

ADS1002: Rossmann Sales Forecasting

Zhen Xue, Gue (33521352)

Chun Kei, Chow (33520771)

Sunaina, Rayaprol (33944091)

Hui Shawn, Lim (34017259)

Table of Contents

Table of Contents	1
Executive Summary	3
Problem Statement.....	3
Objective.....	3
Findings	3
Introduction.....	4
Background.....	4
Data Overview	4
Data Preprocessing.....	5
Filtering and Merging	5
Renaming Variables and Converting Datatypes	6
Imputation.....	6
Encoding Categorical Data	9
Exploratory Data Analysis (EDA)	10
Initial EDA	10
Time Series Analysis	10
Multivariate Linear Regression (MLR) model.....	14
<i>Building MLR</i>	14
<i>Analysis of Correlation</i>	15
<i>Simple modelling</i>	16
<i>Initial Feature Selection</i>	16
<i>Analysis of Coefficients</i>	17
<i>Normalisation</i>	17
<i>Regularisation</i>	18
<i>Multicollinearity</i>	18
<i>MLR model Result</i>	19
Predictive Modelling.....	20
Single Regression Tree	20
Ensemble Learning	22
<i>Model Evaluation</i>	22
<i>Random Forests Regression</i>	22
<i>Bagging (bootstrap aggregation)</i>	24

<i>Boosting</i>	25
<i>Adaptive Boosting (AdaBoost)</i>	26
<i>Gradient Boosting</i>	27
<i>Extreme Gradient Boosting (XGBoost)</i>	27
<i>Stacking</i>	29
Conclusion	31
References	32
Appendix A	33
Appendix B	34
Appendix C	36

Executive Summary

Problem Statement

Retail businesses face challenges in accurately forecasting future sales due to the complex interaction of numerous factors such as promotions, holidays, store locations, and customer behaviour. Rossmann is one of Europe's largest drugstore chains. With over 4,000 stores, Rossmann requires accurate sales forecasting to optimize inventory, staffing, and operations. For a large chain like Rossmann, developing an accurate sales prediction model is critical to optimize operations, manage inventory, and improve customer satisfaction. Without precise forecasting, stores risk either overstocking, which leads to wasted resources, or understocking, which could result in lost revenue and unsatisfied customers. With good models, Rossmann can streamline inventory management, allocate staff efficiently, and ensure that resources are available to meet customer demand.

Objective

Using historical sales data, the goal of this project is to develop a machine learning model capable of predicting daily sales of Rossmann drugstores four weeks in advance. Employing various data science techniques, we aim to accurately forecast Rossmann sales four weeks in advance. In addition, we want to identify the key factors affecting Rossmann drugstore sales.

Findings

We analysed historical sales data of the top 20% of stores in terms of sales. After data preprocessing, we used time-series visualisations, correlation scores and multivariate linear regression on our prepared data to identify our key features affecting *Sales*. These features were found to be: *Open*, *Promo*, and *Customers*. Using these selected features, we created several models in attempt to accurately forecast sales.

Predictive modelling utilises decision tree-based methods. Initially, single decision tree regression is chosen to make predictions. However, due to overfitting issues, we have performed ensemble learning to reduce the model variance and bias. We had tried random forest regression, bagging (bootstrap aggregation), boosting and stacking. Among these, StackingRegressor is the most robust model with the smallest error and difference of training and testing R^2 score.

This model results in the most accurate 4 weeks sales predictions but is unable to predict the maximum sales. The main reason is because peak sales are actually noises with unusually high sales that can be ignored. Despite that, the chosen predictive StackingRegressor model performed brilliantly and had succeeded in our Rossmann drug store sales forecasting project.

Introduction

Background

Rossmann is one of the largest drug store chains in Europe, with a significant presence in Germany and other countries. Founded in 1972, the company offers a wide range of products including personal care, household items, and pharmaceuticals. With over 4,000 stores, Rossmann's success is driven by its commitment to providing affordable and quality products to customers. As the retail industry becomes increasingly data-driven, accurately predicting sales is essential for maintaining the company's competitive edge, optimising inventory and maintaining customer satisfaction. Failure to forecast sales accurately can lead to overstocking, resulting in wasted resources, or understocking, leading to missed revenue opportunities and unsatisfied customers. To address this issue, we aim to develop a machine learning model capable of predicting daily sales for Rossmann Stores, enabling the company to make data-driven decisions and enhance operational efficiency.

Data Overview

The datasets provided for this project comes from Rossmann. For this project, we were tasked to work with two CSV files: the store dataset and the train dataset. The store dataset contains the store details of 1115 Rossmann stores with 10 variables. Next, the train dataset contains a total of 1017209 observations and 9 variables. These observations are the sales data of 1115 stores across various locations collected from Rossmann stores over a 31-month period, from 2013-01-01 to 2015-07-31. Both datasets contain one common variable which describes the ID of a store.

The combined unique variables from both datasets are: *Store* – a unique Id for each store; *Sales* – the turnover for any given day; *Customers* – the number of customers on a given day; *Date* – the date of sales, *DayOfWeek* – numerical categorisation of the day of the week (1 indicates Monday), *Open* – an indicator for whether the store was open; *StateHoliday* – indicates a state holiday; *SchoolHoliday* – indicates if the (Store, Date) was affected by the closure of public schools; *StoreType* – differentiates between 4 different store models (a, b, c, or d); *Assortment* – describes the assortment level of a store (a = basic, b = extra, c = extended); *CompetitionDistance* – distance in meters to the nearest competitor store; *CompetitionOpenSinceYear*, *CompetitionOpenSinceMonth* – gives the approximate year and month of the time the nearest competitor was opened; *Promo* – indicates whether a store is running a promotion on that day; *Promo2* - Promo2 is a continuing and consecutive promotion for some stores; *Promo2SinceYear*, *Promo2SinceWeek* – describes the year and calendar week when the store started participating in Promo2; *PromoInterval* – describes the consecutive intervals of Promo2.

Sales is our target variable, while the rest are our features. For our project, we will focus on analysing only the top 20% of stores in terms on sales. Thus, before we can proceed with exploratory data analysis and subsequent modelling, we need to preprocess these datasets into a single data frame for further use.

Data Preprocessing

The two original datasets, *Rossmann_store* and *Rossmann_train* are as shown in Figure 3.1 and Figure 3.2 respectively. Evidently, there were several aspects of our datasets—such as missing values—that had to be dealt with before we could proceed with analysis. This involved data cleaning, data wrangling and data engineering to obtain a single dataframe of good quality so that we could extract reliable results from the data.

	Store	StoreType	Assortment	CompetitionDistance	CompetitionOpenSinceMonth	CompetitionOpenSinceYear	Promo2	Promo2SinceWeek	Promo2SinceYear	PromoInterval
0	1	c	a	1270.0	9.0	2008.0	0	NaN	NaN	NaN
1	2	a	a	570.0	11.0	2007.0	1	13.0	2010.0	Jan, Apr, Jul, Oct
2	3	a	a	14130.0	12.0	2006.0	1	14.0	2011.0	Jan, Apr, Jul, Oct

Figure 3.1: The first few observations of the store dataset, *Rossmann_store*

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday
0	1	5	2015-07-31	5263	555	1	1	0	1
1	2	5	2015-07-31	6064	625	1	1	0	1
2	3	5	2015-07-31	8314	821	1	1	0	1

Figure 3.2: The first few observations of the train dataset, *Rossmann_train*

Filtering and Merging

First, we are only interested in the top 20% of stores in terms of sales. Next, we want a single dataframe containing all the relevant data so we can analyse the data as a whole.

We began by calculating the average sales for each store ID in the train dataset. These average sales were stored in a separate dataframe, then later on concatenated to the store dataset. Now that the store dataset contained an extra column—the average sales—for each store, we filtered the store dataset by average sales of above the 0.8 quantile (top 20%). Then, we obtained the store IDs of these stores that made the cut, and filtered the train dataset to only include sales history of these store IDs. We then removed the average sales column previously introduced as it serves no further purpose.

With our filtered store and train datasets, we merged them based on their common column *Store* (the store IDs) to obtain a single dataframe of 18 variables. This dataframe contains the historical sales data of the top 223 stores in terms of sales. Figure 3.3 shows the first few observations of this dataframe.

	Store	StoreType	Assortment	CompetitionDistance	CompetitionOpenSinceMonth	CompetitionOpenSinceYear	Promo2	Promo2SinceWeek	Promo2SinceYear	PromoInterval	DayOfWeek	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday
2015-07-31	4	c	c	620.0	9.0	2009.0	0	0	0	NA	5	1395.0	1498	1	1	0	1
2015-07-31	7	a	c	24000.0	4.0	2013.0	0	0	0	NA	5	15344.0	1414	1	1	0	1
2015-07-31	24	a	c	4590.0	3.0	2000.0	1	40	2011	Jan, Apr, Jul, Oct	5	14190.0	1082	1	1	0	1

Figure 3.3: The combined dataframe of *rossmann_store* and *rossmann_train*

Renaming Variables and Converting Datatypes

To reduce confusion, we renamed the variable *PromoInterval* to *Promo2Interval* since it refers to the interval of *Promo2*.

Subsequently, we did some conversion of datatypes. The datatype of *Promo2SinceWeek* and *Promo2SinceYear* were initially objects; these were converted to type int64. Additionally, *Sales* was converted from int64 to float64. This is because in the context of our project, *Sales* is a continuous variable that we intend to predict later on. Further, we converted *Date* to type datetime and set the *Date* as the index of our dataframe.

Imputation

Next, our data contained missing values in five columns: *Promo2SinceWeek*, *Promo2SinceYear*, *Promo2Interval*, *CompetitionOpenSinceMonth* and *CompetitionOpenSinceYear*.

For columns related to *Promo2*, upon further inspection, we noticed that these values were only missing for stores that were not participating in *Promo2*. Thus, for the continuous variables (*Promo2SinceWeek*, *Promo2SinceYear*), we imputed the missing values in these columns with 0. Then, the categorical variable *Promo2Interval* had its missing values imputed with "NA" (short for "Not Applicable") for such stores.

For the actual missing data in *CompetitionOpenSinceMonth* and *CompetitionOpenSinceYear*, we tried various imputation methods such as simple imputation and k-Nearest Neighbours (kNN). We then evaluated these imputation methods to determine which was the most suitable.

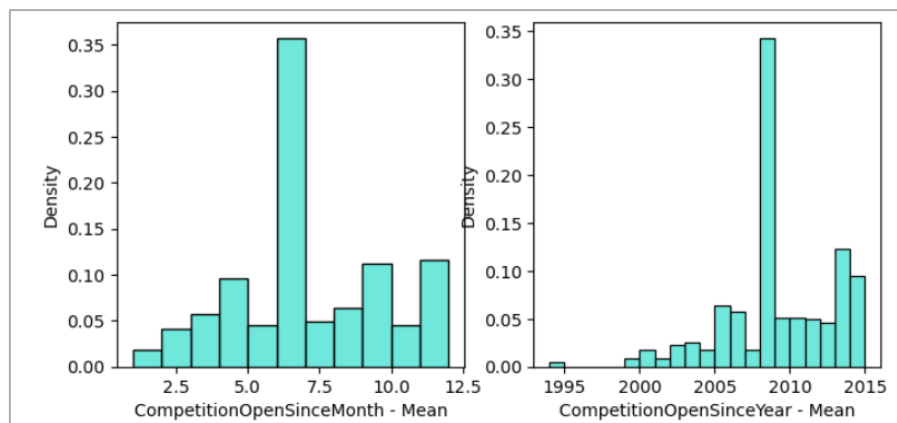


Figure 3.4: Histogram of missing data column using simple imputation

For simple imputation, we used mean and median imputation for missing values and observed the effect on feature distributions. To be specific, we observe an obvious peak in one of the

bins (Figure 3.4). This indicates that simple imputation is straightforward but resulted in unrealistic spikes in distribution graphs, thus it was not a great fit for our dataset.

Hence, we continued the imputation process with a more advanced KNeighborsRegressor imputation technique. In this context, we normalized the data and explored different k values via the elbow method.

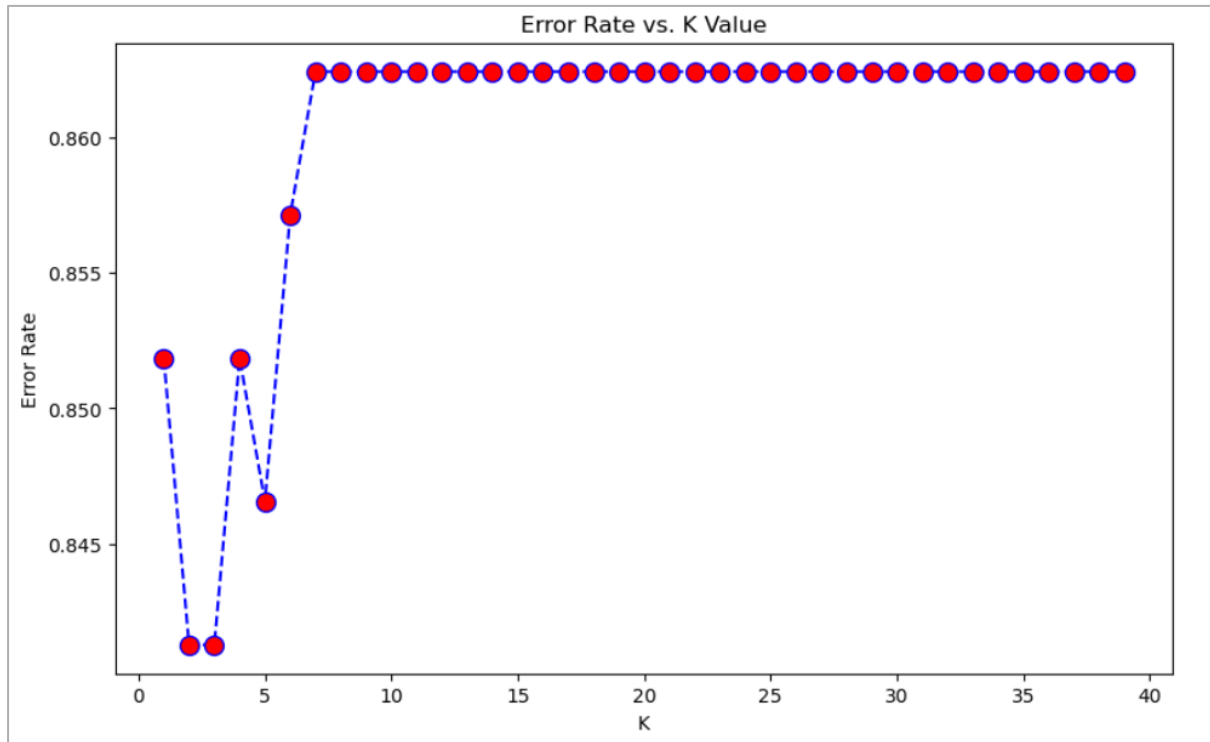


Figure 3.5: *Elbow method to find the optimal k value*

As shown in figure 3.5, we deduced that the most optimal k value is 1, where the error rate is lowest. By using the optimal k value, we instantiated the KNeighborsRegressor while fitting it with our features space, namely *CompetitionOpenSinceMonth* and *CompetitionOpenSinceYear*. Next, the target variable, *Sales* is also fitted into the model. The missing value will then be imputed based on the output of the kNN model.

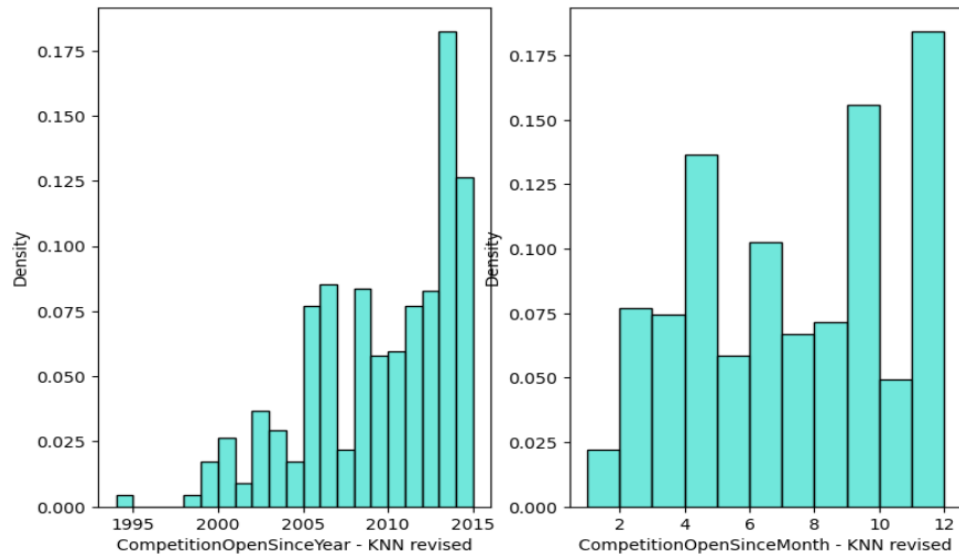


Figure 3.6: Histogram of missing data column using kNN imputation

According to figure 3.6, we can see that there is a bell shape curve histogram overall. This indicates that kNN imputation had imputed the values in a decent way where no extreme values are shown. KNN imputation also offered a better fit compared to simple imputation, maintaining a more even and normal distribution of the data. However, we found that the histograms do not show a perfect bell shape curve. This could be further investigated by looking at the correlation heatmap.

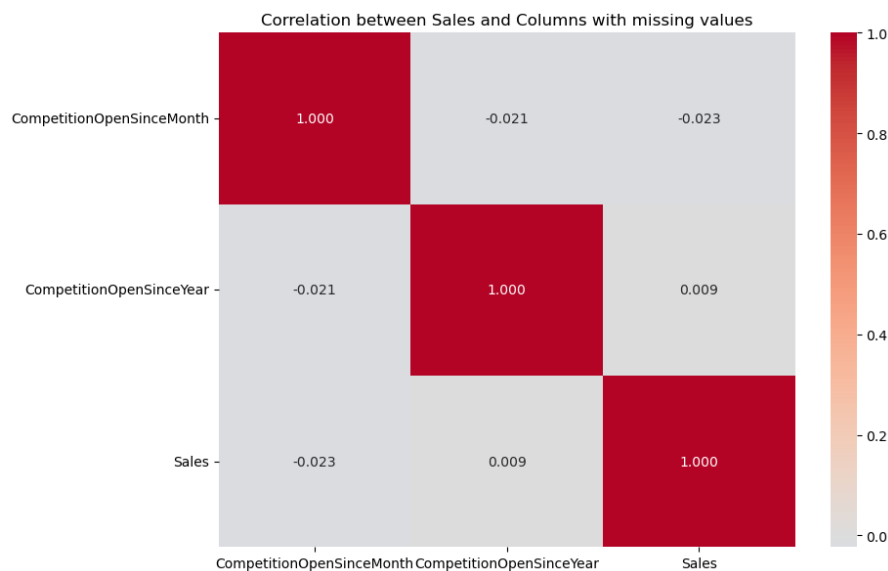


Figure 3.7: Pearson Correlation heatmap between Sales and variable with missing values

According to figure 3.7, we can see that correlation between Sales and CompetitionOpenSinceMonth, CompetitionOpenSinceYear is too small. This indicates that the variable with missing values has minimal to no contribution to the target variable, Sales. Therefore, we will never obtain a perfect bell shape curve upon any way of imputation.

Encoding Categorical Data

Any categorical data present in our dataframe must be converted to numerical form because models and calculations can only be performed on numerical values. This ensures easier model integration. For our data, we employed one-hot encoding to ensure there is no implicit ordering of categories from the encoding procedure.

We found that there are multiple variables that are categorical, including *DayOfWeek*, *StoreType*, *Type of Store*, *Assortment*, *Promo2Interval* and *StateHoliday*. These should be converted to binary columns using pandas method `get_dummies`. In short, we created columns which are formed by each and every of these strings, that could be stated by binary data 0 or 1, using the method as per in Figure 3.8.

```
# Initialise a function since we will be using it multiple times
def create_dummies(df, column_name, rename_dict):
    dummies = (pd.get_dummies(df[column_name], dtype=int)
               .rename(columns=rename_dict)
               )
    return dummies

dummies_DayOfWeek = create_dummies(rm_imputed, 'DayOfWeek', {
    1: 'Monday',
    2: 'Tuesday',
    3: 'Wednesday',
    4: 'Thursday',
    5: 'Friday',
    6: 'Saturday',
    7: 'Sunday'})

dummies_Store_Type = create_dummies(rm_imputed, 'StoreType', {
    "a": "StoreTypeA",
    "b": "StoreTypeB",
    "c": "StoreTypeC",
    "d": "StoreTypeD"})

dummies_Assortment = create_dummies(rm_imputed, 'Assortment', {
    "a": "AssortmentA",
    "b": "AssortmentB",
    "c": "AssortmentC"})

dummies_Promo2Interval = (create_dummies(rm_imputed, 'Promo2Interval', {
    "Jan, Apr, Jul, Oct": "Promo2_Int_TypeA",
    "Feb, May, Aug, Nov": "Promo2_Int_TypeB",
    "Mar, Jun, Sept, Dec": "Promo2_Int_TypeC"})
    .drop(columns=["NA"]))
# we doesn't need any of NA as 0 for these 3 cols could represent no promo dy

dummies_StateHoliday = (create_dummies(rm_imputed, 'StateHoliday', {
    "a": "PublicHoliday",
    "b": "EasterHoliday",
    "c": "Christmas"})
    .drop(columns=["0", "0"]))
# in the original dataset, both int(0), and str(0) are stored for indication of "no holiday"
# we doesn't need any of these as 0 for "a", "b", and "c" could represent a "no holiday" dy
```

Figure 3.8: *Creating dummies using pandas method `get_dummies`*

This method is particularly important when dealing with Multiple Linear Regression, where the independent variables need to be numeric. Including categorical variables without transformation can lead to incorrect results or even model failure. By converting them into binary columns, each category becomes a feature, allowing the model to interpret and weigh their significance properly.

Exploratory Data Analysis (EDA)

Initial EDA

Before we delve into more complex analysis, it is essential to gain a preliminary understanding of the dataset through exploratory data analysis. This will allow us to identify key patterns, trends and potential anomalies in the sales data, which can provide valuable insights for subsequent modelling efforts. The following section will outline the findings from the initial EDA, focusing on the sales trends over time and daily distribution of sales values.

In our initial exploration of the dataset, we performed a series of analyses to better understand the underlying structure of the data. First, we visualised the distribution of sales using a histogram.

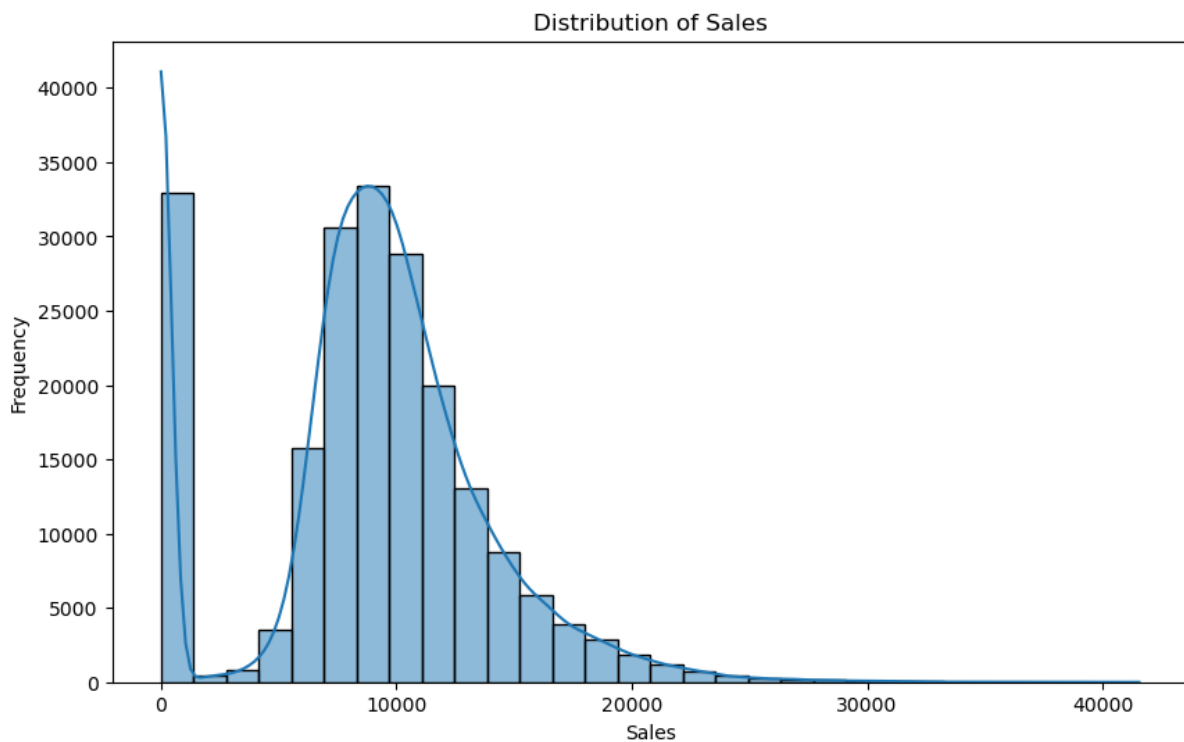


Figure 4.1: *Histogram of the distribution of sales*

Figure 4.1 shows that the distribution is right-skewed, meaning a large proportion of the sales fall within the lower range, while fewer purchases occur at higher sales levels. A significant peak is visible around lower sales values, indicating that most sales are clustered in this range. We can also observe that as the sales values increase, the frequency of the transactions decreases significantly, with only a few transactions exceeding 20,000.

Time Series Analysis

Next, we conducted a time series analysis to observe the trend of sales throughout the years.

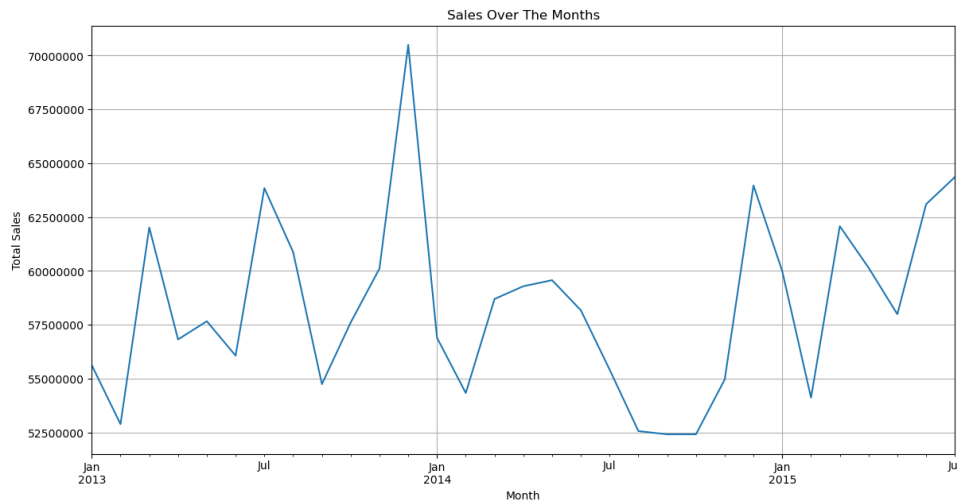


Figure 4.2.1: Time series analysis of sales trend throughout the years

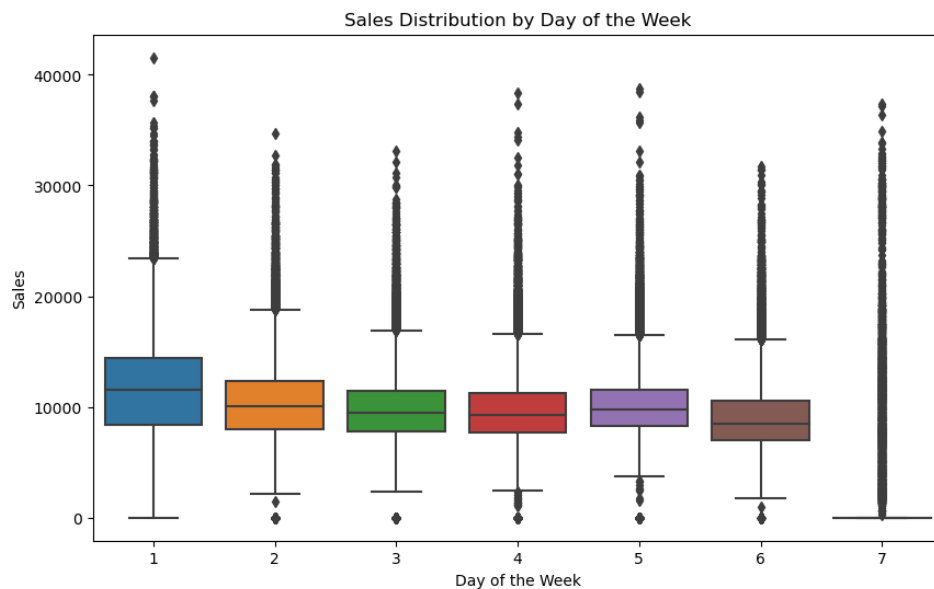


Figure 4.2.2: Boxplot of the trends of sales throughout the days of the week

The time series analysis of sales, in Figure 4.2.1, reveals several fluctuations over the years, with a notable peak in December 2013. This period shows the highest sales recorded, likely due to holiday or seasonal promotions. However, following this surge, sales experience a consistent decline through to October 2014. From this low point, there is a gradual increase, particularly towards December 2014, where sales once again rise, although not reaching the same levels as the previous year.

In terms of weekly sales patterns, the boxplot in Figure 4.2.2 illustrates key trends across different days of the week. Monday (Day 1) has the highest median sales and the widest spread of values, as seen in the length of the whiskers. This suggests that there is considerable variation in sales on this day, possibly due to varying customer demand or store promotions. In contrast, Sunday (Day 7) exhibits the lowest median sales but has the most significant number of

extreme outliers, indicating that some stores experience exceptionally high sales despite the lower median. It is to note however that in Germany, where Rossmann is primarily based, retail stores are often closed on Sundays, with many Rossmann stores included. Thus, the outliers may not be actual outliers in this context; they are just the average sales for those stores which were open.

From Tuesday to Saturday (Days 2 to 6), sales distributions remain relatively consistent with similar median values and tighter ranges. These days exhibit fewer extreme variations, showing more stable sales patterns. Outliers, while present for every day, are particularly notable on Monday and Saturday. These extreme values may point to certain stores or locations experiencing unusually high sales volumes on specific days of the week, likely driven by localized promotions or other factors.

Overall, the initial exploratory analysis reveals key patterns in sales, particularly the seasonality with peaks during December and a noticeable decline from early 2014 until October. This highlights the importance of planning for both peak and low sales periods. Mondays show the highest median sales, while Saturdays have the lowest, though with extreme outliers, indicating occasional high sales in some locations. The presence of outliers, especially on certain days, suggests variability across stores, which could be explored further. Overall, understanding these trends is crucial for optimizing sales strategies and forecasting future performance based on seasonal and weekly demand patterns.

Analysis with Rolling Mean of Sales

To observe long-term patterns, we utilised the rolling mean of the sales with a weekly window in the most recent year of the data, 2015. A rolling mean helps smoothen out the noise in data by calculating the mean of sales for a day based on mean sales of the days before itself, within the window specified. We plot the rolling mean of *Sales* against other various features, namely the promotions (*Promo*, *Promo2*) and store assortment type (*Assortment*).

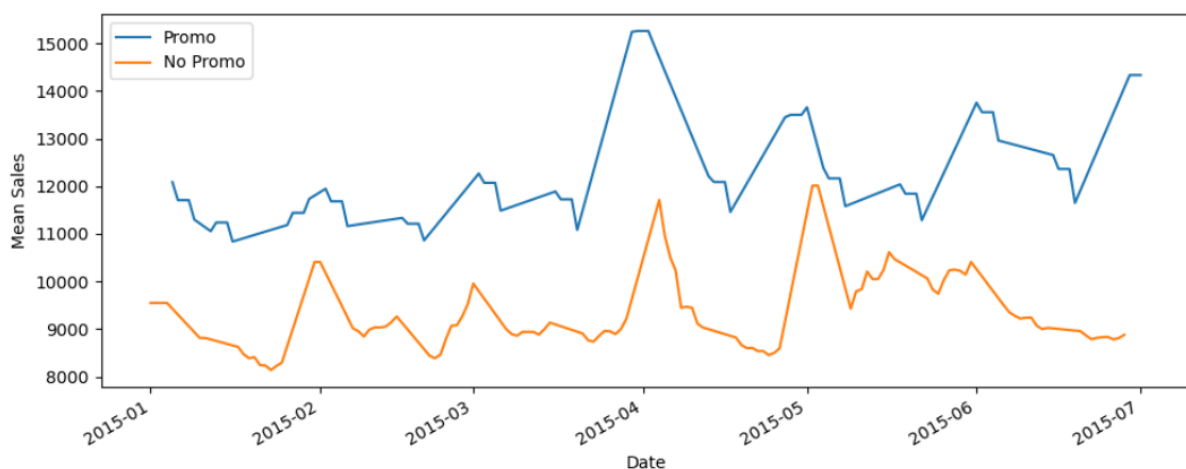


Figure 4.2.3: *Rolling mean of sales for Promo from January to June in 2015*

First, in Figure 4.2.3, we compared the rolling mean of sales for two groups: stores with and without the ongoing promotion *Promo* on each day. Aligning with our expectations, promotions

seemed to have a noticeably positive effect on sales. This is because lower prices would make products more affordable and motivate consumers to get the most of their money's worth.

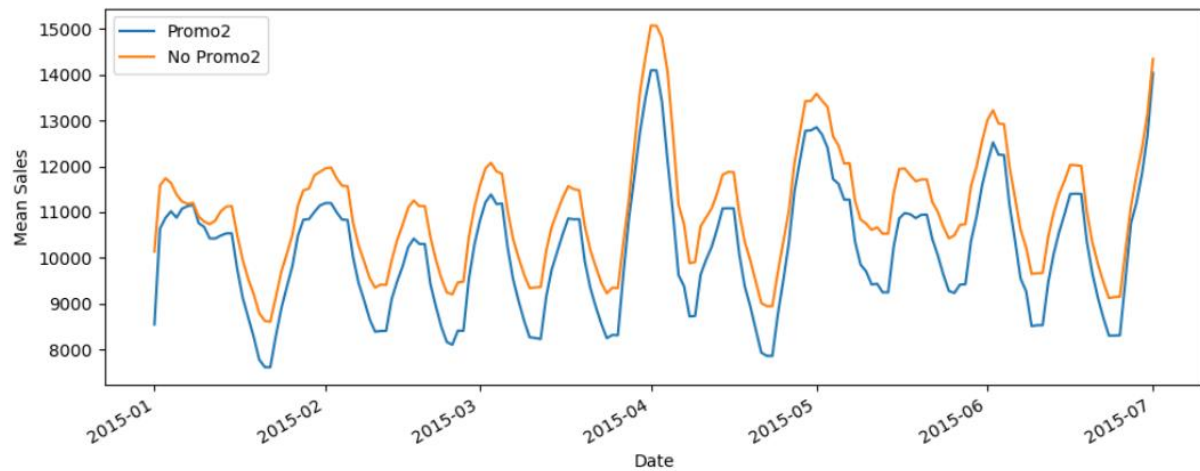


Figure 4.2.4: Rolling mean of sales for Promo2 from January to June in 2015

Contradicting our expectations, *Promo2* seemed to have little, or a slightly negative impact on sales (Figure 4.2.4). This may be due to the nature of *Promo2* being a fixed interval promotion. Thus, consumers of the stores participating in *Promo2* may be used to seeing deals, and so they are less compelled by such promotions (Talbot, 2019). Furthermore, such consumers may put off purchases, awaiting the next promotion period for it may bring even better deals. Without *Promo2* however, consumers have no clear indication of any upcoming promotions and thus just spend as needed immediately.

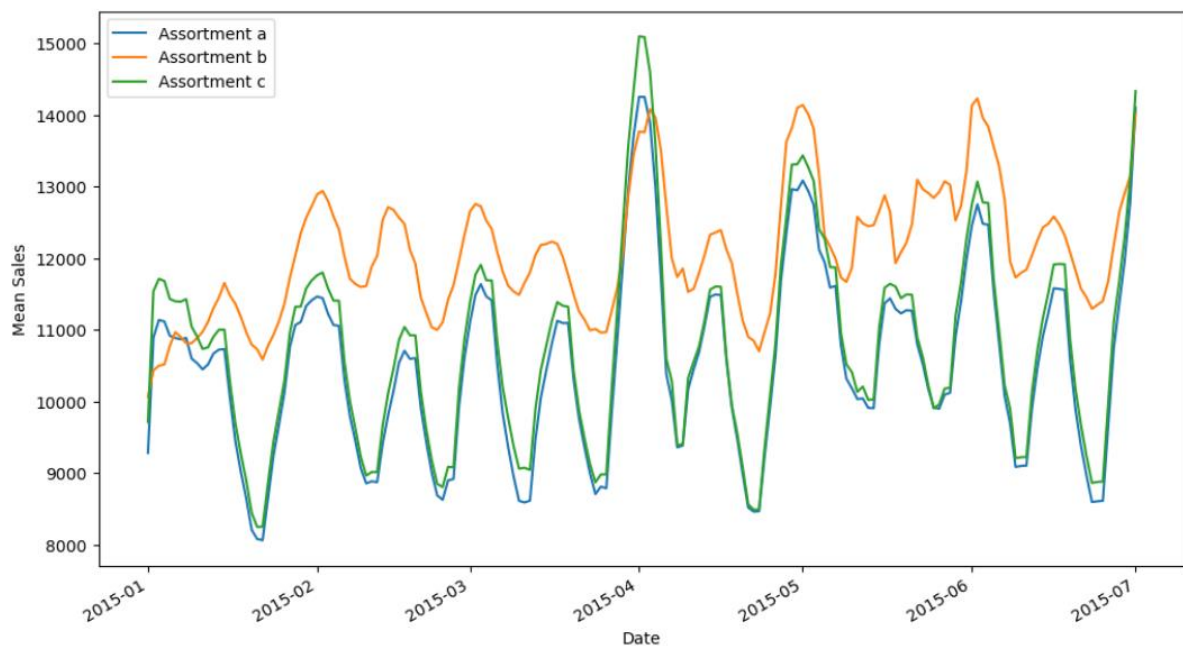


Figure 4.2.5: Rolling mean of sales for Assortment from January to June in 2015

Store assortment level seems to have a mixed effect on sales; on most occasions, assortment type *b* (extra) does better than type *a* or *c* (basic and extended, respectively) (Figure 4.2.5).

This phenomenon may be due to basic stores not carrying enough inventory or popular items. On the other hand, extended stores may be too overwhelming for consumers and may contain more uninteresting items than not, which leads to consumer dissatisfaction (Shakedd, 2023). In contrast, a store with a medium range of assortment (i.e., assortment type b) can contain a wide enough variety without being too overwhelming, thus encouraging a positive shopping and spending experience.

In conclusion, from the few investigated features, only *Promo* seems to have an obvious positive relation to *Sales* thus far.

Multivariate Linear Regression (MLR) model

To identify the top few features affecting sales amongst all the 18 features, we used a MLR model to cumulatively analyse the relations between our target and features.

With an MLR model, we aim to create a matrix of independent variables (features), and targeted variable ('Sales'). Specifically, we are building models in form of:

$$Sales_{pred} = intercept + c_1x_1 + c_2x_2 + c_3x_3 + \dots + c_nx_n + \epsilon$$

which:

$Sales_{pred}$ stands for the predicted sales, *intrecept* stands for the expected value of $Sales_{pred}$ when all coefficients are 0, c_n stands for the coefficient of the certain variable, x_n stands for the variable, and ϵ stands for the error term (term that is not explainable).

Building MLR

```
def construct_MLR(X, Y):
    X_train, X_test, Y_train, Y_test = train_test_split(
        X, Y, test_size=0.2,
    )
    # Construct and fit the model
    linear = LinearRegression()
    linear.fit(X_train, Y_train)

    # Evaluate model performance.
    training_predictions = linear.predict(X_train)
    testing_predictions = linear.predict(X_test)

    # Create a table of the various scores.
    table = pd.DataFrame({
        "R^2": {
            "train": r2_score(Y_train, training_predictions),
            "test": r2_score(Y_test, testing_predictions)
        },
        "RMSE": {
            "train": mean_squared_error(Y_train, training_predictions, squared=False),
            "test": mean_squared_error(Y_test, testing_predictions, squared=False),
        },
        "MAE": {
            "train": mean_absolute_error(Y_train, training_predictions),
            "test": mean_absolute_error(Y_test, testing_predictions),
        },
    })
    return table
```

Figure 4.3.1: MLR construction method

As per in Figure 4.3.1, we were splitting dataset into 80% training set with 20% testing set, which suggests that we were using 80% of the data for training the model and 20% for testing

its accuracy. This helped in achieving optimal performance as this provide a balance between evaluating the model, and the training of model.

Analysis of Correlation

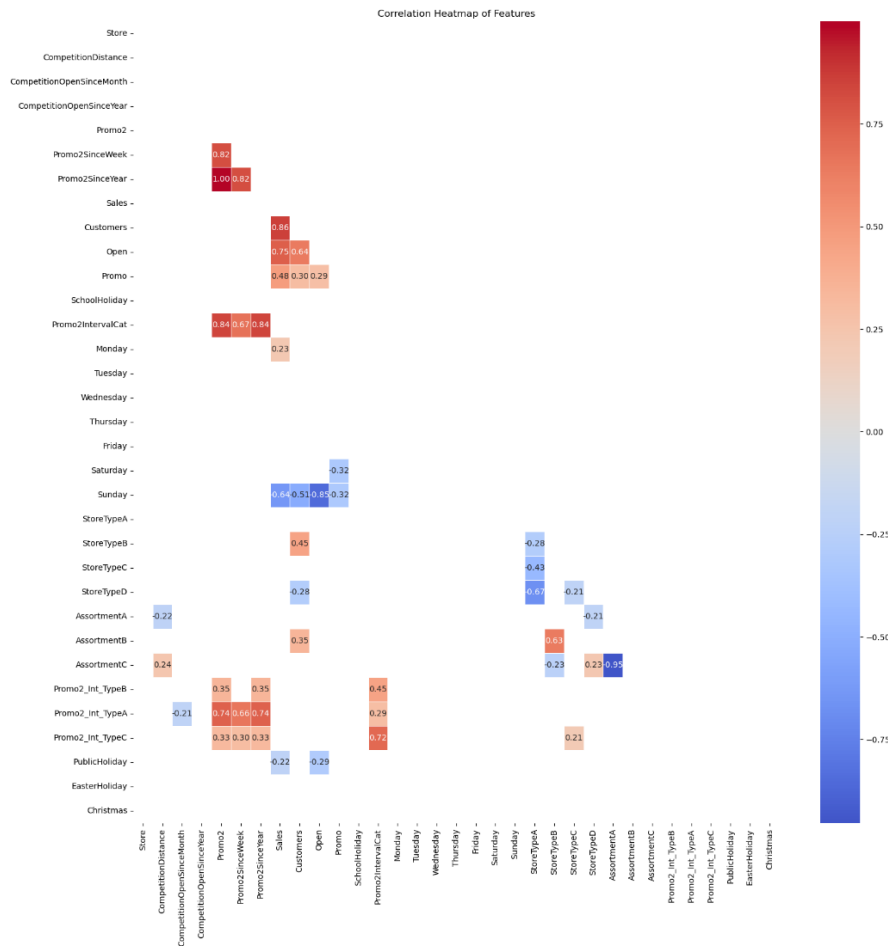


Figure 4.3.2: *Pearson Correlation Heatmap of Features*

From Figure 4.3.2, we generated a heatmap with filtered correlation, that, we filtered out correlations that their absolute value is less than 0.2, as they are too weakly correlated to our targeted variables, yet not bringing about any effect to other variables.

Simple modelling

However, still we did some simple modelling to test the feasibility of the MLR model. From the matrix of independent variables, we were able to build a model, which was built up of all variables.

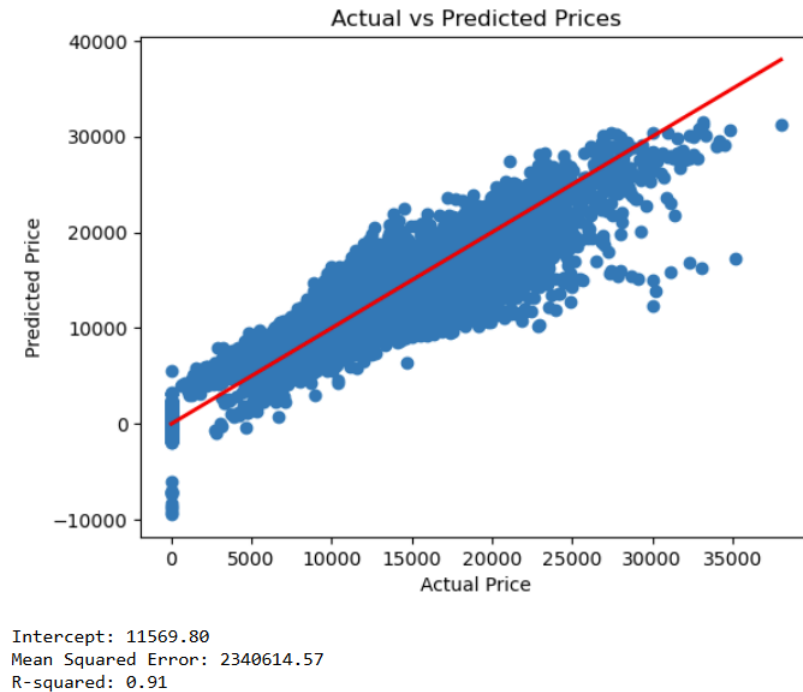


Figure 4.3.3: *Comparison between Actual and Predicted Prices with Simple MLR model*

Figure 4.3.3 shows a result of modelling, which took all of the variables involved in in the dataframe. The intercept suggested that if all factors influencing sales were at their baseline levels, we would still set the baseline sales value as 11569.80.

The mean squared error measured the average squared difference between the predicted and the actual values of the dependent variable, targeted sales.

Besides the R-squared value of 0.91 suggests that 91% of the variability in Sales can be explained by the model. This shows that the MLR model overfits. It is receiving noises from most of the factors, regardless of relevance. It does not generalise well to the testing data.

Thus, we should not be accepting this model.

Initial Feature Selection

By conducting investigation on correlation between variables and targeted feature, our initial step is to filter out all variables that $-0.2 < \text{correlation} < 0.2$, which variables that are poorly correlated to the targeted variable. As the result, we came up with 6 potential variables, namely 'Customers', 'Open', 'Promo', 'Monday', 'Sunday' and 'PublicHoliday'.

Analysis of Coefficients

Next, we may investigate the importance of model coefficients by its coefficient values and standard deviation. As we have to consider both coefficient values and standard deviations, we would consider the importance of each coefficient by multiplying coefficient by the standard deviation. This was done as by the code in Figure 4.3.4.

```
coefs = pd.DataFrame(
    linear.coef_ * X_train.std(axis=0),
    columns=['Coefficient importance'], index=feature_names
)
# do a similar plot of this variable, as above
coefs.plot(kind='bar', figsize=(13, 4))
plt.title('Linear Regression')
plt.axhline(y=0, color='.5')
plt.subplots_adjust(left=.3)
```

Figure 4.3.4: *Analysis of Coefficients*

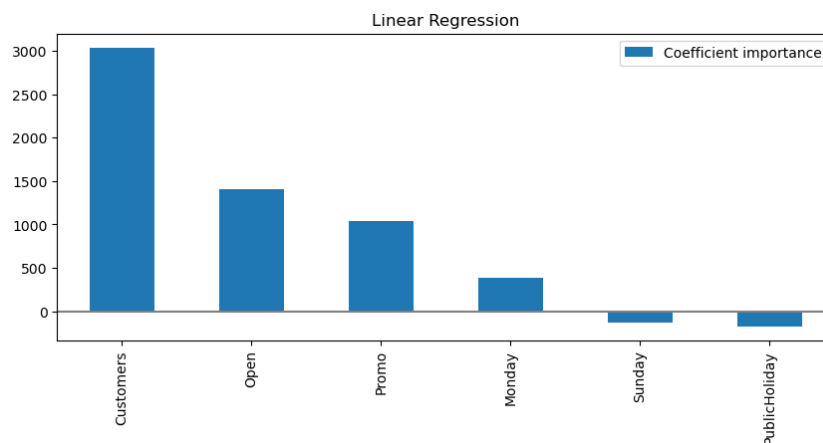


Figure 4.3.5: *Coefficient Importance*

The result in Figure 4.3.5 suggested the order of importance of variables. In this figure 'Customers', 'Open' and 'Promo' seemed to be the most important variable related to the targeted feature.

Normalisation

To compare the importance of coefficients, we normalised the independent variables from the start. The independent variables is normalised by instantiate nX , which:

$$nX = X - X.mean() / X.std()$$

Regularisation

Then, we determined whether we should be using Lasso or Ridge regularisation methods. We considered the case that:

- Feature selection is important – Not all features contribute to the outcome
- Interpretability is crucial – We require a easy-interpretable model
- Strong feature correlation – e.g: 'Sunday' and 'Open' has a high correlation.

Under this case, Lasso regularisation method would be preferable than Ridge.

We performed hyperparameter tuning for a Lasso regression model to find the optimal value of regularisation parameter ' α '. By generating 50 values between 10^{-4} and 10^1 spaced logarithmically, we performed a Lasso regularisation for each and calculated the optimal value of α . In this case, the optimal value of α is 0.0176.

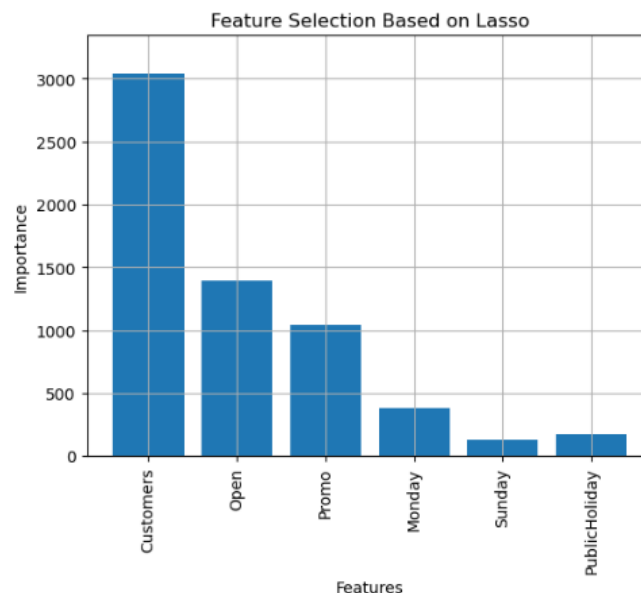


Figure 4.3.6: *Feature Selection Based on Lasso Regularisation*

The Lasso Regularisation result in Figure 4.3.6 suggests a similar result as Figure 4.3.5, which 'Customers', 'Open' and 'Promo' seemed to be the most important variable related to the targeted feature.

Multicollinearity

The Lasso regularisation result also presented the way it detected redundancy of features to the model. For example, it shrunk the coefficients including 'Sunday' and 'PublicHoliday' to zero, as they are strongly related to the other variables. For example, 'Sunday' is highly correlated to the variable 'Open', indicating that 'Sunday' is relatively redundant in the model. The result of determining existence of multicollinearity stabilised the model and improved its generalisability.

MLR model Result

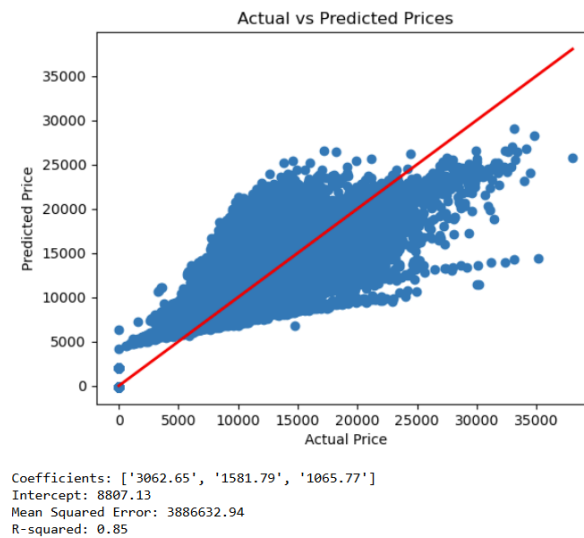


Figure 4.3.7: Final MLR Model

Therefore, we used the 3 most important factors, which are 'Customers', 'Open' and 'Promo', as the result suggested by Analysis of Coefficient using coefficients and standard deviation, also by the result from Lasso Regularisation. Comparing to the initial MLR model, this model have its generalisability increased ($r^2 = 0.85$), that 85% of the variance in the sales data is explained by the model. Although this model have high value of MSE, the lower amount of value in intercept suggested a highly generalised model compared to the initial model.

Thus, the final model (Figure 4.3.7) is interpreted as:

$$Sales_{pred} = 8807.13 + 3062.85x_1 + 1581.79x_2 + 1065.77x_3$$

which:

$Sales_{pred}$ stands for predicted sales, x_1 stands for number of customers, x_2 stands for the availability of the store, and x_3 stands for the availability of promotion.

Predictive Modelling

From our feature selection previously, we have determined that *Open*, *Promo*, and *Customers* are our key factors. Hence, we proceeded with modelling based on these features and our target, *Sales*.

For modelling, we first split our data into training and testing sets. Since our goal is to forecast the sales four weeks in advance, we manually performed a time-based split on our data based on the index, *Date* (Figure 5.1). It is to note that our data was already sorted by date beforehand. As a result, we obtain our training set which contains all the historical sales data excluding the last four weeks (2013-01-01 to 2015-07-03), and our testing set which contains the historical sales data of the last four weeks recorded in the data (2015-07-04 to 2015-07-31).

```
# define training and testing sets, split between last 4 weeks of dataset and the rest of the data
rm_train = rm_imputed.loc[pd.Timestamp('2015-07-03'),:]
rm_test = rm_imputed.loc[pd.Timestamp('2015-07-04'),:]

X_train = rm_train[['Open', 'Promo', 'Customers']]
X_test = rm_test[['Open', 'Promo', 'Customers']]
y_train = rm_train['Sales']
y_test = rm_test['Sales']
```

Figure 5.1: Manual time-based split of the data into training and testing sets

Moving on, we utilised decision tree-based algorithms for our predictive modelling, starting from a single decision tree regressor, then random forests. This is because decision tree-based algorithms can easily handle any complex relationships between our target and features.

Single Regression Tree

A decision tree is a tree-like model, where each branch is a possible decision and each node is a corresponding consequence. The final nodes at the end of the tree are known as leaf nodes, which each represent a predicted target value. A regression tree is a decision tree utilised for regression. A regression tree's target values are the mean value of all the samples that fall under a given node. The maximum depth of a tree can be specified when instantiating the tree model. A tree with greater depth will have more branches and consequently more nodes, thus resulting in more intricate predictions.

We start off with fitting a single regression tree model to the data. This is implemented by instantiating `DecisionTreeRegressor` class from `sklearn`, then fitting this model to our training sets `X_train` and `y_train`. We did not specify a maximum depth and instead let the model decide on an optimal depth, for which it returned a model with a depth of 38. Then, we used this model to predict the target for `X_test` and compared it against `y_test`. The training score of this model is 0.875, while its testing score is 0.821. Because of the high training score and lower testing score, we suspected the likelihood of overfitting. This means that the tree has too many splits, which may have captured noise in the process. So, we looped through different values of maximum depth in the range of 1 to 38 for a regression tree fit to our data, and calculated the training and testing scores of each tree.

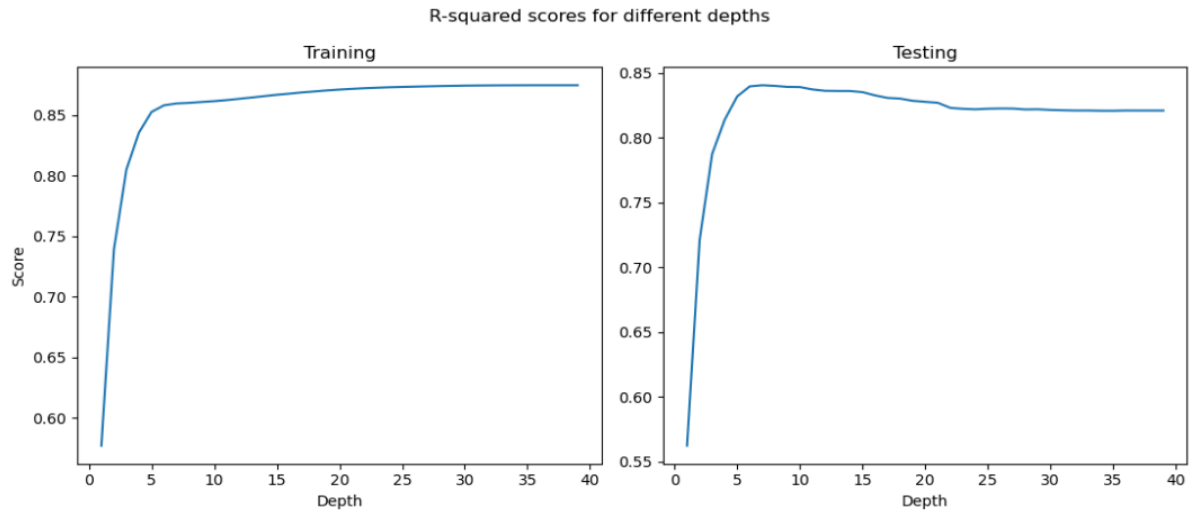


Figure 5.2.1: *Training vs. Testing scores for regression trees of varying depths*

From the plot in Figure 5.2.1, we deduced that the optimal maximum depth of a regression tree for our data occurs at a depth of 7. At a depth of 7, the testing score is the highest, and there is the least difference between training and testing scores. Thus, fitting a regression tree with maximum depth 7 to our training set then making predictions for our testing set gave a training score of 0.86 and a testing score of 0.84. Using the predictions from this improved model, we can visualise the predictions of the total daily sales four weeks in advance in comparison to the actual sales (Figure 5.2.2).

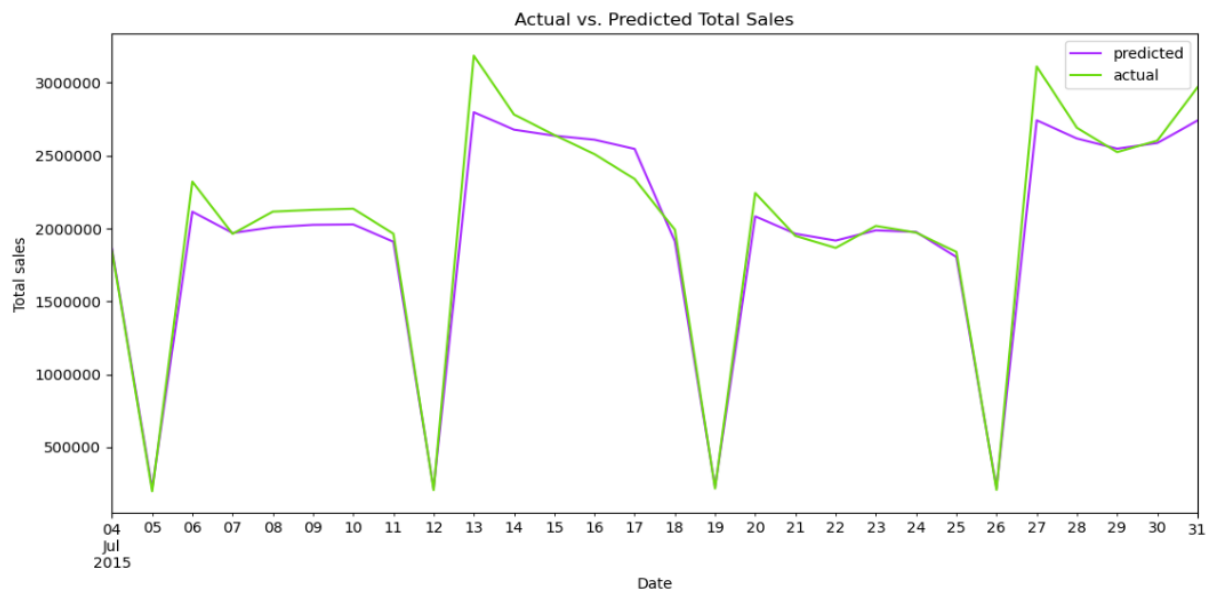


Figure 5.2.2: *Actual vs. predicted total sales for Rossmann drugstores in the most recent four weeks*

Observing Figure 5.2.2, it appears that the single regression tree estimates the pattern of total daily sales fairly well, however it fails to predict accurately the peaks and nuances in the pattern of the actual sales. This may be because a single regression tree is built from the entire training set at once, so it models the training data very well but in turn lacks generalisation on unseen

data (Data Headhunters, 2024). Thus, any patterns in unseen data that are not familiar in the training data will be missed out by the model.

Ensemble Learning

Previously we have done predictions based on a single model, but the Single Regression Tree fails to predict the peaks of the actual sales accurately. Hence, we are using ensemble learning to address this issue. Ensemble learning is a machine learning technique that combines two or models to produce more accurate predictions (Murel & Kavlakoglu, 2024). In our project, we are considering Random Forest Regression, Bagging, Boosting and Stacking.

Model Evaluation

Before we are combining predictive models to form ensembles, we build a function to evaluate the performance of our ensemble models. Since we are working on continuous data to predict the sales, the metrics we are using are R^2 score, mean squared error (MSE) and mean absolute error (MAE). In statistical terms, R^2 score measures how well a regression model is fitted from a range of 0% to 100%. On the other hand, mean squared error is the average squared difference between predicted and actual values while mean absolute error is the average of the absolute difference.

Before evaluating the ensemble models, our function will first fit our model using the `X_train` and `y_train` defined above then carrying out cross-validation. Cross validation is an important technique to reduce overfitting by training several models on training data subsets. In our case, we are using the repeated k-fold cross validation with the standard 5 splits, meaning that we are repeating the process 5 times. After cross validation, this function will use `X_train` and `X_test` to perform predictions then combine the predictions with `y_train` and `y_test` again to calculate the metrics.

```
def evaluate_model(model):
    # Construct and fit the model
    model.fit(X_train,y_train)

    # carry out cross validation
    subset_scores = cross_validate(
        model, X_train, y_train,
        cv=RepeatedKFold(n_splits=5, n_repeats=5, random_state=np.random.RandomState(31287)),
        return_estimator=True,
    )
```

Figure 5.3.1: Code snippet for cross validation

Random Forests Regression

Random Forest Regression has the almost same working principle as Decision Tree, just that this algorithm combines the output of many decisions tree. We will just instantiate the RandomForestRegressor then call the evaluate_model function.

Later, we will perform hyperparameter tuning to find the best configuration for the parameter to tweak our model performance for optimal results. As asserted by Probst et al. (2019), tuning parameters must be done meticulously to optimize the maximum performance. In extend to that, we decided to use to use the RandomizedSearchCV because our search space is too large and exhaustive. Furthermore, randomized search is much preferable in terms of parameter exploration and time efficiency especially in complex models. Among all of the parameters, n_estimators is always the most significant one as it controls the base number of models to loop.

```
# Create the random grid
random_grid = {'n_estimators': [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)],
               'max_features': ['auto', 'sqrt'],
               'max_depth': [int(x) for x in np.linspace(10, 110, num = 11)],
               'min_samples_split': [2, 5, 10],
               'min_samples_leaf': [1, 2, 4],
               'bootstrap': [True, False]}

# Random search of parameters, using 3 fold cross validation then search across 100 different combinations, and use all available cores
rf_random = RandomizedSearchCV(estimator = rfr, param_distributions = random_grid, n_iter = 100, cv = 3, verbose=2, random_state=42, n_jobs = -1)

# Fit the random search model
rf_random.fit(X_train, y_train)
```

Figure 5.3.2: Code for RandomizedSearchCV

```
RandomizedSearchCV
best_estimator_: RandomForestRegressor
RandomForestRegressor(max_depth=10, max_features='sqrt', min_samples_leaf=2,
                      min_samples_split=10, n_estimators=1000, random_state=13)
```

Figure 5.3.3: Output for RandomizedSearchCV using RandomForestRegressor as the base

It can be seen that the most optimal parameter is a maximum depth of 10, minimum samples leaf of 2, minimum samples split of 10 and a base n estimator of 1000.

We then fit the Random Forest Regressor model again using the parameter found from the hyperparameter tuning process. After fitting, we call the evaluate_model function again to check on the model's performance.

	R ²	RMSE	MAE
train	0.862026	1899.599467	1230.237359
test	0.838858	1927.604685	1246.977878

Figure 5.3.4: Training and testing score for RandomForestRegressor

Based on Figure 5.3.4, we can see that the difference between R² score of the training and testing data set is around 0.03, which indicates a difference of approximately 3% between both data sets. This may suggest that the model is less overfit by ignoring inappropriate noises while

capturing the significant patterns. Moreover, if we inspect on the RMSE and MAE value, the difference between both data sets is not more than 50. This tells us that the model actually performs quite decently. The reason is because a difference of less than 50 is a very minimal error in compared to an error of around 2000 (or 1200 in MAE).



Figure 5.3.5: Actual vs. predicted total sales for Rossmann drugstores in the most recent four weeks using RandomForestRegressor

According to our observations on Figure 5.3.5, the predicted sales using the Random Forest Regressor follows the trend of the actual sales quite well. Our model has successfully predicted all the troughs but did not predict the peaks accurately. A reason may be that our model did not capture the patterns of the maximum values exactly as it was in the actual sales. Despite that, the Random Forest model had overcome overfitting which is seen by the decreased difference in the training and testing score.

Bagging (bootstrap aggregation)

In order to make more accurate predictions on the maximum sales, we decided to build a bagging regressor model. Bagging is the most traditional ensemble learning method to reduce variance in a noisy data set. Basically, bagging works by drawing a few samples from the training data set and build a base learner on these samples (Baskin et al., 2017). The procedure is also known as bootstrapping. Later when all the base learners are well built, we then aggregate the predictions of these models for a final prediction. Bagging is used by our team as it helps to reduce overfit while ensuring that all the individual models are not too similar when being trained on the same data.

In this context, we are choosing the DecisionTreeRegressor as our base model. As what we did in the Random Tree Regression, we must perform hyperparameter tuning first to find the best parameter value to maximize model performance. The codes for hyperparameter tuning are almost similar to the Random Tree Regression as we are still using the RandomizedSearchCV.

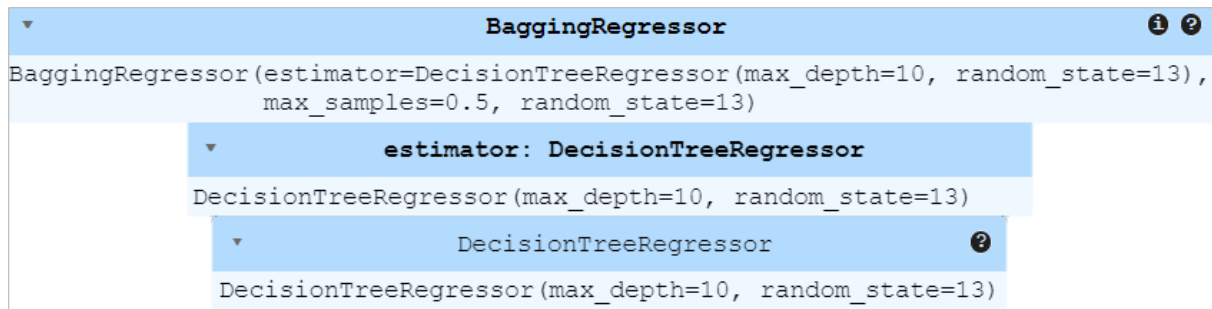


Figure 5.3.6: Output for BaggingRegressor hyperparameter tuning using DecisionTreeRegressor as the base

We can see that the best parameters are maximum depth of 10, maximum bagging samples of 0.5 and number of base estimators (DecisionTreeRegressor) of 50.

Next, using the optimal parameter, we will fit the bagging regressor model using decision tree regressor as the base estimator again. We then call the `evaluate_model` function to check on the model's metric and performance. Note that we still performed cross-validation with 5 folds in the `evaluate_model` function as above.

	R²	RMSE	MAE
train	0.861939	1900.201876	1231.795630
test	0.839208	1925.510839	1247.073879

Figure 5.3.7: Training and testing score for BaggingRegressor

We can see that the training and testing R^2 score difference is still within 3%. This means that the overfitting of the model is within 3% of the actual data set. The result for bagging is actually quite similar to the Random Forest Regression. We believe that because the bagging method which trains many 'trees' and aggregate the predictions together has similar working principle with the Random Forest Regression. Hence, we decided to further refine the model with other ensemble method and stop using the bagging model.

Boosting

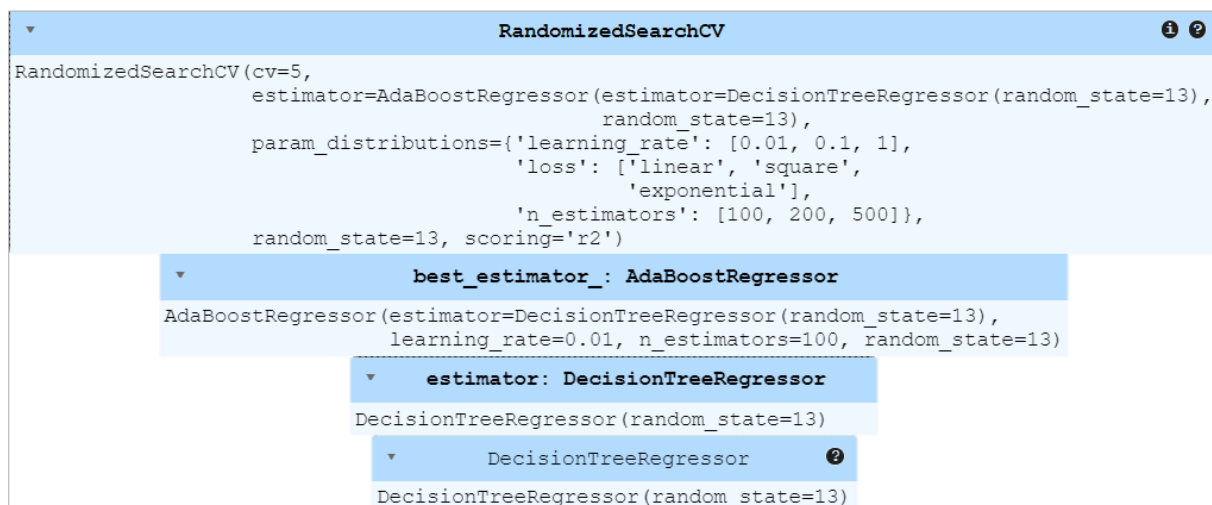
Boosting is also a well-known ensemble method that reduces bias. Our boosting model also works by combining several weak learners (the decision trees) into a strong learner (the boosting model). The key difference between boosting and bagging is boosting trains model sequentially where it modifies the next model every time based on the previous model predictions until it reaches a final prediction. On the other hand, bagging just trains multiple models simultaneously and averages the predictions.

We implemented three types of boosting, which is the Adaptive Boosting (AdaBoost), Gradient Boosting and Extreme gradient boosting (XGBoost).

Adaptive Boosting (AdaBoost)

AdaBoost is the most classic boosting method. According to González et al. (2020), the base learner is trained with certain weightage and the predictions updates the model to be trained again. In other words, our Adaptive Boosting model is trained and modified multiple times successively until the best model is produced.

We have decided to use DecisionTreeRegressor as the base learner again. As usual, we need to perform hyperparameter tuning to get the most optimal parameters for our AdaBoost model. We are using RandomizedSearchCV as in the Bagging model, where an extra parameter ‘loss’ is added. This parameter means how the AdaBoost model updates the model based on weightage after predictions.



```
RandomizedSearchCV(cv=5,
                  estimator=AdaBoostRegressor(estimator=DecisionTreeRegressor(random_state=13),
                                             random_state=13),
                  param_distributions={'learning_rate': [0.01, 0.1, 1],
                                     'loss': ['linear', 'square',
                                             'exponential'],
                                     'n_estimators': [100, 200, 500]},
                  random_state=13, scoring='r2')

best_estimator_: AdaBoostRegressor
AdaBoostRegressor(estimator=DecisionTreeRegressor(random_state=13),
                  learning_rate=0.01, n_estimators=100, random_state=13)

estimator: DecisionTreeRegressor
DecisionTreeRegressor(random_state=13)

DecisionTreeRegressor
DecisionTreeRegressor(random_state=13)
```

Figure 5.3.8: Output for AdaBoostRegressor hyperparameter tuning using DecisionTreeRegressor as the base

Based on Figure 5.3.8, we can see that the best parameter is a learning rate of 0.01, 100 weak learners for the model, and a loss of ‘linear’ where the model is updated based on the linear prediction errors.

Next, we call the function evaluate_model again to test the performance of model. In this function, cross validation is carried out again after fitting of model before the metrics calculation.

	R²	RMSE	MAE
train	0.861033	1906.423430	1238.787319
test	0.840104	1920.135348	1246.718493

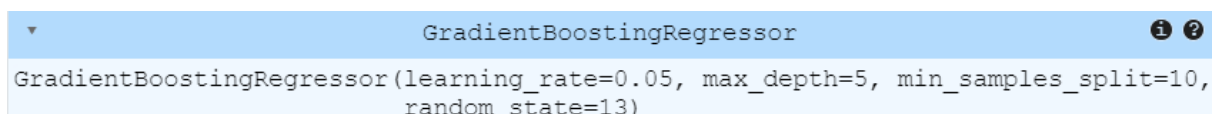
Figure 5.3.9: Training and testing score for AdaBoostRegressor

We can see that the difference between test and train R² score is reduced to approximately 0.02 (Figure 5.3.9). This means that the model is further being refined to reduce overfit compared to the bagging model.

Gradient Boosting

Gradient Boosting has a similar working principle with Adaptive Boosting. The key difference between these two boosting models is Gradient Boosting uses a loss function like mean-squared-error (MSE) to refine the model after every prediction. In contrary, AdaBoost only modifies the model based on simple error calculations such as linear loss as stated above. Moreover, Bahad and Saxena (2019) also stated that Gradient Boosting tends to minimize residuals error while AdaBoost only reweight the sample and model based on single errors. Basically, Gradient Boosting is always more robust than Adaptive Boosting.

Again, we have to perform hyperparameter tuning using RandomizedSearchCV. The Gradient Boosting model is relatively easier to tune because it does not need a base learner. After tuning all the essential parameters, we call the `evaluate_model` function again to fit the model with the tuned parameters and perform cross-validation to make predictions and calculate the metrics.



```
GradientBoostingRegressor(learning_rate=0.05, max_depth=5, min_samples_split=10,
random_state=13)
```

Figure 5.3.10: *GradientBoostingRegressor with the optimal parameters after hyperparameter tuning*

	R²	RMSE	MAE
train	0.860339	1911.178702	1244.316960
test	0.839802	1921.948979	1251.552007

Figure 5.3.11: *Training and testing score for GradientBoostingRegressor*

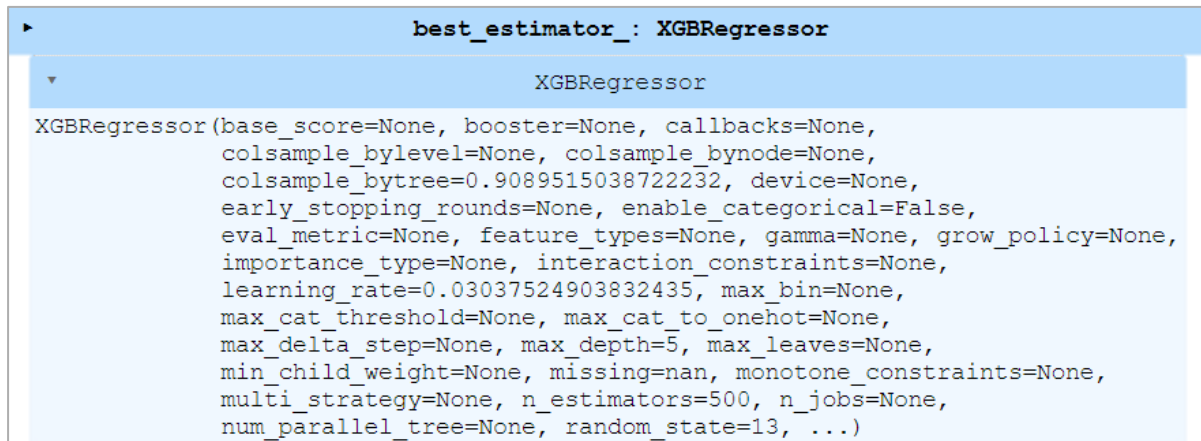
Looking at Figure 5.3.11, we can see that the difference between training and testing R^2 score of the Gradient Boosting model is almost similar to the AdaBoost model. However, the difference of RMSE and MAE value for both data set had decreased to around 10 only. This is a good indicator that tells us that the model actually makes predictions very similar to the actual sales. To further reduce the difference between the training and testing R^2 score, we decided to implement a more enhanced version of the Gradient Boosting model, which is the XGBoost model

Extreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting works almost the same way with the normal Gradient Boosting, just that XGBoost utilizes techniques such as regularisation to control overfit. Also, Bentéjac et al. (2020) asserted that XGBoost will always be the faster algorithm when working with

large data sets. This works very well for our project as we are dealing with large sales data sets where we need a more efficient model.

Furthermore, XGBoost supports more hyperparameter for fine-tuning. A very important parameter is the learning rate which controls the contribution of each tree. As usual, we are using the RandomizedSearchCV for tuning with a 5 fold cross-validation.



```
best_estimator_: XGBRegressor
XGBRegressor(
  base_score=None, booster=None, callbacks=None,
  colsample_bylevel=None, colsample_bynode=None,
  colsample_bytree=0.9089515038722232, device=None,
  early_stopping_rounds=None, enable_categorical=False,
  eval_metric=None, feature_types=None, gamma=None, grow_policy=None,
  importance_type=None, interaction_constraints=None,
  learning_rate=0.03037524903832435, max_bin=None,
  max_cat_threshold=None, max_cat_to_onehot=None,
  max_delta_step=None, max_depth=5, max_leaves=None,
  min_child_weight=None, missing=nan, monotone_constraints=None,
  multi_strategy=None, n_estimators=500, n_jobs=None,
  num_parallel_tree=None, random_state=13, ...)

```

Figure 5.3.12: *XGBRegressor with the optimal parameters after hyperparameter tuning*

Note that we only focus on the learning_rate tuning as XGBRegressor has too many parameters such as missing value imputation parameters which is irrelevant in our case.

	R²	RMSE	MAE
train	0.857752	1928.797622	1243.596359
test	0.837598	1935.123993	1249.317690

Figure 5.3.13: *Training and testing score for XGBRegressor*

Based on Figure 5.3.13, we can see that the difference of R^2 score value between training and testing data set has decreased to less than 0.02. We can then conclude that the overfitting issue is successfully resolved. In addition, the difference of RMSE value has decreased to around 7 while MAE has decreased to around 6 for both data set. This is an excellent indicator that our XGBRegressor model in minimizing the errors between the actual and predicted sales.

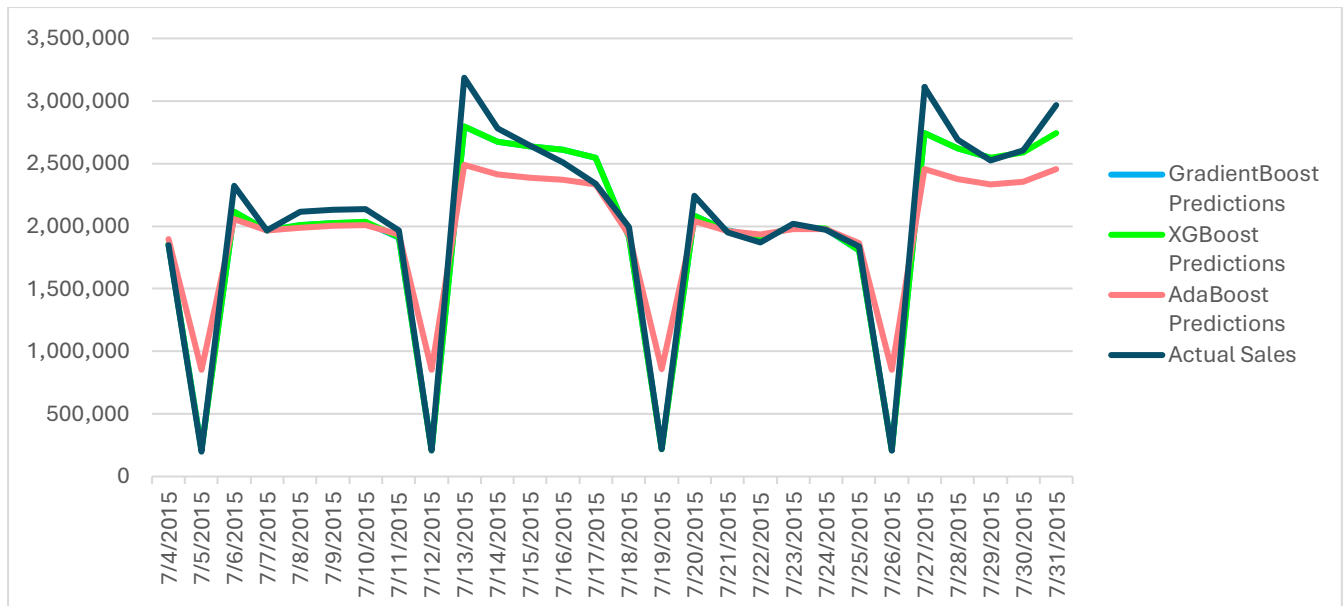


Figure 5.3.14: Actual vs. predicted total sales for Rossmann drugstores in the most recent four weeks for different Boosting model

By observing Figure 5.3.14, we can see that the light green line predictions (XGBoost) overlaps with the actual sales line the best. This shows that the XGBoost model follows the trends of the actual sales and made predictions most accurately. However, we still observe that all of our model still fails to predict the exact peak sales, specifically during 13th of July. Hence, we will move on to another ensemble learning which combines different types of models, the Stacking model.

Stacking

Stacking or stacked generalisation is an ensemble learning technique that combines numerous predictions from the base model (level-0-Model) to obtain a meta-model (level-1-Model). The predictions made from the numerous base models is used as the input for the meta-model.

We decided to use the stacking model as it combines the strength of multiple models we have built previously. The diversity of the meta-model can also rectify the errors of the individual models.

The building and working principle of a stacking model is relatively easy. All we have to do is just stacking a few base models into the StackingRegressor model. The base models we have chosen are RandomForestRegressor, GradientBoostingRegressor and XGBRegressor. This is because these three models performed the best as tested previously.

Additionally, we will not have to perform any hyperparameter tuning anymore because all the base models have been tuned previously. Hence, we can just instantiate the StackingRegressor model with the three base model and perform cross validation before fitting the model. Finally, the stacking model will perform predictions, and we call the function `evaluate_model` again to calculate the model metrics for performance evaluation.

	R^2	RMSE	MAE
train	0.859846	1914.549783	1237.765094
test	0.840016	1920.664010	1242.543162

Figure 5.3.15: Training and testing score for *StackingRegressor*

We can observe that the difference of R^2 score between the training and testing score is minimized to around 0.19 only. This is the best R^2 score difference found across all the models we built. Additionally, both the RMSE and MAE difference is only around 5 for both the data set, also the lowest across all models. This tells us that the model did the best job in terms of minimizing prediction errors. Hence, we can confidently say that the Stacking model is the most robust model in predicting the Rossmann drug store sales.

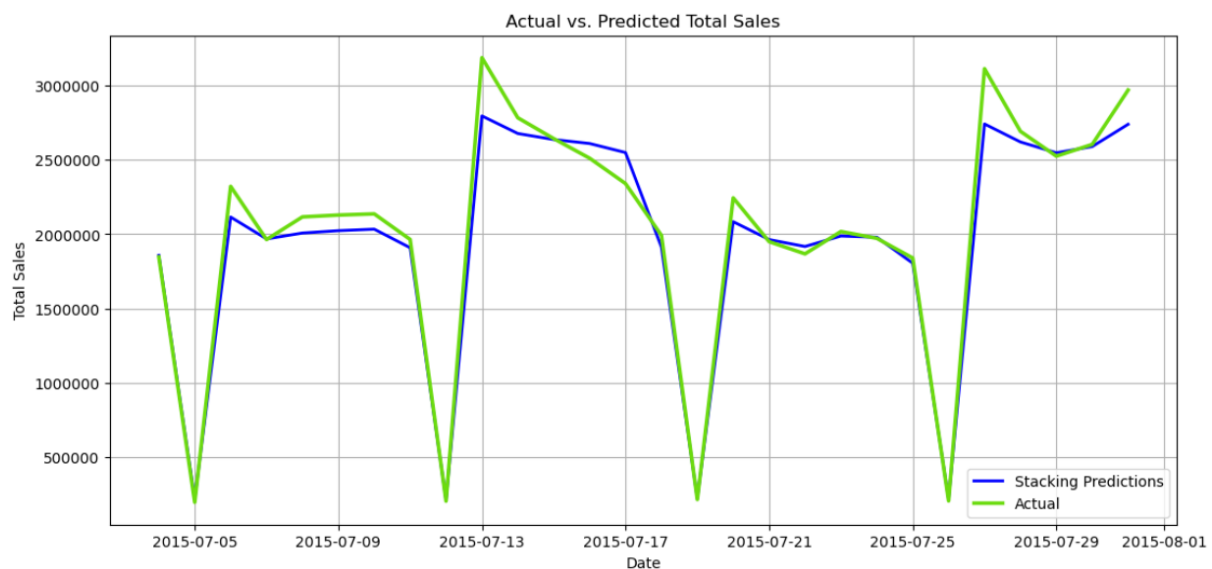


Figure 5.3.16: Actual vs. predicted total sales for Rossmann drugstores in the most recent four weeks for different Stacking model

After we resolved the overfitting issue, we compare again the predicted sales of the Stacking model and the actual sales. Despite we have built a very robust model with a very small root-mean-squared-error and mean-absolute-error, it seems that the model is still unable to capture the peak sales.

A reason may be that ensemble learning and cross validation techniques will always treat the maximum values as ‘abnormal points’ or noise to reduce overfitting. Hence, we can infer that the peak points are actually abnormal sales where some random events that are almost impossible to foresee might have happen. For instance, maybe a tragic event on a particular day that requires a lot of drugs may cause unusually high sales. Due to this reason, we will never predict the peaks exactly same as the actual data.

Conclusion

In this project, our aim was to determine the key factors affecting Rossmann drugstore sales between 2013 to 2015 and hence use existing data to predict the sales four weeks in advance. First, we cleaned, filtered, merged, and did some engineering on the original two datasets provided to obtain a single combined data frame in a format that is efficient for further analysis. Then, we analysed the time-series patterns in the data to find that *Promo* is a key factor of Rossmann sales. Additionally, MLR on our data further revealed the remaining key factors: *Open* and *Customers*. With these selected features, we utilised various decision-tree based algorithms and techniques to build a reliable model to predict the sales four weeks in advance.

From this process, we found that a Random Forest model with Stacking yielded the most accurate sales predictions. The base models used in stacking were RandomForestRegressor, GradientBoostingRegressor and XGBRegressor. This model has a training accuracy of 0.859, and a testing accuracy of 0.840. The model predicts the overall total sales trend over the next four weeks quite well. As expected, random forests coupled with ensemble learning techniques provided us with a model of lower variance, thus less prone to overfitting on the training data. However, because these techniques ignore extreme values to prevent high variance, our final model is unable to predict the exact peaks in the data as accurately.

The predicted overall trend starts off with the highest total sales on Monday, with a gradual decrease until Saturday, then a sudden dip on Sunday where the sales are much lower. This pattern repeats weekly. This is similarly reflected by the actual sales, except the differences between total sales on a daily basis are slightly more significant. The key factors influencing sales of Rossmann stores are *Open*, *Promo*, and *Customers*.

This project may be further improved by collecting and analysing additional data during the timeframe of the collected data. For example, the daily weather, economic conditions, and location of each store may have also played a part in influencing sales. Additionally, the original dataset may have contained outliers that went by undetected; further inspection would be advantageous.

References

- Bahad, P., & Saxena, P. (2019). Study of AdaBoost and Gradient Boosting algorithms for predictive analytics. In G. S. Tomar et al. (Eds.), *International Conference on Intelligent Computing and Smart Communication 2019* (pp. 235–244). Algorithms for intelligent systems. https://doi.org/10.1007/978-981-15-0633-8_22
- Baskin, I. I., Marcou, G., Horvath, D., & Varnek, A. (2017). Bagging and boosting of regression models. In A. Varnek (Eds.), *Tutorials in Chemoinformatics* (pp. 249–255). John Wiley & Sons. <https://doi.org/10.1002/9781119161110.ch16>
- Bentéjac, C., Csörgő, A., & Martínez-Muñoz, G. (2021). A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, 54, 1937–1967. <https://doi.org/10.1007/s10462-020-09896-5>
- Data Headhunters (2024, January 4). *Decision trees vs random forests: comparing predictive power*. <https://dataheadhunters.com/academy/decision-trees-vs-random-forests-comparing-predictive-power/>
- González, S., García S., Del Ser, J., Rokach, L., & Herrera, F. (2020). A practical tutorial on bagging and boosting based ensembles for machine learning: Algorithms, software tools, performance study, practical perspectives and opportunities. *Information Fusion*, 64, 205–237. <https://doi.org/10.1016/j.inffus.2020.07.007>
- Murel, J., & Kavlakoglu, E. (2024, March 18). *What is ensemble learning?* <https://www.ibm.com/topics/ensemble-learning#:~:text=Ensemble%20learning%20is%20a%20machine,than%20a%20single%20model%20alone.>
- Probst, P., Boulesteix, A.-L., & Bischl, B. (2019). Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, 20(53), 1–32. <https://www.jmlr.org/papers/v20/18-444.html>
- Shakedd, A. (2023, December 7). *Department stores are getting smaller (and smarter)*. Chain Store Age. <https://chainstoreage.com/department-stores-are-getting-smaller-and-smarter>
- Talbot, P. (2019, April 23). *Why marketing promotions are losing steam*. Forbes. <https://www.forbes.com/sites/paultalbot/2019/04/23/why-marketing-promotions-are-losing-steam/>

Appendix A

Bagging Model and Hyperparameter tuning

```

from sklearn.ensemble import BaggingRegressor
from sklearn.model_selection import RandomizedSearchCV

# Define the base estimator (DecisionTreeRegressor)
base_estimator = DecisionTreeRegressor(random_state=13)

# Set the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [10, 20, 50], # Number of base estimators (Bagging)
    'max_samples': [0.5, 0.7, 1.0], # Max samples for Bagging
    'max_features': [0.5, 0.7, 1.0],
    'estimator__max_depth': [None, 10, 20] # Max features for Bagging
}

# Initialize the BaggingRegressor with RandomForest as the base estimator
bagging_model = BaggingRegressor(estimator=base_estimator, random_state=13)

# Perform GridSearchCV on BaggingRegressor
grid_search_bagging = RandomizedSearchCV(estimator=bagging_model, param_distributions=param_grid, cv=3, scoring='neg_mean_squared_error', n_jobs=-1)

# Fit the model with the training data
grid_search_bagging.fit(X_train, y_train)

```

Figure A1: *Code for bagging model hyperparameter tuning*

```

bagging_model = BaggingRegressor(estimator=DecisionTreeRegressor(max_depth=10,random_state=13),
                                n_estimators=10,max_samples=0.5,max_features=1.0, random_state=13)
evaluate_model(bagging_model)

```

Figure A2: *Code for bagging model*

Appendix B

Boosting Model and Hyperparameter tuning

```

from sklearn.ensemble import AdaBoostRegressor
from sklearn.model_selection import RandomizedSearchCV

# Define the base estimator
base_rf = DecisionTreeRegressor(random_state=13)

# Define the AdaBoost model using DecisionTree as the base estimator
adaboost_model = AdaBoostRegressor(estimator=base_rf, random_state=13)

# Define the parameter grid for tuning
param_grid_adaboost = {
    'n_estimators': [100, 200, 500], # This is for AdaBoost, number of weak learners
    'learning_rate': [0.01, 0.1, 1], # AdaBoost Learning rate
    'loss': ['linear', 'square', 'exponential'] # Loss function for AdaBoost
}

# Perform RandomizedSearchCV
grid_search_adaboost = RandomizedSearchCV(estimator=adaboost_model, param_distributions=param_grid_adaboost, cv=5,
                                          scoring='r2', n_iter=10, random_state=13)

# Fit the grid search model
grid_search_adaboost.fit(X_train, y_train)

```

Figure B1: Code for AdaBoostRegressor model hyperparameter tuning

```

adaboost_model = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=10, random_state=13),
                                   n_estimators=100, loss='linear', learning_rate=0.01, random_state=13)
evaluate_model(adaboost_model)

```

Figure B2: Code for AdaBoostRegressor model

```

from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import RandomizedSearchCV

# Define the parameter grid for tuning
param_grid_gbr = {
    'n_estimators': [100, 500, 1000],
    'learning_rate': [0.01, 0.1, 0.05],
    'max_depth': [3, 5, 7],
    'min_samples_split': [2, 5, 10]
}

# Perform grid search
grid_search_gbr = RandomizedSearchCV(estimator=gbr_model, param_distributions=param_grid_gbr, cv=5, scoring='r2')
grid_search_gbr.fit(X_train, y_train)

```

Figure B3: Code for GradientBoostingRegressor model hyperparameter tuning

```

gbr_model = GradientBoostingRegressor(n_estimators=100, min_samples_split=10, max_depth=5, learning_rate=0.05, random_state=13)
evaluate_model(gbr_model)

```

Figure B4: Code for GradientBoostingRegressor model

```

from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform
from xgboost import XGBRegressor

# Randomized search for XGBRegressor
param_dist_xgb = {
    'n_estimators': [100, 500, 1000],
    'learning_rate': uniform(0.01, 0.3),
    'max_depth': [3, 5, 7],
    'subsample': uniform(0.6, 0.4),
    'colsample_bytree': uniform(0.6, 0.4)
}

random_search_xgb = RandomizedSearchCV(estimator=xgb_model, param_distributions=param_dist_xgb,
                                       n_iter=50, cv=5, scoring='r2', random_state=13)

random_search_xgb.fit(X_train, y_train)

```

Figure B5: *Code for XGBRegressor model hyperparameter tuning*

```

xgb_model = XGBRegressor(learning_rate = 0.03037524903832435, max_depth = 5, n_estimators = 500, random_state=13)
evaluate_model(xgb_model)

```

Figure B6: *Code for XGBRegressor model*

Appendix C

Stacking Model

```

from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import RidgeCV

# Define the base model to be stacked
estimators = [
    ('rf', GradientBoostingRegressor(n_estimators=300, min_samples_split=20, max_depth=4, learning_rate=0.01, random_state=13)),
    ('rfr', RandomForestRegressor(n_estimators=1000, min_samples_split = 10,
                                min_samples_leaf = 10, max_depth = 10
                                ,max_features='sqrt',random_state=13)),
    ('xgb', XGBRegressor(learning_rate = 0.03037524903832435,max_depth = 5,n_estimators = 500, random_state=13))
]

# Stacking regressor with regularized meta-model
stacking_model = StackingRegressor(estimators=estimators, final_estimator=RidgeCV(alphas=[0.1, 1.0, 10.0, 100.0], cv=5))
evaluate_model(stacking_model)

```

Figure C1: *Code for StackingRegressor model*