del zeros_sales,zeros_prices,zeros_calendar,zeros_days,group,daily_sales,df_states,d
df_states_yearly,df_states_sum,df_states_months,df_stores,df_stores_sum,df_stores_av
df_categories_yearly,df_categories_sum,df_categories_avg,df_categories_weekday,df_st
df_state_categ_monthly,df_dept,df_dept_monthly,df_store_dept_sum,df_dept_store_sum,d
events_2,events,events_df,df_item_store_prices,df_item_store_prices_sorted

gc.collect();
```

In [ ]:
```
# Percentage of current system-wide memory usage
psutil.virtual_memory().percent
```

6.2 Melting and Encoding the data

In [32]:
```
# First we will add days 1942-1969 to the sales dataframe for use in the submission

for d in range(1942,1970):
    col = 'd_' + str(d)
    df_sales[col] = 0
    df_sales[col] = df_sales[col].astype(np.int16)
```

In [6]:
```
# Melting and merging all dataframes into a single one that includes all data

# Unpivot the sales dataframe from wide to long format, setting some columns as iden
df_modeling = pd.melt(df_sales,
                      id_vars=['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'st
                      var_name='d',
                      value_name='sold').dropna()

# Combine all three dataframes into a
df_modeling = pd.merge(df_modeling, df_calendar, on='d', how='left')
df_modeling = pd.merge(df_modeling, df_prices, on=['store_id','item_id','wm_yr_wk'],
```

In [ ]:
```
# Label Encoding all categorical values (transform them to numerical labels)

# Store the categories along with their codes so that they are later retrieved durin
d_id = dict(zip(df_modeling.id.cat.codes, df_modeling.id))
d_item_id = dict(zip(df_modeling.item_id.cat.codes, df_modeling.item_id))
d_dept_id = dict(zip(df_modeling.dept_id.cat.codes, df_modeling.dept_id))
d_cat_id = dict(zip(df_modeling.cat_id.cat.codes, df_modeling.cat_id))
d_store_id = dict(zip(df_modeling.store_id.cat.codes, df_modeling.store_id))
d_state_id = dict(zip(df_modeling.state_id.cat.codes, df_modeling.state_id))

cols = df_modeling.dtypes.index.tolist() # get all the columns in a list
types = df_modeling.dtypes.values.tolist() # get all the column types in a list
```

```
for i,type in enumerate(types):
    if type.name == 'category':
        df_modeling[cols[i]] = df_modeling[cols[i]].cat.codes # replace all categori
```

In [50]:

```
df_modeling.d = df_modeling['d'].apply(lambda x: x.split('_')[1]).astype(np.int16) #
df_modeling.drop('date',axis=1,inplace=True) # remove the date column
```

In [51]:

```
# Mean Encoding for selected groups of variables

df_modeling['sold_avg_item'] = df_modeling.groupby('item_id')['sold'].transform('mea
df_modeling['sold_avg_state'] = df_modeling.groupby('state_id')['sold'].transform('m
df_modeling['sold_avg_store'] = df_modeling.groupby('store_id')['sold'].transform('m
df_modeling['sold_avg_cat'] = df_modeling.groupby('cat_id')['sold'].transform('mean'
df_modeling['sold_avg_dept'] = df_modeling.groupby('dept_id')['sold'].transform('mea
df_modeling['sold_avg_cat_dept']
                    \= df_modeling.groupby(['cat_id','dept_id'])['sold'].transform('
df_modeling['sold_avg_store_item']
                    \= df_modeling.groupby(['store_id','item_id'])['sold'].transform
df_modeling['sold_avg_cat_item']
                    \= df_modeling.groupby(['cat_id','item_id'])['sold'].transform('
df_modeling['sold_avg_dept_item']
                    \= df_modeling.groupby(['dept_id','item_id'])['sold'].transform(
df_modeling['sold_avg_state_store']
                    \= df_modeling.groupby(['state_id','store_id'])['sold'].transfor
df_modeling['sold_avg_state_store_cat']
                    \= df_modeling.groupby(['state_id','store_id','cat_id'])['sold']
df_modeling['sold_avg_store_cat_dept']
                    \= df_modeling.groupby(['store_id','cat_id','dept_id'])['sold'].
```

6.3 Engineering Lag Features

Calculate each variable with a shift(): Shift index by a desired number of periods to compare the correlation with the other variables

In [52]:

```
lags = [1,2,3,6,12,24,36]
for lag in lags:
    df_modeling['sold_lag_'+str(lag)] = df_modeling.groupby(['id', 'item_id', 'dept_
                                            as_index=False)['sold'].shift(lag).astype(
```

Out[52]:

	id	item_id	dept_id	cat_id	store_id	state_id	d	sold	wm_yr_wk	weekday	...	sold_avg_st
0	14370	1437	3	1	0	0	1	0	11101	2	...	
1	14380	1438	3	1	0	0	1	0	11101	2	...	
2	14390	1439	3	1	0	0	1	0	11101	2	...	
3	14400	1440	3	1	0	0	1	0	11101	2	...	
4	14410	1441	3	1	0	0	1	0	11101	2	...	

5 rows × 40 columns

6.4 Engineering Window Features

In [53]:
```
# Calculate the rolling average (weekly) of sold items

df_modeling['sold_avg_rolling'] = \
df_modeling.groupby(['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id'])[
.transform(lambda x: x.rolling(window=7).mean()).astype(np.float16)
```

Out[53]:

	id	item_id	dept_id	cat_id	store_id	state_id	d	sold	wm_yr_wk	weekday	...	sold_avg_st
0	14370	1437	3	1	0	0	1	0	11101	2	...	
1	14380	1438	3	1	0	0	1	0	11101	2	...	
2	14390	1439	3	1	0	0	1	0	11101	2	...	
3	14400	1440	3	1	0	0	1	0	11101	2	...	
4	14410	1441	3	1	0	0	1	0	11101	2	...	

5 rows × 41 columns

In [54]:
```
# Calculate the expanding average of sold items

df_modeling['sold_avg_expanding'] = \
df_modeling.groupby(['id', 'item_id', 'dept_id', 'cat_id', 'store_id', 'state_id'])[
.transform(lambda x: x.expanding(2).mean()).astype(np.float16)
```

6.5 Finalize the dataframe for modeling

In [55]:
```
# Remove data for 35 first days (max lags was 36 so it would introduce a lot of Null
df_modeling = df_modeling[df_modeling['d']>=36]
```

In [57]:
```
df_modeling.head()
```

Out[57]:

	id	item_id	dept_id	cat_id	store_id	state_id	d	sold	wm_yr_wk	weekday	...	sol
1067150	14370	1437	3	1	0	0	36	0	11106	2	...	
1067151	14380	1438	3	1	0	0	36	0	11106	2	...	
1067152	14390	1439	3	1	0	0	36	0	11106	2	...	
1067153	14400	1440	3	1	0	0	36	0	11106	2	...	
1067154	14410	1441	3	1	0	0	36	0	11106	2	...	

5 rows × 42 columns

In [56]:
```
# Save the dataframe as a binary file on the hard drive
df_modeling.to_pickle('df_modeling.pkl')
```

7 Modeling

Go to contents

Statistical models

- Baseline
- Exponential smoothing
- Holt linear
- SARIMAX

Machine learning models

- LightGBM
- XGBoost

Performance Measure: RMSE

7.1 Statistical models

In [2]:
```
# Read the saved downcasted dataframes

df_sales = pd.read_pickle('df_sales.pkl')
df_calendar = pd.read_pickle('df_calendar.pkl')
df_prices = pd.read_pickle('df_prices.pkl')
```

In [4]:
```
# Split the data for modeling

date_col = [c for c in df_sales.columns if c.startswith('d_')] # all the columns tha

dataset = df_sales.copy()
dataset.id = dataset.id.str.replace('evaluation','validation')
dataset = dataset.set_index(['id'])

# Training dataset: The actual dataset used to train the model (the model sees and l
dataset_train = dataset[date_col[:1913]]  # training dataset includes days 1-1913 (2
dataset_train.columns = df_calendar.date[:1913]

# Validation dataset: The sample of data used to fine-tune the model hyperparameters
dataset_val = dataset[date_col[1913:1941]]  # validation dataset includes days 1914-
dataset_val.columns = df_calendar.date[1913:1941]
dataset_val

# Test or Evaluation dataset: The sample of data used to provide an unbiased evaluat
dataset_test = df_sales[date_col[1941:]] # test dataset includes days 1942-1969 (201
```

7.1.1 Baseline

The first approach is the very simple naive approach. It simply forecasts the next day's sales as the current day's sales as follows:

$$\hat{y}_{t+1} = y_t$$

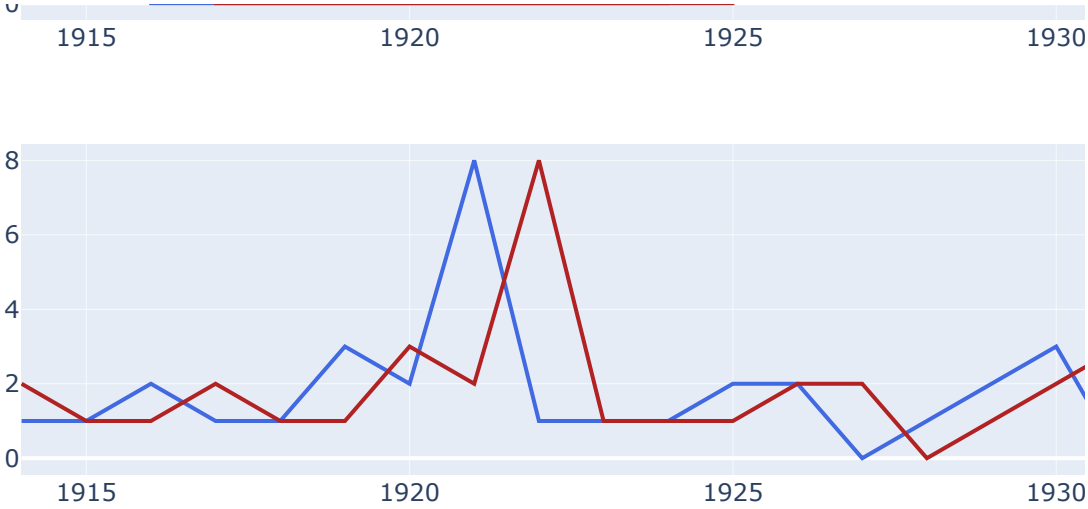
where yt+1 is the predicted value for the next day's sales and yt is today's sales

In [341…
```
pred_naive = []
for i in range(len(dataset_val.columns)):
    if i == 0:
        pred_naive.append(dataset_train[dataset_train.columns[-1]].values)
    else:
```

```
        pred_naive.append(dataset_val[dataset_val.columns[i-1]].values)

pred_naive = np.transpose(np.array([row.tolist() for row in pred_naive]))
```

In [91]:

```
# Plot the predicted vs actual sales for 3 randomly selected items

fig = make_subplots(rows=3, cols=1)

for i in range(3):
    rand_item = randrange(len(dataset_train))

    # Plot the true values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(1914, 1941),
                             y=dataset_val.iloc[rand_item].values,
                             mode='lines',
                             marker=dict(color="royalblue"),
                             showlegend=False,
                             name="Actual value"),
                  row=i+1, col=1)

    # Plot the predicted values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(1914, 1941),
                             y=pred_naive[rand_item],
                             mode='lines',
                             marker=dict(color="firebrick"),
                             showlegend=False,
                             name="Predicted value"),
                  row=i+1, col=1)

fig.update_layout(height=800, width=1000, title_text="Naive approach")

fig.show()
```



In [6]:
```
# RMSE of validation dataset
rmse_naive = mean_squared_error(dataset_val.values, pred_naive, squared=False)
rmse_naive
```

Out[6]:
```
2.6125732980899596
```

In [155…
```
a_file = open("rmse_naive.pkl", "wb")
pickle.dump(rmse_naive, a_file)
a_file.close()
```

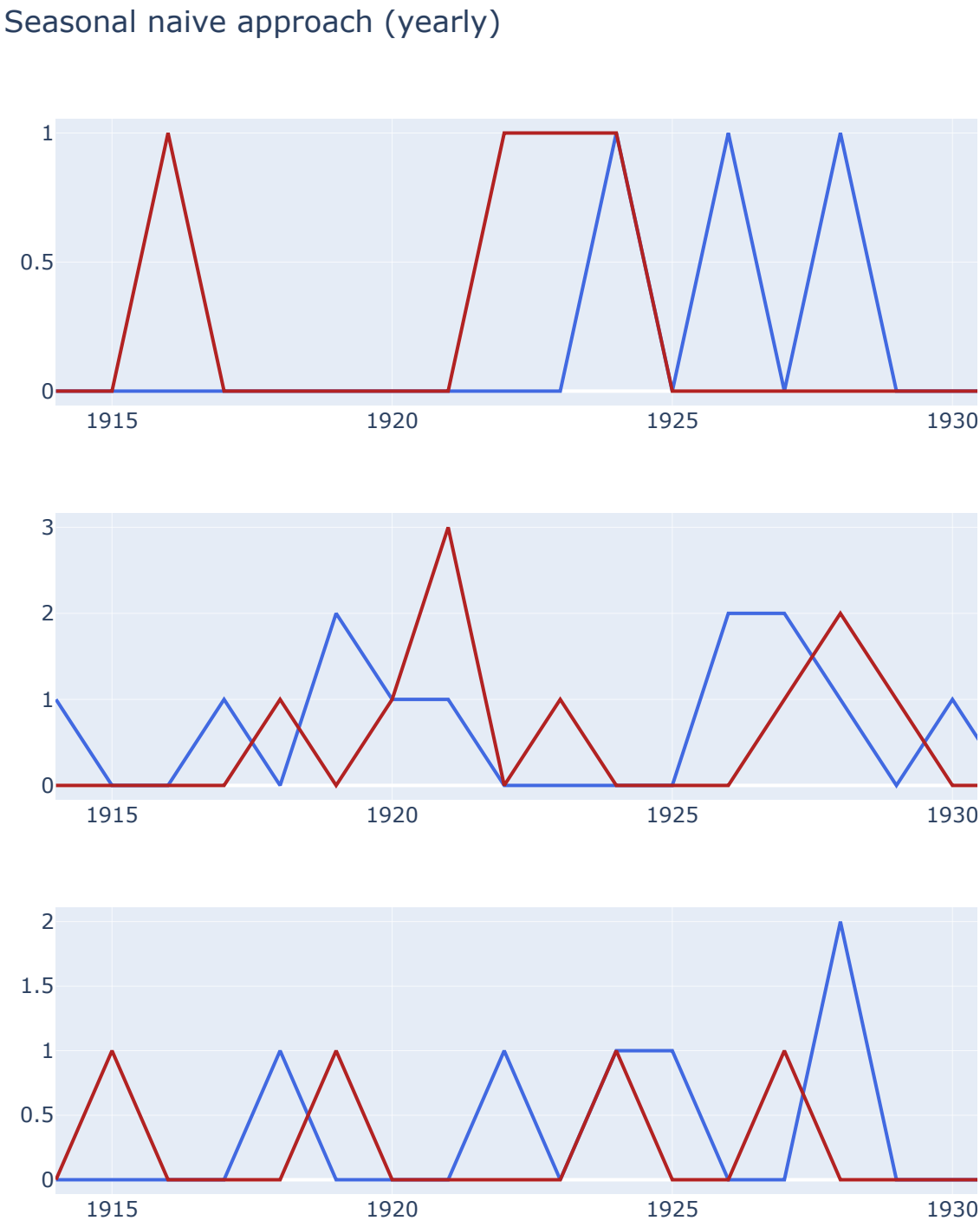
Seasonal naive approach

We will instead use the seasonal naive approach as the benchmark, since it is more realistic (we will not actually have the data for the previous day when making the predictions)

In [378…
```
pred_snaive = []
season = 366 # considering a yearly seasonality
for i in range(len(dataset_val.columns)):
        pred_snaive.append(dataset_train[dataset_train.columns[i-season]].values)

pred_snaive = np.transpose(np.array([row.tolist() for row in pred_snaive]))
```

In [385…
```
# Plot the predicted vs actual sales for 3 randomly selected items

fig = make_subplots(rows=3, cols=1)

for i in range(3):
    rand_item = randrange(len(dataset_train))

    # Plot the true values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(1914, 1941),
                             y=dataset_val.iloc[rand_item].values,
                             mode='lines',
                             marker=dict(color="royalblue"),
                             showlegend=False,
                             name="Actual value"),
```

```
                    row=i+1, col=1)

    # Plot the predicted values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(1914, 1941),
                             y=pred_snaive[rand_item],
                             mode='lines',
                             marker=dict(color="firebrick"),
                             showlegend=False,
                             name="Predicted value"),
                  row=i+1, col=1)

fig.update_layout(height=800, width=1000, title_text="Seasonal naive approach (yearl

fig.show()
```



In [386…

```
# RMSE of validation dataset
rmse_snaive = mean_squared_error(dataset_val.values, pred_snaive, squared=False)
rmse_snaive
```

Out[386…

```
3.2205538983481645
```

In [387…

```
a_file = open("rmse_snaive.pkl", "wb")
pickle.dump(rmse_snaive, a_file)
a_file.close()
```

Evaluation:

The result from the seasonal naive approach can be used as the performance baseline for our forecasting

7.1.2 Exponential smoothing

In exponential smoothing the previous time steps are exponentially weighted and added up to generate the forecast. The weights decay as we move further backwards in time. The model can be summarized as follows:

$$\hat{y}_{t+1} = \alpha \cdot y_t + \alpha \cdot (1-\alpha) \cdot y_{t-1} + \alpha \cdot (1-\alpha)^2 \cdot y_{t-2} + \cdots \quad \hat{y}_{t+1} = \alpha \cdot y_t + (1-\alpha) \cdot \hat{y}_t$$

In the above equations, α is the smoothing parameter. The forecast yt+1 is a weighted average of all the observations in the series y1, … ,yt. The rate at which the weights decay is controlled by the parameter α. This method gives different weightage to different time steps, instead of giving the same weightage to all time steps (like the moving average method). This ensures that recent sales data is given more importance than old sales data while making the forecast.

In [403…

```
sample = 1000 # We are using a sample of 1000 items due to memory constraints

pred_es = []
for row in tqdm(dataset_train.head(sample).values):
    fit = ExponentialSmoothing(row, seasonal_periods=7, initialization_method='estim
                               trend='add', seasonal='add').fit()
    pred_es.append(fit.forecast(len(dataset_val.columns)))
pred_es = np.array(pred_es).reshape((-1, len(dataset_val.columns)))
```

Training was completed in 05:33

In [ ]:

```
# Save the fitted model and the predictions
joblib.dump(fit, 'es_fitted_model.pkl')
joblib.dump(pred_es, 'pred_es.pkl')
```

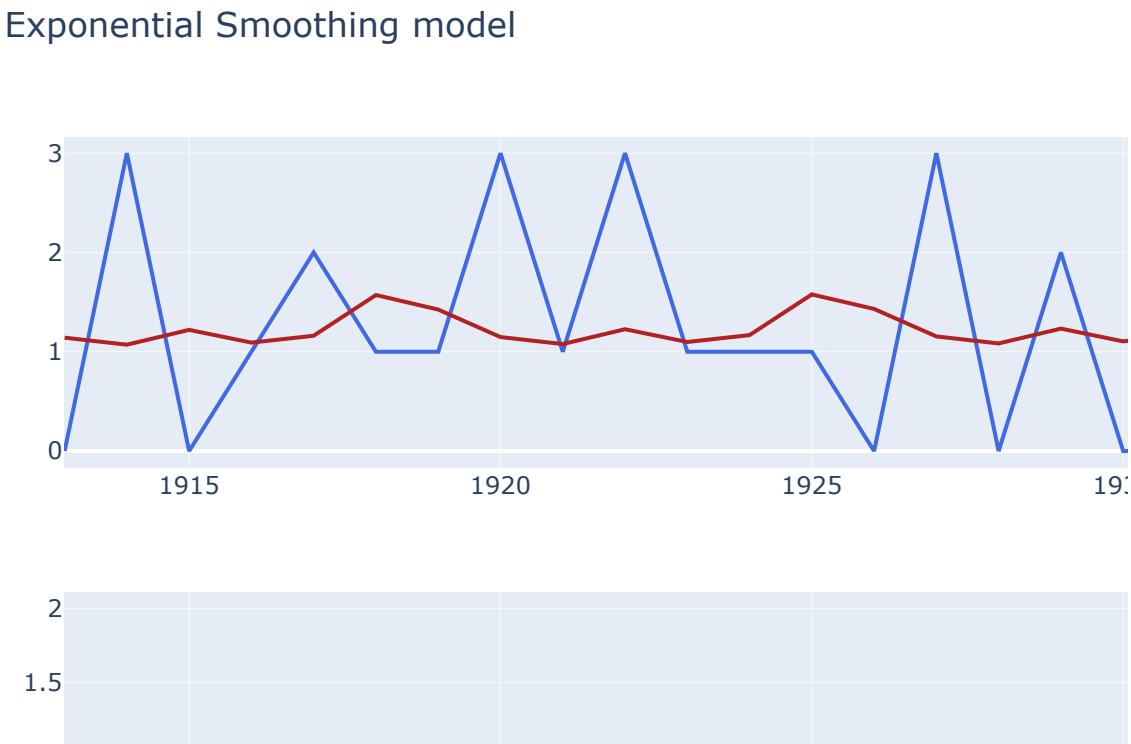
In [ ]:

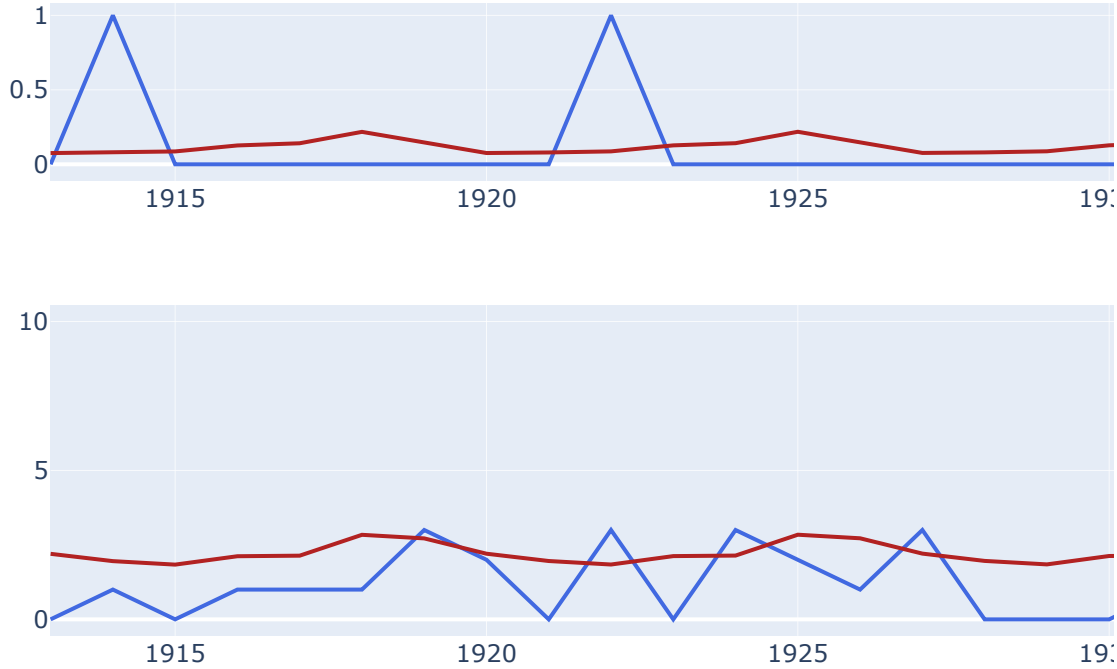
```
# Prepare for submission

# Prepare the validation results
validation = pd.DataFrame(pred_es).reset_index()
# columns F1-F28 corresponding to d_1914 - d_1941
```

```
validation.columns = ['id'] + [f'F{i+1}' for i in range(28)]
# remap the category id to their respective categories
validation.id = validation.id.map(d_id).str.replace('evaluation','validation')

# Prepare the submission file
#submission_es = pd.concat([validation, evaluation]).reset_index(drop=True)
#submission_es.to_csv('submission_es.csv', index=False)
validation.to_csv('submission_es.csv', index=False)
```

In [414…

```
fig = make_subplots(rows=3, cols=1)

for i in range(3):
    rand_item = randrange(sample)

    # Plot the true values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(len(dataset_train.columns),
                                         len(dataset_train.columns)+len(dataset_val.
                             y=dataset_val.iloc[rand_item].values,
                             mode='lines',
                             marker=dict(color="royalblue"),
                             showlegend=False,
                             name="Actual value"),
                  row=i+1, col=1)

    # Plot the predicted values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(len(dataset_train.columns),
                                         len(dataset_train.columns)+len(dataset_val.
                             y=pred_es[rand_item],
                             mode='lines',
                             marker=dict(color="firebrick"),
                             showlegend=False,
                             name="Predicted value"),
                  row=i+1, col=1)


fig.update_layout(height=800, width=1000, title_text="Exponential Smoothing model")
fig.show()
```





In [415…

```
# RMSE of validation dataset
rmse_es = mean_squared_error(dataset_val.head(sample), pred_es, squared=False)
rmse_es
```

Out[415…

```
1.9462948475032351
```

In [156…

```
a_file = open("rmse_es.pkl", "wb")
pickle.dump(rmse_es, a_file)
a_file.close()
```

Evaluation:

Exponential smoothing predicts the mean sales with accuracy, however it generates a rather flattened result due to the low weightage given to older time steps.

7.1.3 Holt Linear

Holt linear attempts to capture the high-level trends in time series data using a linear function. The method can be summarized as follows:

- Forecast equation $\hat{y}_{t+h} = l_t + h \cdot b_t$
- Level equation $l_t = a \cdot y_t + (1 - \alpha) \cdot (l_{t-1} + b_{t-1})$
- Trend equation $b_t = \beta \cdot (l_t - l_{t-1}) + (1 - \beta) \cdot b_{t-1}$

In the above equations, α and β are constants which can be configured. The values lt and bt represent the level and trend values repsectively. The trend value is the slope of the linear

forecast function and the level value is the y-intercept of the linear forecast function. The slope and y-intercept values are continuously updated using the second and third update equations. Finally, the slope and y-intercept values are used to calculate the forecast, yt+h (in equation 1), which is h time steps ahead of the current time step.

In [416…

```
pred_holt = []
for row in tqdm(dataset_train.values): # show a progress bar
    fit = Holt(row).fit(smoothing_level = 0.3, smoothing_slope = 0.01) # create the
    pred_holt.append(fit.forecast(len(dataset_val.columns)))
pred_holt = np.array(pred_holt).reshape((-1, len(dataset_val.columns)))
```

Training was completed in 15:50

In [ ]:

```
# Save the fitted model and the predictions
joblib.dump(fit, 'holt_fitted_model.pkl')
joblib.dump(pred_holt, 'pred_holt.pkl')
```

In [149…

```
# Prepare for submission

# Prepare the validation results
validation = pd.DataFrame(pred_holt).reset_index()
# columns F1-F28 corresponding to d_1914 - d_1941
validation.columns = ['id'] + [f'F{i+1}' for i in range(28)]
# remap the category id to their respective categories
validation.id = validation.id.map(d_id).str.replace('evaluation','validation')

# Prepare the submission file
#submission_holt = pd.concat([validation, evaluation]).reset_index(drop=True)
#submission_holt.to_csv('submission_holt.csv', index=False)
validation.to_csv('submission_holt.csv', index=False)
```

In [424…

```
fig = make_subplots(rows=3, cols=1)

for i in range(3):
    rand_item = randrange(len(dataset_val))

    # Plot the true values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(len(dataset_train.columns),
                                         len(dataset_train.columns)+len(dataset_val.
                             y=dataset_val.iloc[rand_item].values,
                             mode='lines',
                             marker=dict(color="royalblue"),
                             showlegend=False,
                             name="Actual value"),
                  row=i+1, col=1)

    # Plot the predicted values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(len(dataset_train.columns),
                                         len(dataset_train.columns)+len(dataset_val.
                             y=pred_holt[rand_item],
                             mode='lines',
                             marker=dict(color="firebrick"),
                             showlegend=False,
                             name="Predicted value"),
                  row=i+1, col=1)
```

```
fig.update_layout(height=800, width=1000, title_text="Holt Linear model")
fig.show()
```

In [153…
```
# RMSE of validation dataset
rmse_holt = mean_squared_error(dataset_val.values, pred_holt, squared=False)
rmse_holt
```

Out[153…
```
2.3421180796575163
```

In [154…
```
a_file = open("rmse_holt.pkl", "wb")
```

```
pickle.dump(rmse_holt, a_file)
a_file.close()
```

Evaluation:

Holt linear is able to predict high-level trends in the sales but it does not capture the short-term volatility in the daily sales.

7.1.4 SARIMAX

SARIMAX stands for **S**easonal **A**uto**r**egressive **I**ntegrated **M**oving **A**verage E**x**ogenous
ARIMA models aim to describe the correlations in the time series.

We need to first check for stationarity using the Augmented Dickey-Fuller test (aka ADF):

- if the test statistics is more than 5% of the critical value and the p-value is larger than 0.05 -> the moving average is not constant over time and the null hypothesis of the Dickey-Fuller test cannot be rejected
- if the time series is not stationary, we need to transform it into stationary
- if the test statistic is less than 1% of the critical value and the p-value is 0.01, we can say that the time series is stationary with 99% confidence
- if the time series is stationary, then we can apply ARIMA to forecast the future values

In [43]:
```
# Calculate the daily sales (sum of all items sold per day)
daily_sales = pd.DataFrame(df_sales[date_col].sum(axis =0),columns=["sales"])

# Convert the index to a datetime column
base = datetime.datetime(2011,1,29) # set the start date
daily_sales['date'] = [base + datetime.timedelta(days=x) for x in range(len(date_col
daily_sales.set_index('date', drop=True, inplace=True)
daily_sales.sort_index(inplace=True)
```

Out[43]:

	sales
date	
2011-01-29	32631
2011-01-30	31749
2011-01-31	23783
2011-02-01	25412
2011-02-02	19146
...	...
2016-06-15	0
2016-06-16	0
2016-06-17	0
2016-06-18	0
2016-06-19	0

1969 rows × 1 columns

In [69]:

```
# Separate the daily sales by category from the initial sales dataset
df_categories = df_sales.groupby("cat_id")[date_col].sum().T

# Configure the index to include the dates
base = datetime.datetime(2011,1,29) # set the start date
df_categories['date'] = [base + datetime.timedelta(days=x) for x in range(len(date_c
df_categories = df_categories.set_index('date', drop=True).sort_index()
df_categories
```

Out[69]:

cat_id date	FOODS	HOBBIES	HOUSEHOLD
2011-01-29	23178.0	3764.0	5689.0
2011-01-30	22758.0	3357.0	5634.0
2011-01-31	17174.0	2682.0	3927.0
2011-02-01	18878.0	2669.0	3865.0
2011-02-02	14603.0	1814.0	2729.0
...	...	...	...
2016-06-15	0.0	0.0	0.0
2016-06-16	0.0	0.0	0.0
2016-06-17	0.0	0.0	0.0
2016-06-18	0.0	0.0	0.0
2016-06-19	0.0	0.0	0.0

1969 rows × 3 columns

In [85]:

```
# Performing the ADF test on the daily_sales dataset (sum of all items sold per day)

from statsmodels.tsa.stattools import adfuller
adf_sum = adfuller(daily_sales, autolag = 'AIC')
print("1. ADF : ",adf_sum[0])
print("2. P-Value : ", adf_sum[1])
print("3. Num Of Lags : ", adf_sum[2])
print("4. Num Of Observations Used For ADF Regression and Critical Values Calculatio
print("5. Critical Values :")
for key, val in dftest[4].items():
    print("\t",key, ": ", val)
```

```
1. ADF :  -1.8124676284926875
2. P-Value :  0.3742896656883241
3. Num Of Lags :  26
4. Num Of Observations Used For ADF Regression and Critical Values Calculation : 194
2
5. Critical Values :
         1% :  -3.433721763505903
         5% :  -2.8630294391572706
         10% :  -2.567562917840866
```

In [88]:

```
# Performing the ADF test on the three categories time series (sum of items sold per

adf_foods = adfuller(df_categories.FOODS, autolag='AIC')
adf_hobbies = adfuller(df_categories.HOBBIES, autolag='AIC')
```

```
adf_household = adfuller(df_categories.HOUSEHOLD, autolag='AIC')

print("p-value of FOODS time serie is: {}".format(float(adf_foods[1])))
print("p-value of HOBBIES time serie is: {}".format(float(adf_hobbies[1])))
print("p-value of HOUSEHOLD time serie is: {}".format(float(adf_household[1])))
```

```
p-value of FOODS time serie is: 0.9680368207260177
p-value of HOBBIES time serie is: 0.2639990369374694
p-value of HOUSEHOLD time serie is: 0.3742896656883241
```

In [134…

```
pred_sarimax = []
for row in tqdm(dataset_train[dataset_train.columns[-len(dataset_val.columns):]].val
    fit = sm.tsa.statespace.SARIMAX(row, seasonal_order=(0, 1, 1, 7)).fit()
    pred_sarimax.append(fit.forecast(len(dataset_val.columns)))
pred_sarimax = np.array(pred_arima).reshape((-1, len(dataset_val.columns)))
```

Training was completed in 28:33

In [ ]:

```
# Save the fitted model and the predictions
joblib.dump(fit, 'sarimax_fitted_model.pkl')
joblib.dump(pred_sarimax, 'pred_sarimax.pkl')
```

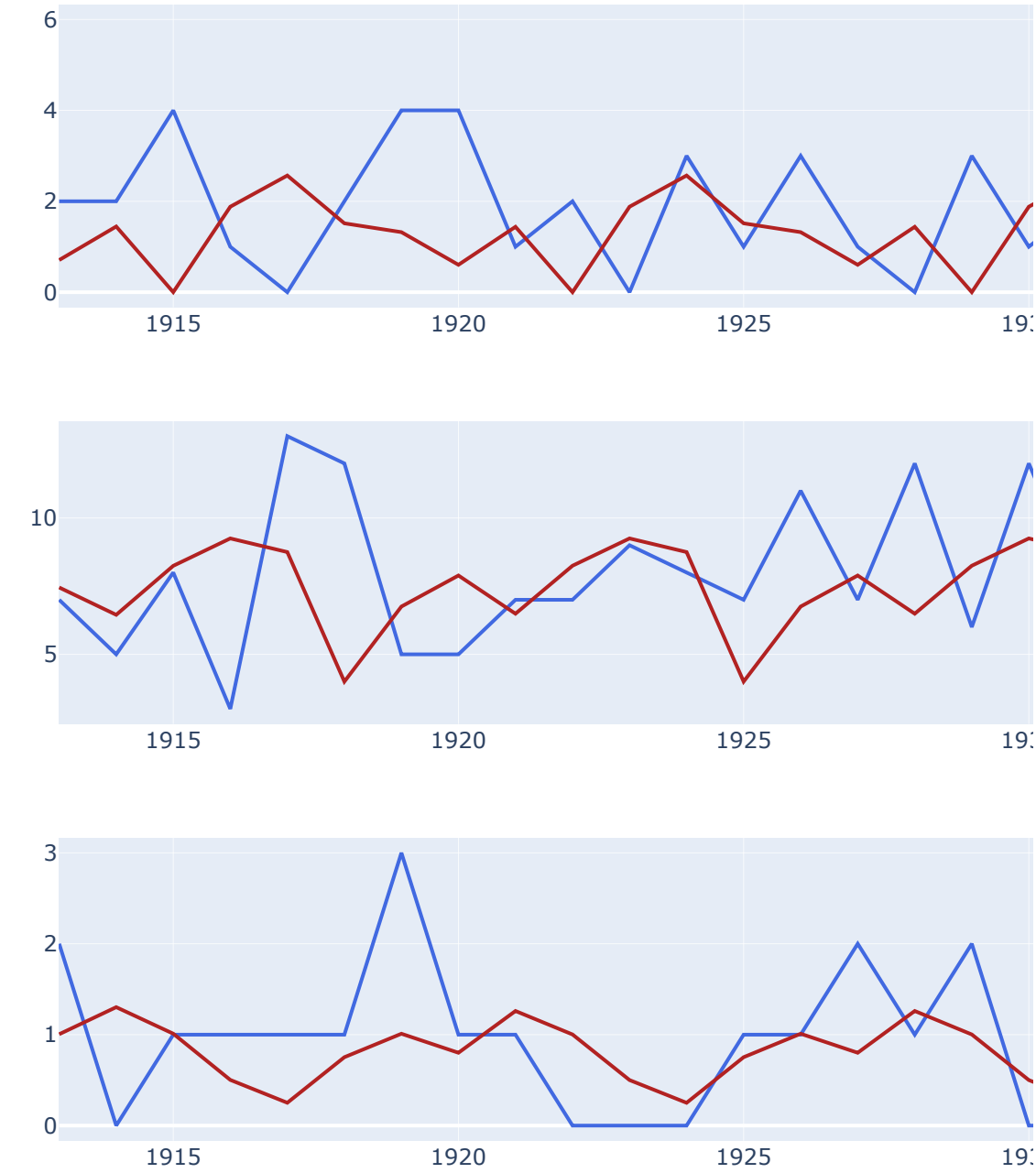
In [161…

```
# Prepare for submission

# Prepare the validation results
validation = pd.DataFrame(pred_sarimax).reset_index()
# columns F1-F28 corresponding to d_1914 - d_1941
validation.columns = ['id'] + [f'F{i+1}' for i in range(28)]
# remap the category id to their respective categories
validation.id = validation.id.map(d_id).str.replace('evaluation','validation')

# Prepare the submission file
#submission_sarimax = pd.concat([validation, evaluation]).reset_index(drop=True)
#submission_sarimax.to_csv('submission_sarimax.csv', index=False)
validation.to_csv('submission_sarimax.csv', index=False)
```

In [142…

```
fig = make_subplots(rows=3, cols=1)

for i in range(3):
    rand_item = randrange(len(dataset_train))

    # Plot the true values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(len(dataset_train.columns),
                                         len(dataset_train.columns)+len(dataset_val.
                             y=dataset_val.iloc[rand_item].values,
                             mode='lines',
                             marker=dict(color="royalblue"),
                             showlegend=False,
                             name="Actual value"),
                  row=i+1, col=1)

    # Plot the predicted values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(len(dataset_train.columns),
                                         len(dataset_train.columns)+len(dataset_val.
                             y=pred_sarimax[rand_item],
                             mode='lines',
                             marker=dict(color="firebrick"),
                             showlegend=False,
```

```
                                name="Predicted value"),
                    row=i+1, col=1)

fig.update_layout(height=800, width=1000, title_text="SARIMAX model")
fig.show()
```



In [156…

```
# RMSE of validation dataset
rmse_sarimax = mean_squared_error(dataset_val.values, pred_sarimax, squared=False)
rmse_sarimax
```

Out[156…

```
2.4794258389613835
```

In [160…

```
# Save the RMSE result locally
a_file = open("rmse_sarimax.pkl", "wb")
pickle.dump(rmse_sarimax, a_file)
a_file.close()
```

Evaluation:

SARIMAX was able to find both low-level and high-level trends and it was able to predict a rather accurate periodic function for each item.

7.2 Machine Learning Models

In [2]:

```
# Prepare the data for modeling
data = pd.read_pickle('df_modeling.pkl') # read the saved dataframe for modeling

# Used for LBGM and XGBoost modeling
y_val_true = data[(data['d']>=1914) & (data['d']<1942)][['id','d','sold']] # id,d,so
y_test_true = data[data['d']>=1942][['id','d','sold']]                     # id,d,so
y_val_preds = y_val_true['sold']   # sold values on days 1914-1941 (to be overwritte
y_test_preds = y_test_true['sold'] # sold values on days 1942-1969 (to be overwritte

# Get the store codes
store_codes = data.store_id.unique().tolist()

# Load the saved d_id and d_store_id dictionaries
a_file = open("d_id.pkl", "rb")
d_id = pickle.load(a_file)
a_file = open("d_store_id.pkl", "rb")
d_store_id = pickle.load(a_file)
```

In [6]:

```
# Prepare the data for plotting the results
plot_y_true = pd.pivot(data, index='id', columns='d', values='sold').reset_index()
plot_y_true.id = plot_y_true.id.map(d_id).str.replace('evaluation','validation')
plot_y_true.set_index(['id'], inplace=True)
plot_y_true.index = plot_y_true.index.astype('str')
plot_y_true = plot_y_true[plot_y_true.index.str.contains('validation')]
plot_y_true = plot_y_true.loc[:, :1941]
```

7.2.2 XGBoost

XGBoost requires that

- the time series dataset is transformed into a supervised learning problem (which we have done at the df_modeling dataframe).
- a specialized technique called walk-forward validation is used for evaluating the model (evaluating the model using k-fold cross validation which randomizes the dataset would not work because the model must always be trained on the past to predict the future).

Formula for computation of predictions:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

Where estimation y_i is the prediction, x_i is a vector of features, f_k(x_i) are the values computed for each tree, and K is the total number of trees.

In [66]:

```
# Modeling per store
for store in store_codes:
    df = data[data['store_id']==store]

    # Split the data
    X_train = df[df['d']<1914].drop('sold',axis=1) # All columns except Sold on days
    y_train = df[df['d']<1914]['sold']             # Sold values on days 0-1913
    X_val = df[(df['d']>=1914) & (df['d']<1942)].drop('sold',axis=1) # All columns e
    y_val = df[(df['d']>=1914) & (df['d']<1942)]['sold']             # Sold values o
    X_test = df[df['d']>=1942].drop('sold',axis=1) # All columns except Sold on days

    # Train and validate the model
    model = XGBRegressor(eta=0.3, # typical final values: 0.01-0.2
                         min_child_weight=1,
                         max_depth=6) # typical values:3-10

    print('------ Prediction for Store: {} ------------'.format(d_store_id[store]))

    # Use the train data on days 0-1913 to fit the model,
    # and the val data on days 1914-1941 to evaluate the model (against RMSE)
    model.fit(X_train, y_train,
              eval_set=[(X_train,y_train), (X_val,y_val)],
              eval_metric='rmse',
              verbose=20, early_stopping_rounds=20)

    y_val_preds[X_val.index] = model.predict(X_val) # save predictions of Sold items
    y_test_preds[X_test.index] = model.predict(X_test) # save predictions of Sold it

    # Save the model for each store
    filename = 'xgb_model_'+str(d_store_id[store])+'.pkl'
    joblib.dump(model, filename)

    del model, X_train, y_train, X_val, y_val # clear the temporary data
    gc.collect()
```

```
------ Prediction for Store: CA_1 ------------
[0]     validation_0-rmse:3.26529       validation_1-rmse:2.96288
[20]    validation_0-rmse:1.90991       validation_1-rmse:1.75834
[40]    validation_0-rmse:1.85899       validation_1-rmse:1.72826
[60]    validation_0-rmse:1.82905       validation_1-rmse:1.72174
[80]    validation_0-rmse:1.80068       validation_1-rmse:1.72448
[82]    validation_0-rmse:1.79834       validation_1-rmse:1.72418
------ Prediction for Store: CA_2 ------------
[0]     validation_0-rmse:2.27429       validation_1-rmse:2.68300
[20]    validation_0-rmse:1.46803       validation_1-rmse:1.62867
[40]    validation_0-rmse:1.43197       validation_1-rmse:1.59856
[60]    validation_0-rmse:1.41359       validation_1-rmse:1.58581
[80]    validation_0-rmse:1.39810       validation_1-rmse:1.58724
[81]    validation_0-rmse:1.39802       validation_1-rmse:1.58707
------ Prediction for Store: CA_3 ------------
```

```
[0]    validation_0-rmse:4.92387       validation_1-rmse:3.83552
[20]   validation_0-rmse:2.62848       validation_1-rmse:2.06531
[40]   validation_0-rmse:2.55973       validation_1-rmse:2.02106
[60]   validation_0-rmse:2.52028       validation_1-rmse:2.01002
[80]   validation_0-rmse:2.47475       validation_1-rmse:2.01009
[85]   validation_0-rmse:2.46546       validation_1-rmse:2.01161
------ Prediction for Store: CA_4 ------------
[0]    validation_0-rmse:1.66048       validation_1-rmse:1.65783
[20]   validation_0-rmse:1.10843       validation_1-rmse:1.16669
[40]   validation_0-rmse:1.08345       validation_1-rmse:1.14974
[60]   validation_0-rmse:1.06942       validation_1-rmse:1.14766
[67]   validation_0-rmse:1.06621       validation_1-rmse:1.16393
------ Prediction for Store: TX_1 ------------
[0]    validation_0-rmse:2.68378       validation_1-rmse:2.47997
[20]   validation_0-rmse:1.59627       validation_1-rmse:1.43645
[40]   validation_0-rmse:1.55011       validation_1-rmse:1.41740
[60]   validation_0-rmse:1.51988       validation_1-rmse:1.41457
[80]   validation_0-rmse:1.49672       validation_1-rmse:1.40957
[99]   validation_0-rmse:1.47303       validation_1-rmse:1.40396
------ Prediction for Store: TX_2 ------------
[0]    validation_0-rmse:3.49988       validation_1-rmse:2.89835
[20]   validation_0-rmse:1.93192       validation_1-rmse:1.57926
[40]   validation_0-rmse:1.87277       validation_1-rmse:1.55713
[60]   validation_0-rmse:1.82711       validation_1-rmse:1.57162
[68]   validation_0-rmse:1.81739       validation_1-rmse:1.58585
------ Prediction for Store: TX_3 ------------
[0]    validation_0-rmse:2.98410       validation_1-rmse:2.89467
[20]   validation_0-rmse:1.63902       validation_1-rmse:1.65006
[40]   validation_0-rmse:1.59292       validation_1-rmse:1.63718
[60]   validation_0-rmse:1.55265       validation_1-rmse:1.65469
[65]   validation_0-rmse:1.54679       validation_1-rmse:1.65390
------ Prediction for Store: WI_1 ------------
[0]    validation_0-rmse:1.98548       validation_1-rmse:2.15779
[20]   validation_0-rmse:1.24339       validation_1-rmse:1.36534
[40]   validation_0-rmse:1.21272       validation_1-rmse:1.34662
[60]   validation_0-rmse:1.19704       validation_1-rmse:1.34048
[80]   validation_0-rmse:1.18440       validation_1-rmse:1.34178
[87]   validation_0-rmse:1.18094       validation_1-rmse:1.34505
------ Prediction for Store: WI_2 ------------
[0]    validation_0-rmse:3.17313       validation_1-rmse:4.04802
[20]   validation_0-rmse:2.01178       validation_1-rmse:2.44065
[40]   validation_0-rmse:1.94867       validation_1-rmse:2.42356
[60]   validation_0-rmse:1.90267       validation_1-rmse:2.42569
[63]   validation_0-rmse:1.89588       validation_1-rmse:2.42962
------ Prediction for Store: WI_3 ------------
[0]    validation_0-rmse:3.20102       validation_1-rmse:3.04456
[20]   validation_0-rmse:1.78314       validation_1-rmse:1.73287
[40]   validation_0-rmse:1.72063       validation_1-rmse:1.69774
[60]   validation_0-rmse:1.68347       validation_1-rmse:1.70533
[64]   validation_0-rmse:1.67733       validation_1-rmse:1.70698
```

In [70]:

```
# RMSE of validation dataset
pred_xgb = y_val_preds
rmse_xgb = mean_squared_error(y_val_true['sold'].values, pred_xgb, squared=False)
rmse_xgb
```

Out[70]:

```
1.6837653
```

In [11]:

```
a_file = open("rmse_xgb.pkl", "wb")
pickle.dump(rmse_xgb, a_file)
a_file.close()
```

In [74]:

```
## Prepare the submission

# Prepare the validation results
validation = y_val_true.copy()
validation['sold'] = y_val_preds
# re-pivot the dataframe from long to wide format
validation = pd.pivot(validation, index='id', columns='d', values='sold').reset_inde
# rename the columns to have F1-F28 for each day (corresponding to d_1914 - d_1941)
validation.columns = ['id'] + ['F' + str(i + 1) for i in range(28)]
# remap the category id to their respective categories
validation.id = validation.id.map(d_id).str.replace('evaluation','validation')

# Prepare the evaluation results
evaluation = y_test_true.copy()
evaluation['sold'] = y_test_preds
evaluation = helper[['id','d','sold']]
# re-pivot the dataframe from long to wide format
evaluation = pd.pivot(evaluation, index='id', columns='d', values='sold').reset_inde
# rename the columns to have F1-F28 for each day (corresponding to d_1942 - d_1969)
evaluation.columns=['id'] + ['F' + str(i + 1) for i in range(28)]
# remap the category id to their respective categories
evaluation.id = evaluation.id.map(d_id)

# Prepare the submission
# concatenate the validation and evaluation results
submission_xgb = pd.concat([validation,evaluation]).reset_index(drop=True)
submission_xgb.to_csv('submission_xgb.csv',index=False)
```

In [254…

```
# Prepare the data for plotting

plot_y_val = submission_xgb.set_index(['id'])
plot_y_val.columns = [i for i in range(1914, 1942)]
plot_y_val.index = plot_y_val.index.astype('str')
plot_y_val = plot_y_val[plot_y_val.index.str.contains('validation')]
```

In [256…

```
# Plot the performance of the model for 3 random items

fig = make_subplots(rows=3, cols=1)

for i in range(3):
    rand_item = randrange(len(plot_y_true))

    # Plot the true values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(1914, 1942),
                             y=plot_y_true.iloc[rand_item, 1914-36:].values,
                             mode='lines',
                             marker=dict(color="seagreen"),
                             showlegend=False,
                             name="Actual value"),
                  row=i+1, col=1)

    # Plot the predicted values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(1914, 1942),
                             y=plot_y_val.iloc[rand_item].values,
                             mode='lines',
                             marker=dict(color="darkorange"),
                             showlegend=False,
                             name="Predicted value"),
                  row=i+1, col=1)
```

```
fig.update_layout(height=800, width=1000, title_text="XGBoost")
fig.show()
```

7.2.1 LightGBM

LightGBM is short for Light Gradient Boosted Machine, a library developed at Microsoft that provides an efficient implementation of the gradient boosting algorithm.

The primary benefit of the LightGBM is the changes to the training algorithm that make the process dramatically faster, and in many cases, result in a more effective model.

In [33]:

```
# Modeling per store
for store in store_codes:
    df = data[data['store_id']==store]

    # Split the data
    X_train = df[df['d']<1914].drop('sold',axis=1) # All columns except Sold on days
    y_train = df[df['d']<1914]['sold']             # Sold values on days 0-1913
    X_val = df[(df['d']>=1914) & (df['d']<1942)].drop('sold',axis=1) # All columns e
    y_val = df[(df['d']>=1914) & (df['d']<1942)]['sold']             # Sold values o
    X_test = df[df['d']>=1942].drop('sold',axis=1) # All columns except Sold on days

    # Train and validate the model
    model = lgbm.LGBMRegressor(n_estimators=1000,
                           learning_rate=0.3,
                           subsample=0.8,
                           colsample_bytree=0.8,
                           max_depth=8,
                           num_leaves=50,
                           min_child_weight=300)

    print('------ Prediction for Store: {} ------------'.format(d_store_id[store]))

    # Use the train data on days 0-1913 to fit the model,
    # and the val data on days 1914-1941 to evaluate the model (against RMSE)
    model.fit(X_train, y_train,
              eval_set=[(X_train,y_train), (X_val,y_val)],
              eval_metric='rmse',
              verbose=20, early_stopping_rounds=20)

    y_val_preds[X_val.index] = model.predict(X_val) # save predictions of Sold items
    y_test_preds[X_test.index] = model.predict(X_test) # save predictions of Sold it

    # Save the model for each store
    filename = 'lgbm_model_'+str(d_store_id[store])+'.pkl'
    joblib.dump(model, filename)

    del model, X_train, y_train, X_val, y_val # clear the temporary data
    gc.collect()
```

```
------ Prediction for Store: CA_1 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.88305        training's l2: 3.54586  valid_1's rmse: 1.75
145     valid_1's l2: 3.06758
[40]    training's rmse: 1.77639        training's l2: 3.15555  valid_1's rmse: 1.66
873     valid_1's l2: 2.78465
[60]    training's rmse: 1.73524        training's l2: 3.01106  valid_1's rmse: 1.64
071     valid_1's l2: 2.69193
[80]    training's rmse: 1.7107 training's l2: 2.92649  valid_1's rmse: 1.6259  vali
d_1's l2: 2.64356
[100]   training's rmse: 1.69597        training's l2: 2.87632  valid_1's rmse: 1.61
956     valid_1's l2: 2.62298
[120]   training's rmse: 1.68653        training's l2: 2.8444   valid_1's rmse: 1.61
682     valid_1's l2: 2.61412
[140]   training's rmse: 1.67776        training's l2: 2.81487  valid_1's rmse: 1.61
561     valid_1's l2: 2.6102
[160]   training's rmse: 1.67004        training's l2: 2.78903  valid_1's rmse: 1.61
507     valid_1's l2: 2.60845
[180]   training's rmse: 1.66723        training's l2: 2.77967  valid_1's rmse: 1.61
399     valid_1's l2: 2.60497
[200]   training's rmse: 1.66098        training's l2: 2.75884  valid_1's rmse: 1.61
328     valid_1's l2: 2.60266
[220]   training's rmse: 1.65667        training's l2: 2.74456  valid_1's rmse: 1.61
```

```
436     valid_1's l2: 2.60615
Early stopping, best iteration is:
[200]   training's rmse: 1.66098        training's l2: 2.75884  valid_1's rmse: 1.61
328     valid_1's l2: 2.60266
------ Prediction for Store: CA_2 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.4159 training's l2: 2.00476  valid_1's rmse: 1.61053 vali
d_1's l2: 2.59381
[40]    training's rmse: 1.36546        training's l2: 1.86447  valid_1's rmse: 1.56
261     valid_1's l2: 2.44174
[60]    training's rmse: 1.34342        training's l2: 1.80478  valid_1's rmse: 1.54
485     valid_1's l2: 2.38657
[80]    training's rmse: 1.33211        training's l2: 1.77451  valid_1's rmse: 1.53
988     valid_1's l2: 2.37122
[100]   training's rmse: 1.32315        training's l2: 1.75071  valid_1's rmse: 1.53
567     valid_1's l2: 2.35828
[120]   training's rmse: 1.31593        training's l2: 1.73168  valid_1's rmse: 1.53
341     valid_1's l2: 2.35135
[140]   training's rmse: 1.31108        training's l2: 1.71893  valid_1's rmse: 1.53
626     valid_1's l2: 2.3601
Early stopping, best iteration is:
[126]   training's rmse: 1.31488        training's l2: 1.7289   valid_1's rmse: 1.53
297     valid_1's l2: 2.35
------ Prediction for Store: CA_3 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 2.52712        training's l2: 6.38631  valid_1's rmse: 2.01
241     valid_1's l2: 4.04979
[40]    training's rmse: 2.42351        training's l2: 5.87341  valid_1's rmse: 1.93
862     valid_1's l2: 3.75824
[60]    training's rmse: 2.3671 training's l2: 5.60315  valid_1's rmse: 1.9058  vali
d_1's l2: 3.63206
[80]    training's rmse: 2.33284        training's l2: 5.44214  valid_1's rmse: 1.88
975     valid_1's l2: 3.57114
[100]   training's rmse: 2.31296        training's l2: 5.34977  valid_1's rmse: 1.88
222     valid_1's l2: 3.54276
[120]   training's rmse: 2.2996 training's l2: 5.28817  valid_1's rmse: 1.88058 vali
d_1's l2: 3.53658
[140]   training's rmse: 2.29154        training's l2: 5.25116  valid_1's rmse: 1.87
972     valid_1's l2: 3.53336
Early stopping, best iteration is:
[126]   training's rmse: 2.2972 training's l2: 5.27712  valid_1's rmse: 1.87875 vali
d_1's l2: 3.52971
------ Prediction for Store: CA_4 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.09535        training's l2: 1.19978  valid_1's rmse: 1.16
925     valid_1's l2: 1.36713
[40]    training's rmse: 1.06611        training's l2: 1.1366   valid_1's rmse: 1.14
281     valid_1's l2: 1.30601
[60]    training's rmse: 1.05402        training's l2: 1.11097  valid_1's rmse: 1.13
615     valid_1's l2: 1.29083
[80]    training's rmse: 1.04623        training's l2: 1.09461  valid_1's rmse: 1.13
346     valid_1's l2: 1.28473
[100]   training's rmse: 1.04292        training's l2: 1.08769  valid_1's rmse: 1.13
199     valid_1's l2: 1.28141
[120]   training's rmse: 1.03968        training's l2: 1.08093  valid_1's rmse: 1.13
177     valid_1's l2: 1.28091
Early stopping, best iteration is:
[111]   training's rmse: 1.04102        training's l2: 1.08372  valid_1's rmse: 1.13
133     valid_1's l2: 1.27991
------ Prediction for Store: TX_1 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.53148        training's l2: 2.34542  valid_1's rmse: 1.32
274     valid_1's l2: 1.74965
[40]    training's rmse: 1.48952        training's l2: 2.21866  valid_1's rmse: 1.29
```

```
573     valid_1's l2: 1.67891
[60]    training's rmse: 1.46345        training's l2: 2.14167  valid_1's rmse: 1.28
315     valid_1's l2: 1.64646
[80]    training's rmse: 1.44872        training's l2: 2.0988   valid_1's rmse: 1.27
805     valid_1's l2: 1.63342
[100]   training's rmse: 1.43891        training's l2: 2.07045  valid_1's rmse: 1.27
592     valid_1's l2: 1.62796
Early stopping, best iteration is:
[97]    training's rmse: 1.4399 training's l2: 2.07331  valid_1's rmse: 1.27543 vali
d_1's l2: 1.62672
------ Prediction for Store: TX_2 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.81458        training's l2: 3.2927   valid_1's rmse: 1.48
703     valid_1's l2: 2.21127
[40]    training's rmse: 1.7589 training's l2: 3.09373  valid_1's rmse: 1.46016 vali
d_1's l2: 2.13208
[60]    training's rmse: 1.72454        training's l2: 2.97405  valid_1's rmse: 1.44
657     valid_1's l2: 2.09255
[80]    training's rmse: 1.70304        training's l2: 2.90034  valid_1's rmse: 1.43
942     valid_1's l2: 2.07193
[100]   training's rmse: 1.68826        training's l2: 2.85024  valid_1's rmse: 1.43
587     valid_1's l2: 2.06173
[120]   training's rmse: 1.67579        training's l2: 2.80829  valid_1's rmse: 1.43
5       valid_1's l2: 2.05921
Early stopping, best iteration is:
[117]   training's rmse: 1.67712        training's l2: 2.81274  valid_1's rmse: 1.43
486     valid_1's l2: 2.05882
------ Prediction for Store: TX_3 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.54831        training's l2: 2.39728  valid_1's rmse: 1.54
622     valid_1's l2: 2.3908
[40]    training's rmse: 1.49998        training's l2: 2.24993  valid_1's rmse: 1.51
64      valid_1's l2: 2.29947
[60]    training's rmse: 1.47188        training's l2: 2.16642  valid_1's rmse: 1.50
564     valid_1's l2: 2.26694
[80]    training's rmse: 1.45626        training's l2: 2.12071  valid_1's rmse: 1.49
85      valid_1's l2: 2.24551
[100]   training's rmse: 1.44536        training's l2: 2.08907  valid_1's rmse: 1.49
097     valid_1's l2: 2.223
[120]   training's rmse: 1.43791        training's l2: 2.06758  valid_1's rmse: 1.48
92      valid_1's l2: 2.21773
[140]   training's rmse: 1.42903        training's l2: 2.04211  valid_1's rmse: 1.48
584     valid_1's l2: 2.20773
[160]   training's rmse: 1.42262        training's l2: 2.02383  valid_1's rmse: 1.48
403     valid_1's l2: 2.20235
Early stopping, best iteration is:
[159]   training's rmse: 1.42273        training's l2: 2.02417  valid_1's rmse: 1.48
371     valid_1's l2: 2.20141
------ Prediction for Store: WI_1 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.19801        training's l2: 1.43523  valid_1's rmse: 1.32
517     valid_1's l2: 1.75608
[40]    training's rmse: 1.16295        training's l2: 1.35244  valid_1's rmse: 1.30
11      valid_1's l2: 1.69285
[60]    training's rmse: 1.14724        training's l2: 1.31615  valid_1's rmse: 1.30
13      valid_1's l2: 1.69339
Early stopping, best iteration is:
[57]    training's rmse: 1.14887        training's l2: 1.31991  valid_1's rmse: 1.29
284     valid_1's l2: 1.67143
------ Prediction for Store: WI_2 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.84173        training's l2: 3.39196  valid_1's rmse: 2.19
842     valid_1's l2: 4.83306
[40]    training's rmse: 1.76615        training's l2: 3.11927  valid_1's rmse: 2.14
```

```
775     valid_1's l2: 4.61283
[60]    training's rmse: 1.72826        training's l2: 2.9869   valid_1's rmse: 2.12
753     valid_1's l2: 4.52638
[80]    training's rmse: 1.70448        training's l2: 2.90525  valid_1's rmse: 2.12
209     valid_1's l2: 4.50326
[100]   training's rmse: 1.69035        training's l2: 2.85727  valid_1's rmse: 2.11
471     valid_1's l2: 4.47199
[120]   training's rmse: 1.6742 training's l2: 2.80294  valid_1's rmse: 2.11798 vali
d_1's l2: 4.48583
Early stopping, best iteration is:
[101]   training's rmse: 1.68993        training's l2: 2.85586  valid_1's rmse: 2.11
41      valid_1's l2: 4.46941
------ Prediction for Store: WI_3 ------------
Training until validation scores don't improve for 20 rounds
[20]    training's rmse: 1.66492        training's l2: 2.77197  valid_1's rmse: 1.61
517     valid_1's l2: 2.60877
[40]    training's rmse: 1.59876        training's l2: 2.55604  valid_1's rmse: 1.57
31      valid_1's l2: 2.47464
[60]    training's rmse: 1.56608        training's l2: 2.45261  valid_1's rmse: 1.56
008     valid_1's l2: 2.43384
[80]    training's rmse: 1.54571        training's l2: 2.38923  valid_1's rmse: 1.55
151     valid_1's l2: 2.4072
[100]   training's rmse: 1.53243        training's l2: 2.34833  valid_1's rmse: 1.54
723     valid_1's l2: 2.39392
[120]   training's rmse: 1.52304        training's l2: 2.31965  valid_1's rmse: 1.54
709     valid_1's l2: 2.39347
Early stopping, best iteration is:
[112]   training's rmse: 1.52604        training's l2: 2.32881  valid_1's rmse: 1.54
468     valid_1's l2: 2.38602
```

In [41]:

```
# RMSE of validation dataset
pred_lgbm = y_val_preds
rmse_lgbm = mean_squared_error(y_val_true['sold'].values, pred_lgbm, squared=False)
rmse_lgbm
```

Out[41]:

```
1.5548662237564475
```

In [8]:

```
a_file = open("rmse_lgbm.pkl", "wb")
pickle.dump(rmse_lgbm, a_file)
a_file.close()
```

In [64]:

```
## Prepare the submission

# Prepare the validation results
validation = y_val_true.copy()
validation['sold'] = y_val_preds
# re-pivot the dataframe from long to wide format
validation = pd.pivot(validation, index='id', columns='d', values='sold').reset_inde
# rename the columns to have F1-F28 for each day
validation.columns=['id'] + ['F' + str(i + 1) for i in range(28)]
# remap the category id to their respective categories
validation.id = validation.id.map(d_id).str.replace('evaluation','validation')

# Prepare the evaluation results
evaluation = y_test_true.copy()
evaluation['sold'] = y_test_preds
evaluation = helper[['id','d','sold']]
# re-pivot the dataframe from long to wide format
evaluation = pd.pivot(evaluation, index='id', columns='d', values='sold').reset_inde
# rename the columns to have F1-F28 for each day
evaluation.columns=['id'] + ['F' + str(i + 1) for i in range(28)]
```

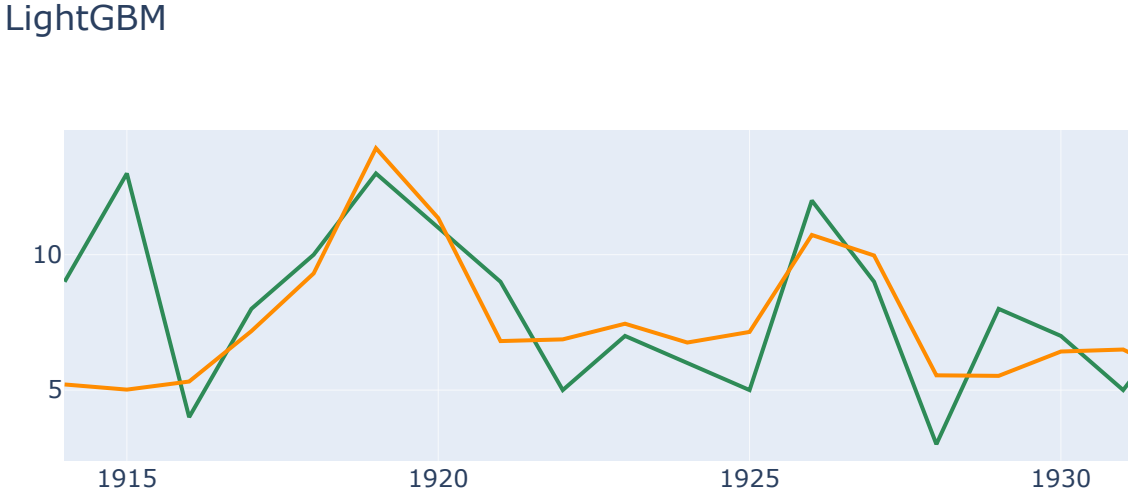
```
# remap the category id to their respective categories
evaluation.id = evaluation.id.map(d_id)

# Prepare the submission
# concatenate the validation and evaluation results
submission_lgbm = pd.concat([validation,evaluation]).reset_index(drop=True)
submission_lgbm.to_csv('submission_lgbm.csv',index=False)
```

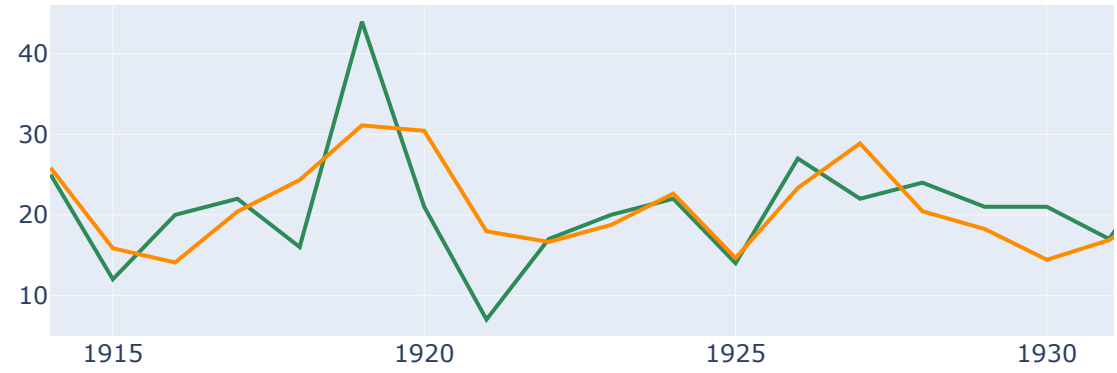
In [252…

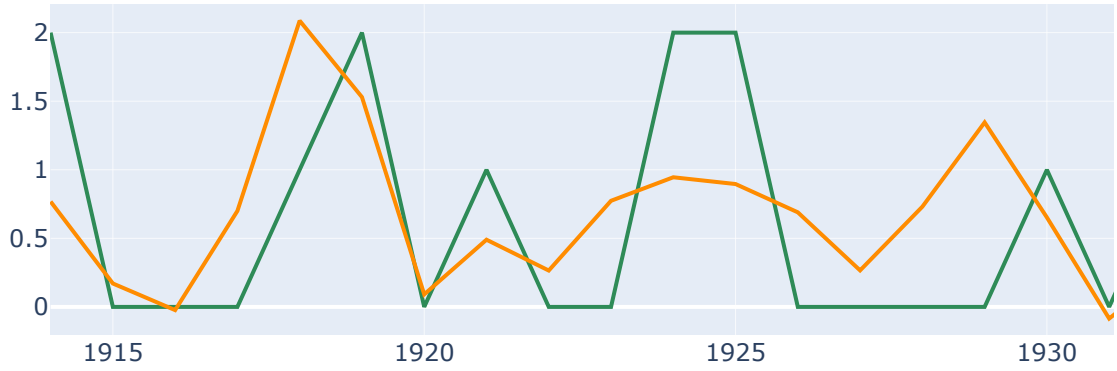
```
# Prepare the data for plotting

plot_y_val = submission_lgbm.set_index(['id'])
plot_y_val.columns = [i for i in range(1914, 1942)]
plot_y_val.index = plot_y_val.index.astype('str')
plot_y_val = plot_y_val[plot_y_val.index.str.contains('validation')]
```

In [253…

```
# Plot the performance of the model for 3 random items

fig = make_subplots(rows=3, cols=1)

for i in range(3):
    rand_item = randrange(len(plot_y_true))

    # Plot the true values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(1914, 1942),
                             y=plot_y_true.iloc[rand_item, 1914-36:].values,
                             mode='lines',
                             marker=dict(color="seagreen"),
                             showlegend=False,
                             name="Actual value"),
                  row=i+1, col=1)

    # Plot the predicted values for days 1914-1941
    fig.add_trace(go.Scatter(x=np.arange(1914, 1942),
                             y=plot_y_val.iloc[rand_item].values,
                             mode='lines',
                             marker=dict(color="darkorange"),
                             showlegend=False,
                             name="Predicted value"),
                  row=i+1, col=1)


fig.update_layout(height=800, width=1000, title_text="LightGBM")
fig.show()
```







8 Model selection

Go to contents

Metric for model selection:

Root Mean Square Error (RMSE) is the standard deviation of the residuals (prediction errors). Residuals are a measure of how far from the regression line data points are; RMSE is a measure of how spread out these residuals are. In other words, it tells you how concentrated the data is around the line of best fit. Root mean square error is commonly used in climatology, forecasting, and regression analysis to verify experimental results.

In [3]:

```
# Load all RMSE

a_file = open("rmse_snaive.pkl", "rb")
rmse_snaive = pickle.load(a_file)
a_file = open("rmse_es.pkl", "rb")
rmse_es = pickle.load(a_file)
a_file = open("rmse_holt.pkl", "rb")
rmse_holt = pickle.load(a_file)
a_file = open("rmse_sarimax.pkl", "rb")
rmse_sarimax = pickle.load(a_file)
a_file = open("rmse_lgbm.pkl", "rb")
rmse_lgbm = pickle.load(a_file)
a_file = open("rmse_xgb.pkl", "rb")
rmse_xgb = pickle.load(a_file)
```

In [5]:

```
# Create a dataframe with all RMSE

rmse_all = [rmse_snaive, rmse_es, rmse_holt, rmse_sarimax, rmse_xgb, rmse_lgbm]
methods = ["Baseline approach", "Exponential Smoothing", "Holt Linear", "SARIMAX", '
df_rmse = pd.DataFrame(np.transpose([rmse_all, methods]))
df_rmse.columns = ["RMSE", "Model"]
df_rmse = df_rmse.astype({'RMSE': 'float'})
```

In [6]:

```
# Plot the RMSE of each model
fig = px.bar(df_rmse,
             y="RMSE",
             x="Model",
             color="Model",
             color_discrete_sequence= px.colors.qualitative.Set2,
             title="Root Mean Square Error of each model")

fig.update_layout(height = 400, width = 800)
fig.show()
```