Individual Project:
Demand Forecasting for A Fast-Food Restaurant Chain

Logistics and Supply Chain Analytics

Persa Marathefti

11/15/2021

Contents

1 Introduction

Forecasting is a fundamental activity for many companies as it enables them to create data-backed planning strategies. For most supply chains, accurate demand forecasting leads to successful inventory management, which, in turn, could result in higher profits. Demand forecasting is vital for companies operating in the F&B sector, such as restaurants, where customer satisfaction is a priority. Therefore, ensuring that customer orders can be fulfilled by keeping ingredients in stock is critical. In addition, perishable goods, such as vegetables, require daily restocking as they must be fresh. This category could benefit from daily demand forecasting that could result in considerable savings.
The purpose of this project is to forecast the daily lettuce demand for two weeks (06/16/2015 to 06/29/2015) for a fast-food restaurant in the US.

2 Data Preprocessing

This section displays the data preprocessing steps to generate a time-series of the daily lettuce demand for the restaurant with StoreID: 44673.

2.1 Loading the data

The provided dataset contains transactional data for purchased menu items from 3/5/2015 to 6/15/2015, recipe lists, ingredient lists, and metadata for four fast-food restaurants in the US.

The ten provided tables are displayed below.

```
# loading the data
pos_ordersale                 <- read.csv("pos_ordersale.csv") # not needed
menuitem                      <- read.csv("menuitem.csv")
menu_items                    <- read.csv("menu_items.csv")
recipes                       <- read.csv("recipes.csv")
sub_recipes                   <- read.csv("sub_recipes.csv")
ingredients                   <- read.csv("ingredients.csv")
portion_uom_types             <- read.csv("portion_uom_types.csv")
recipe_ingredient_assignments <- read.csv("recipe_ingredient_assignments.csv")
sub_recipe_ingr_assignments   <- read.csv("sub_recipe_ingr_assignments.csv")
recipe_sub_recipe_assignments <- read.csv("recipe_sub_recipe_assignments.csv")
```

For this project, I did not load the store_restaurant table with specific information for each restaurant as the storeID is available in other tables of interest. In addition, I decided not to use the pos_ordersale table because transaction information data were already reflected in the menuitem table.

2.2 Data Wrangling

The next step was to explore the loaded tables and decide how to join them to obtain the final time series. This process is displayed below.

To begin with, I renamed the column in the menuitem table to "MenuItemId" so that it matched the name of the column in the menu_items table. This was a necessary step to enable me to join the two tables, as shown in the next steps. Next, the menuitem table was filtered to include only transaction information for the store of interest (StoreID: 46673).

```
# rename menuitem column "Id" to "MenuItemId" to match column in menu_items
menuitem <- menuitem %>%
            rename(MenuItemId = Id)
# filter menuitem for store of interest - Store ID:46673
menuitems_46673 <-  menuitem %>%
                    select(date, MD5KEY_MENUITEM, MD5KEY_ORDERSALE, StoreNumber,
                           Quantity, PLU, MenuItemId) %>% # keep only columns with useful information
                    filter(StoreNumber == 46673)
```

The first table join was between menuitem_46673 and menu_items on the PLU and MenuItemId columns, followed by a join with the recipes table on RecipeId. The resulting dataframe contained daily transaction information for store 46673, along with order quantities and a recipe identifier.

```
# table joins

# join: menuitems_46673, menu_items, recipes
menu_recipes_46673 <- left_join(menuitems_46673, menu_items, by = c("PLU", "MenuItemId")) %>%
    left_join(y = recipes, by = "RecipeId") %>%
    select(date, MD5KEY_MENUITEM, MD5KEY_ORDERSALE, Quantity, PLU, RecipeId)
```

This table was then used to create a new aggregated table that displayed the total number of orders of a specific recipe on each day (Recipe_Quantity). To achieve this I used the group_by function, and specified the group by columns to be date and RecipeId.

```
# group by columns: date, RecipeId

# aggregate: sum of Quantity
menu_tot_recipes_per_day_46673 <- menu_recipes_46673 %>%
    group_by(date, RecipeId) %>%
    summarise(Recipe_Quantity = sum(Quantity))
```

The first 5 observations in this table are displayed below.

```
kable(menu_tot_recipes_per_day_46673[1:5, ])
```

date	RecipeId	Recipe_Quantity
15-03-05	6	2
15-03-05	7	4
15-03-05	10	1
15-03-05	12	7
15-03-05	15	1

Next, I wanted to create a table that contained all the recipes that listed lettuce as an ingredient. To do so, I started by retrieving the unit of measurement for lettuce by joining the ingredient table with the portion_uom_types table and filtering the result for items that contained the word 'Lettuce'.

```
# join: ingredients, portion_uom

# filter: ingredient of interest -> Lettuce
lettuce_uom <- left_join(ingredients, portion_uom_types, by = "PortionUOMTypeId") %>%
    filter(IngredientName %like% "Lettuce") %>%
    select(IngredientId, IngredientName, PortionTypeDescription)
```

The table lettuce_uom is displayed below. As can be seen, two ingredients contain the word lettuce in their name, Lettuce and Lettuce - Metric measured in ounces and grams, respectively.

IngredientId	IngredientName	PortionTypeDescription
27	Lettuce	Ounce
291	Lettuce - Metric	Gram

The lettuce_uom table was then joined with the recipe_ingredient_assingments to match the two lettuce

ingredients with all the recipes that contain them.

```
# join: lettuce_uom,recipe_ingredient_assignments
ingr_assign_rec_lettuce <- left_join(lettuce_uom, recipe_ingredient_assignments,
    by = "IngredientId")
```

The same process was repeated for joining lettuce_uom to sub_recipe_ingr_assignment. However, this table had to be connected further with the tables sub_recipes and recipe_sub_recipe_assignments so that the final dateframe would also match ingredients to recipes not just sub_recipes. Please note that the column Quantity in the resulting table was the product of Quantity from sub_recipe_ingr_assingments and Factor from recipe_sub_recipe_assignments, which captures the quantity of a specific sub recipe used in a recipe. It represents the total quantity of lettuce needed in a recipe. The selected columns of this table were picked to match the columns in ingr_assign_rec_lettuce.

```
# join: lettuce_uom, sub_recipe_ingr_assignments, sub_recipes,
# recipe_sub_recipe_assignments
ingr_assign_sub_rec_lettuce <- left_join(lettuce_uom, sub_recipe_ingr_assignments,
    by = "IngredientId") %>%
    left_join(sub_recipes, by = "SubRecipeId") %>%
    left_join(recipe_sub_recipe_assignments, by = "SubRecipeId") %>%
    mutate(Quantity = Quantity * Factor) %>%
    select(IngredientId, IngredientName, PortionTypeDescription, RecipeId, Quantity)
```

The two tables ingr_assign_rec_lettuce and ingr_assing_sub_rec_lettuce were then combined to form one table.

```
ingr_assignments <- bind_rows(ingr_assign_rec_lettuce, ingr_assign_sub_rec_lettuce)
```

The resulting table displays the ingredient along with their unit of measurement, the recipe that they are used in and how much quantity is needed in each recipe.

IngredientId	IngredientName	PortionTypeDescription	RecipeId	Quantity
27	Lettuce	Ounce	687	0
27	Lettuce	Ounce	688	0
27	Lettuce	Ounce	689	0
27	Lettuce	Ounce	690	0
27	Lettuce	Ounce	959	12

The final step was to join the tables menu_tot_recipes_per_day_46673 and ingr_assignments and aggregate the result by date to obtain values for the daily lettuce demand. After joining the two tables I wanted to check whether any unit of measurement conversions were needed, however, in the recipes ordered for restaurant 46673 only the ingredient "Lettuce" was used that was measured in ounces. Hence, no conversions needed.

```
# join: menu_tot_recipes_per_day_46673, ingr_assignments
ingr_assignments_menu_tot_recipes_per_day_46673 <- inner_join(menu_tot_recipes_per_day_46673,
    ingr_assignments, by = "RecipeId")
# all lettuce in Ounces -> no conversions required
unique(ingr_assignments_menu_tot_recipes_per_day_46673$PortionTypeDescription)
```

```
## [1] "Ounce      "
```

The ingr_assignments_menu_tot_recipes_per_day_46673 was modified to include an extra column that displayed the total lettuce quantity used for each recipe on each day. It is calculated as the product of total times a recipe was ordered on a specific date (Recipe_Quantity) and the quantity of lettuce required in a specific recipe (Quantity). This table was then aggregated by the date column to give the sum of total quantity of lettuce used per day for the period 03/05/2015 to 06/15/2015.

```
# new columns: total_quantity = Recipe_Quantity*Quantity

# group_by column: date

# aggregate: sum of total_quantity
lettuce_ts_df <- ingr_assignments_menu_tot_recipes_per_day_46673 %>%
    mutate(total_quantity = Recipe_Quantity * Quantity) %>%
    group_by(date) %>%
    summarise(store_46673_ounces = sum(total_quantity))

# convert the date column to Date class
lettuce_ts_df$date <- as.Date(lettuce_ts_df$date, format = "%y-%m-%d")
# set date column as index
lettuce_ts_df <- lettuce_ts_df %>%
    column_to_rownames(., var = "date")
```

The first 5 observations of the final df are displayed below.

	store_46673_ounces
2015-03-05	152
2015-03-06	100
2015-03-07	54
2015-03-08	199
2015-03-09	166
2015-03-10	143

The following part verifies that the dataframe contains observations for each date between '2015-03-05' and '2015-06-15'.

```
# check that we have a complete df
sum(is.na(lettuce_ts_df$store_46673_ounces))
```

```
## [1] 0
```

```
time.check = seq(as.Date("2015-03-05"), as.Date("2015-06-15"), by = "day")
# expected number of days between the two dates
length(time.check) == dim(lettuce_ts_df)[1]
```

```
## [1] TRUE
```

2.3 Time Series Generation

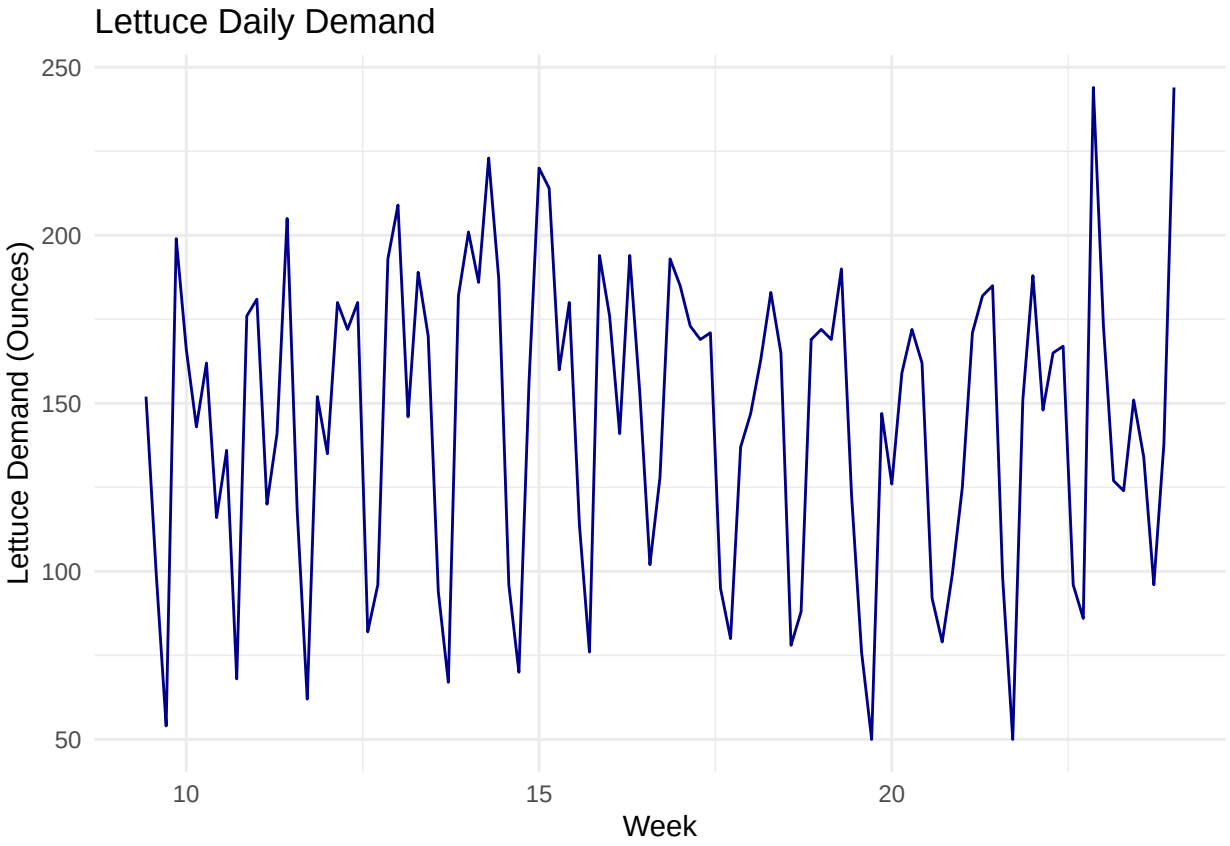
The final dataframe was converted to a time series object with weekly frequency = 7, as we have daily data and start set to c(startW, startD), where StartW: number of the week of the first date in the dataframe,

and
StartD: weekday number of the the first date in the dataframe (Sunday = 0)
The first date in the dataframe, "2015-03-05" was in the 9th week of that year, StartW = 9, and a Thursday, StartD = 4.

```
# find the week and day of the week of the first observation
startW <- as.numeric(strftime("2015-03-05", format = "%W"))
startD <- as.numeric(strftime("2015-03-05", format = " %w"))

# time series creation
ts <- ts(lettuce_ts_df[, 1], start = c(startW, startD), frequency = 7)
```

The following graph depicts the time series. As can be seen, there seems to be a weekly seasonal pattern, but no clear trend.



2.4 Train-Test Split

Before moving on to the application of different models to obtain a forecast, I split the time series into two sets, a train set (70%) and a test set (30%). To obtain the exact date on which the ts should be split, I calculated the number of observations out of 103 that would account for 70% of the total observations (`103*70/100`), and used the website link to get the date that falls 72 days away from 03-05-2015. The date was calculated as 05-15-2015. Then, I used the same method that was described in the previous section to fill in the end parameter of the window function. ts.test was created by setting start to one day after the endD used for obtaining ts.train.

```r
# find the week and day of the week for the date 2015-05-16
endW <- as.numeric(strftime("2015-05-15", format = "%W"))
endD <- as.numeric(strftime("2015-05-15", format = " %w"))

# split dataset into test and train - 70:30
ts.train <- window(ts, end = c(endW, endD))
ts.test <- window(ts, start = c(endW, endD + 1))

length(ts.test)/(length(ts.train) + length(ts.test))
```

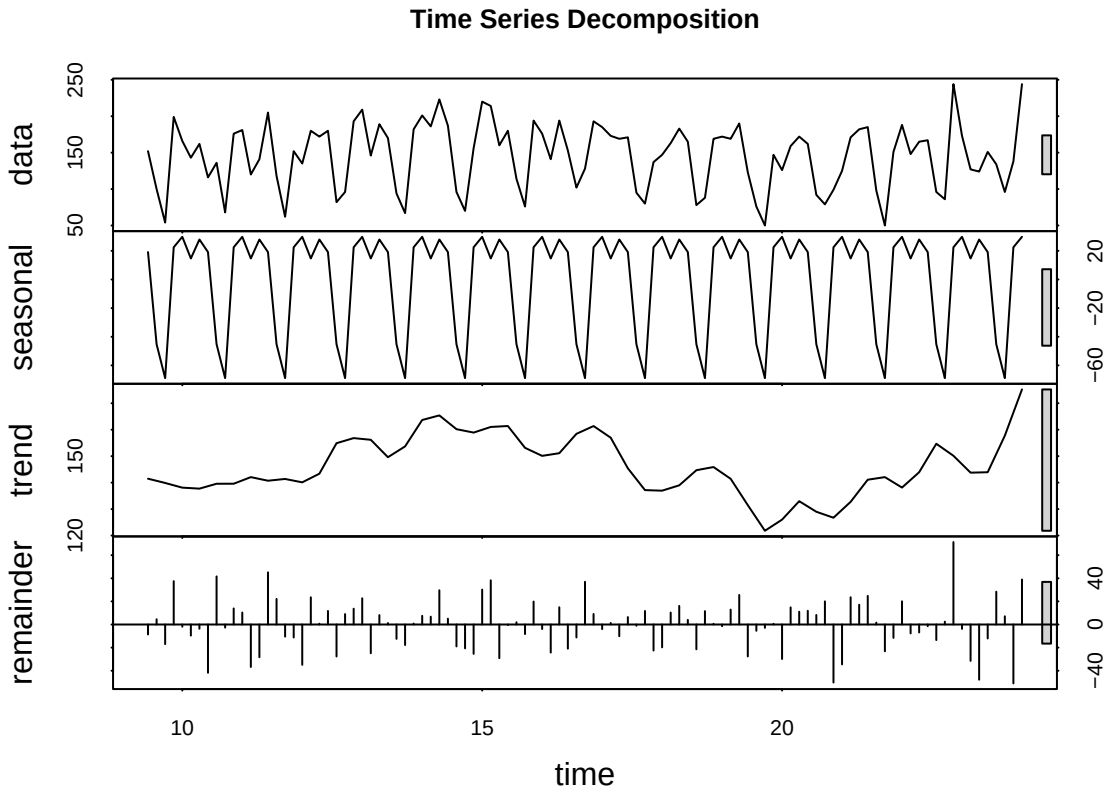
```
## [1] 0.3009709
```

3 Holt-Winters model

In this section, I fit a Holt-Winters model to the time series. Holt-Winters is a brute force method used when a time series displays trends or seasonality. From the initial plot of the time series, it was clear that a seasonal pattern is present. Thus, this model is an appropriate one to base the forecast of daily lettuce demand on.

At first, to confirm that seasonality is present, I used the stl() function to decompose the time series. The decomposition of the ts time series into trend, seasonal, and remainder factors is displayed below.

```r
# ts decomposition
plot(stats::stl(ts, s.window = "periodic"), main = "Time Series Decomposition")
```



The length of the gray bar on the right-hand side is an indicator of the importance of each factor. The

longer the bar, the less important a component is. Hence, one can observe that the trend factor is not significant. On the other hand, the seasonal factor is a great contributor to the variation of the time series and displays an additive pattern.

Based on the time series decomposition, the model should be of the form "ANA" as:
- A: it is typical for the error type to be additive
- N: no trend was observed in the time series decomposition
- A: an additive seasonal component was observed in the time series decomposition

Using the function ets() with model ="ZZZ", I fit a Holt-Winters model to ts.train. The fitted model is displayed below.

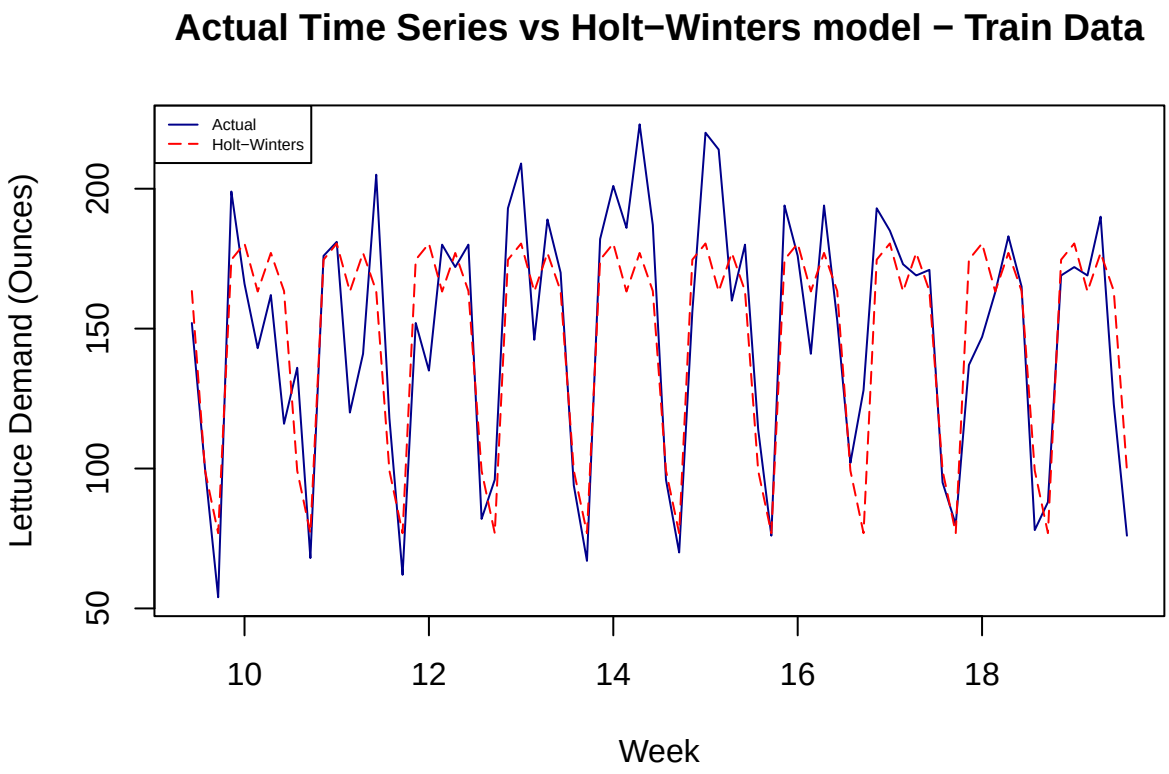
```
# Holt-Winters model
ts.ets <- ets(ts.train, model = "ZZZ")  # ETS(A,N,A)
ts.ets
```

```
## ETS(A,N,A)
##
## Call:
##  ets(y = ts.train, model = "ZZZ")
##
##   Smoothing parameters:
##     alpha = 1e-04
##     gamma = 1e-04
##
##   Initial states:
##     l = 147.8491
##     s = 29.2015 15.4243 32.5774 26.772 -70.9483 -48.6118
##           15.5849
##
##   sigma:  23.8919
##
##      AIC     AICc      BIC
## 775.2956 778.9022 798.0623
```

As the function has returned a model with the expected form **ETS(A,N,A)**, the model specification is proper and fitting another Holt-Winters model is not necessary.

The below plot compares the fitted ts.ets on the train data with the actual time series (ts.train). The Holt-Winters model follows a similar seasonal pattern as the original time series.

```
# plot the actual time series against the Holt-Winters model time series
plot(ts.train, main = "Actual Time Series vs Holt-Winters model - Train Data", xlab = "Week",
    ylab = "Lettuce Demand (Ounces)", lty = 1, col = "dark blue")
lines(fitted(ts.ets), col = "red", lty = 5)
legend("topleft", legend = c("Actual", "Holt-Winters"), col = c("dark blue", "red"),
    box.lty = 1, lty = c(1, 5), cex = 0.5)
```



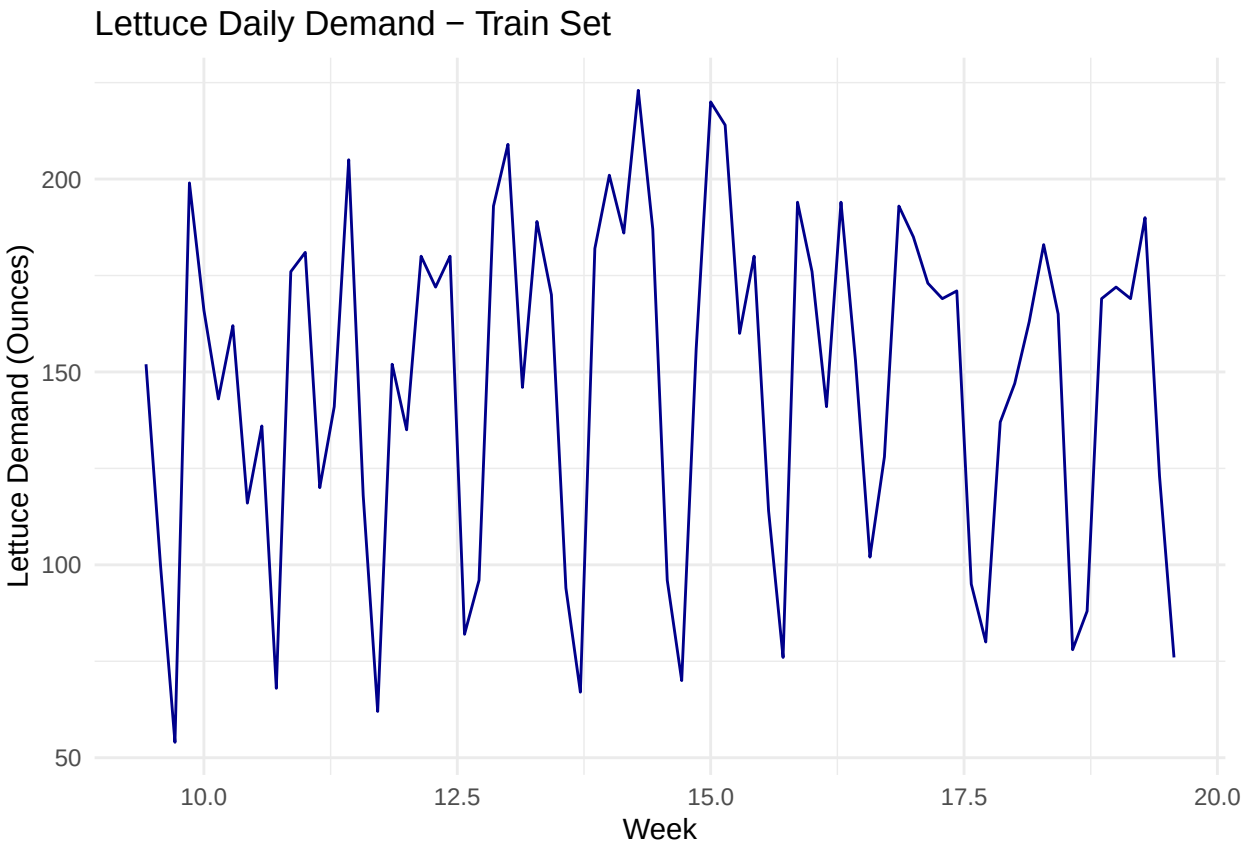
4 ARIMA model

In this section, I fit an ARIMA model to the time series using the Box-Jenkins procedure. ARIMA is another popular time series method for forecasting, that is based on the autocorrelations in the data.

4.1 Identification

The first step in the Box-Jenkins procedure is to visualise the time series and check that it is stationary. If it is not, then data pre-processing may needed to make it stationary before an ARMA model can be fitted. As mentioned when fitting the Holt-Winters model, the plot of the time series indicates that there is seasonality, but no trends.

```
# plot of the daily lettuce demand - train set
autoplot(ts.train, ts.colour = "dark blue")+
   ggtitle("Lettuce Daily Demand - Train Set") +
  xlab("Week") +
  ylab("Lettuce Demand (Ounces)")+
  theme_minimal()
```



A time series with seasonality or trend is not stationary, therefore some pre-processing is required before fitting an ARMA model.
To verify that the initial time series is trend stationary, I run the ADF, PP and KPSS tests, and ndiffs(). For a 5% significance level, the small p-values in the ADF and PP tests, indicate that the null hypothesis that the time series is non-stationary is rejected. The same conclusion is obtained when looking at the KPSS test, which has a high p-value, an indication that the null hypothesis that the time series is stationary is not rejected. These results are in line with the previous observations as the time series does not display any trends. The same is confirmed by the ndiffs() function.

```
# stationarity tests no trends observed
adf.test(ts.train)
```

```
##
##  Augmented Dickey-Fuller Test
##
## data:  ts.train
## Dickey-Fuller = -7.5767, Lag order = 4, p-value = 0.01
## alternative hypothesis: stationary
```

```
pp.test(ts.train)
```

```
##
##  Phillips-Perron Unit Root Test
##
## data:  ts.train
## Dickey-Fuller Z(alpha) = -44.63, Truncation lag parameter = 3, p-value
```

```
## = 0.01
## alternative hypothesis: stationary
```

```
kpss.test(ts.train)
```

```
##
##  KPSS Test for Level Stationarity
##
## data:  ts.train
## KPSS Level = 0.11095, Truncation lag parameter = 3, p-value = 0.1
```

The same is confirmed by the ndiffs() function.

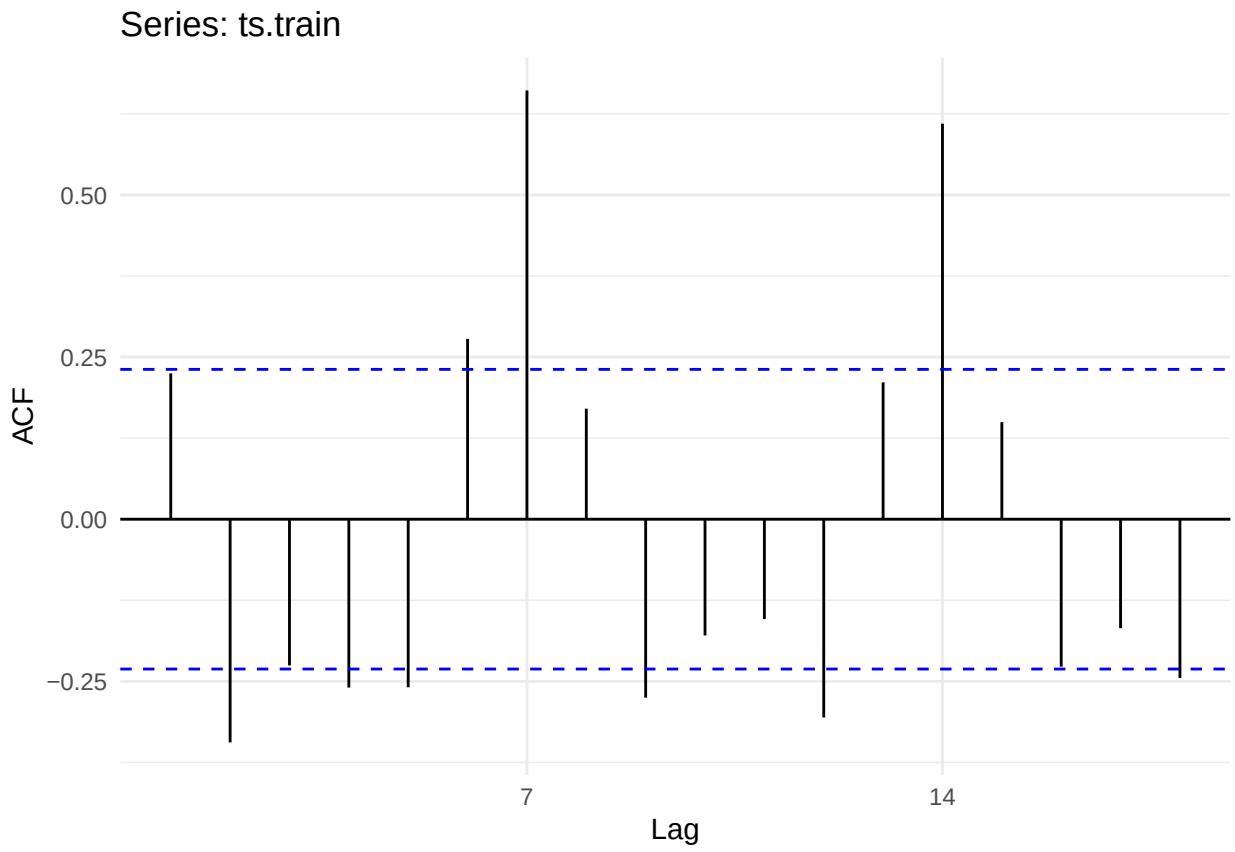
```
# no differencing needed to remove trends
ndiffs(ts.train)
```

```
## [1] 0
```

Next, to check for seasonality, I plotted the ACF and PACF, and run the function nsdiffs() that estimates the number of differences needed to remove the variation from the seasonal components.

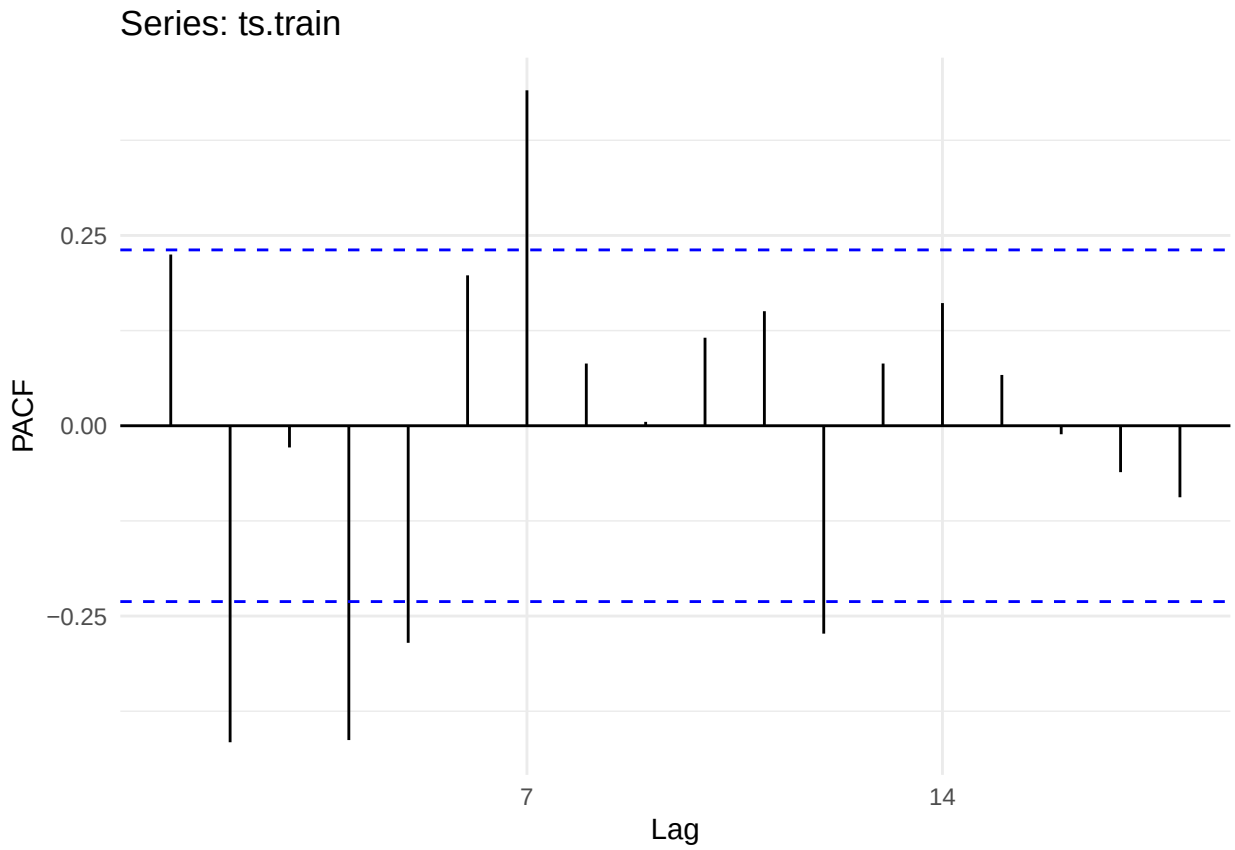
The ACF plot shows a spike at lag=7, that verifies the existence of a seasonal MA component.

```
# ACF plot
ggAcf(ts.train) + theme_minimal()
```



The PACF, further confirms this as there is a decay pattern at at lag = 7.

```
# PACF plot
ggPacf(ts.train) + theme_minimal()
```



The nsdiffs() function suggests that differencing once is enough to remove the seasonal variation.

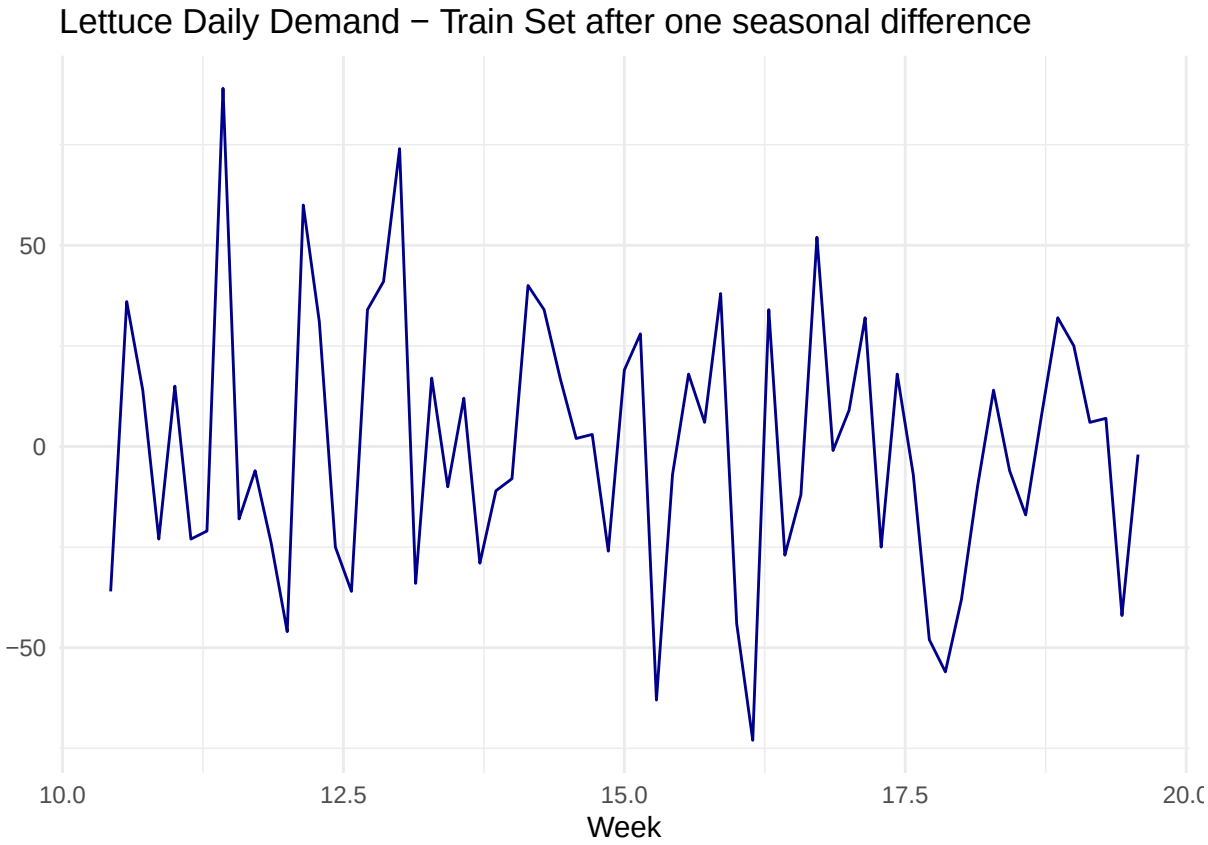
```
nsdiffs(ts.train)
```

```
## [1] 1
```

The plot of the time series after one seasonal difference does not display any clear trends or seasonality.

```
# stationarity tests after seasonal difference
ts.train.diff <- diff(ts.train, differences = 1, lag=7)

autoplot(ts.train.diff, ts.colour = "dark blue") +
  ggtitle("Lettuce Daily Demand - Train Set after one seasonal difference") +
  xlab("Week") +
  theme_minimal()
```



The above observations suggest that the ARIMA model should have the following form:
- non-seasonal part: no trends observed => (0,0,0)
- seasonal part: spikes at lag = 7 in the ACF (MA(1)), decaying PACF, one seasonal difference => (0,1,1)[7]

Final model: **ARIMA(0,0,0)X(0,1,1)[7]**

The results of the auto.arima() function on ts.train are displayed below:

```
# choose optimal p and q based on information criteria
auto.arima(ts.train, trace = TRUE, ic = "bic")
```

```
##
##  ARIMA(2,0,2)(1,1,1)[7] with drift         : 640.7863
##  ARIMA(0,0,0)(0,1,0)[7] with drift         : 645.5713
##  ARIMA(1,0,0)(1,1,0)[7] with drift         : 634.9045
##  ARIMA(0,0,1)(0,1,1)[7] with drift         : 626.6436
##  ARIMA(0,0,0)(0,1,0)[7]                    : 641.3987
##  ARIMA(0,0,1)(0,1,0)[7] with drift         : 649.7311
##  ARIMA(0,0,1)(1,1,1)[7] with drift         : 630.8179
##  ARIMA(0,0,1)(0,1,2)[7] with drift         : 630.8179
##  ARIMA(0,0,1)(1,1,0)[7] with drift         : 634.8238
##  ARIMA(0,0,1)(1,1,2)[7] with drift         : Inf
##  ARIMA(0,0,0)(0,1,1)[7] with drift         : 623.2872
##  ARIMA(0,0,0)(1,1,1)[7] with drift         : 627.3567
##  ARIMA(0,0,0)(0,1,2)[7] with drift         : 627.3686
##  ARIMA(0,0,0)(1,1,0)[7] with drift         : 631.0858
##  ARIMA(0,0,0)(1,1,2)[7] with drift         : Inf
##  ARIMA(1,0,0)(0,1,1)[7] with drift         : 626.7575
```

```
##  ARIMA(1,0,1)(0,1,1)[7] with drift         : 629.4884
##  ARIMA(0,0,0)(0,1,1)[7]                    : 619.2069
##  ARIMA(0,0,0)(1,1,1)[7]                    : 623.2675
##  ARIMA(0,0,0)(0,1,2)[7]                    : 623.2816
##  ARIMA(0,0,0)(1,1,0)[7]                    : 626.9188
##  ARIMA(0,0,0)(1,1,2)[7]                    : Inf
##  ARIMA(1,0,0)(0,1,1)[7]                    : 622.6551
##  ARIMA(0,0,1)(0,1,1)[7]                    : 622.5429
##  ARIMA(1,0,1)(0,1,1)[7]                    : 625.4192
##
##  Best model: ARIMA(0,0,0)(0,1,1)[7]
```

```
## Series: ts.train
## ARIMA(0,0,0)(0,1,1)[7]
##
## Coefficients:
##          sma1
##       -0.8184
## s.e.   0.1882
##
## sigma^2 estimated as 637.7:  log likelihood=-305.43
## AIC=614.86   AICc=615.05   BIC=619.21
```

The automatic function has chosen the same model as the one described above.

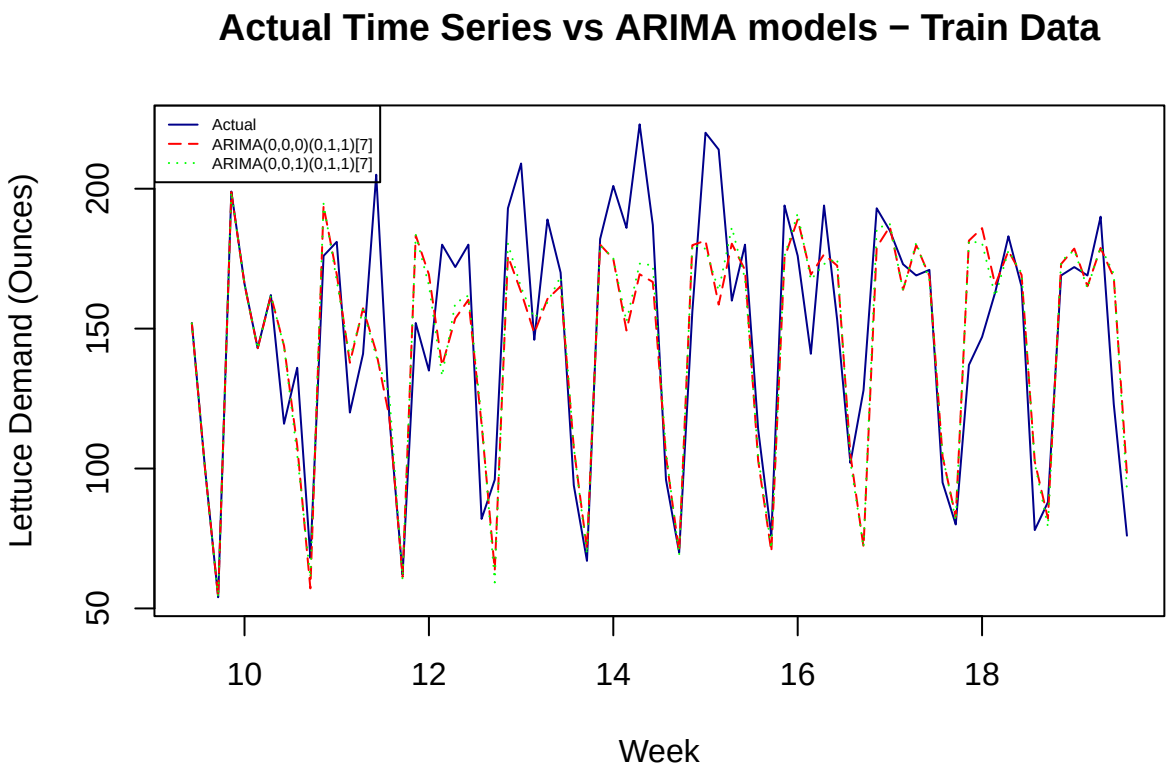
4.2 Estimation

The next step, is to estimate the ARIMA model using maximum likelihood estimators, this is done by the Arima() function. Based on the BIC values of the models obtained from the auto.arima() function, I selected two models with the lowest BIC.
1. ARIMA(0,0,0)(0,1,1)[7]
2. ARIMA(0,0,1)(0,1,1)[7]

```
# two candidate models: ARIMA(0,0,0)(0,1,1)[7] , ARIMA(0,0,1)(0,1,1)[7]
ts.m1 <- Arima(ts.train, order = c(0, 0, 0), seasonal = list(order = c(0, 1, 1),
    period = 7))
ts.m2 <- Arima(ts.train, order = c(0, 0, 1), seasonal = list(order = c(0, 1, 1),
    period = 7))
```

The below plot shows the two ARIMA models against the original time series. The two fitted models are quite similar.

```
# plot the actual time series against the ARIMA modelS time series
plot(ts.train, main = "Actual Time Series vs ARIMA models - Train Data", xlab = "Week",
    ylab = "Lettuce Demand (Ounces)", lty = 1, col = "dark blue")
lines(fitted(ts.m1), col = "red", lty = 5)
lines(fitted(ts.m2), col = "green", lty = 3)
legend("topleft", legend = c("Actual", "ARIMA(0,0,0)(0,1,1)[7]", "ARIMA(0,0,1)(0,1,1)[7]"),
    col = c("dark blue", "red", "green"), box.lty = 1, lty = c(1, 5, 3), cex = 0.5)
```



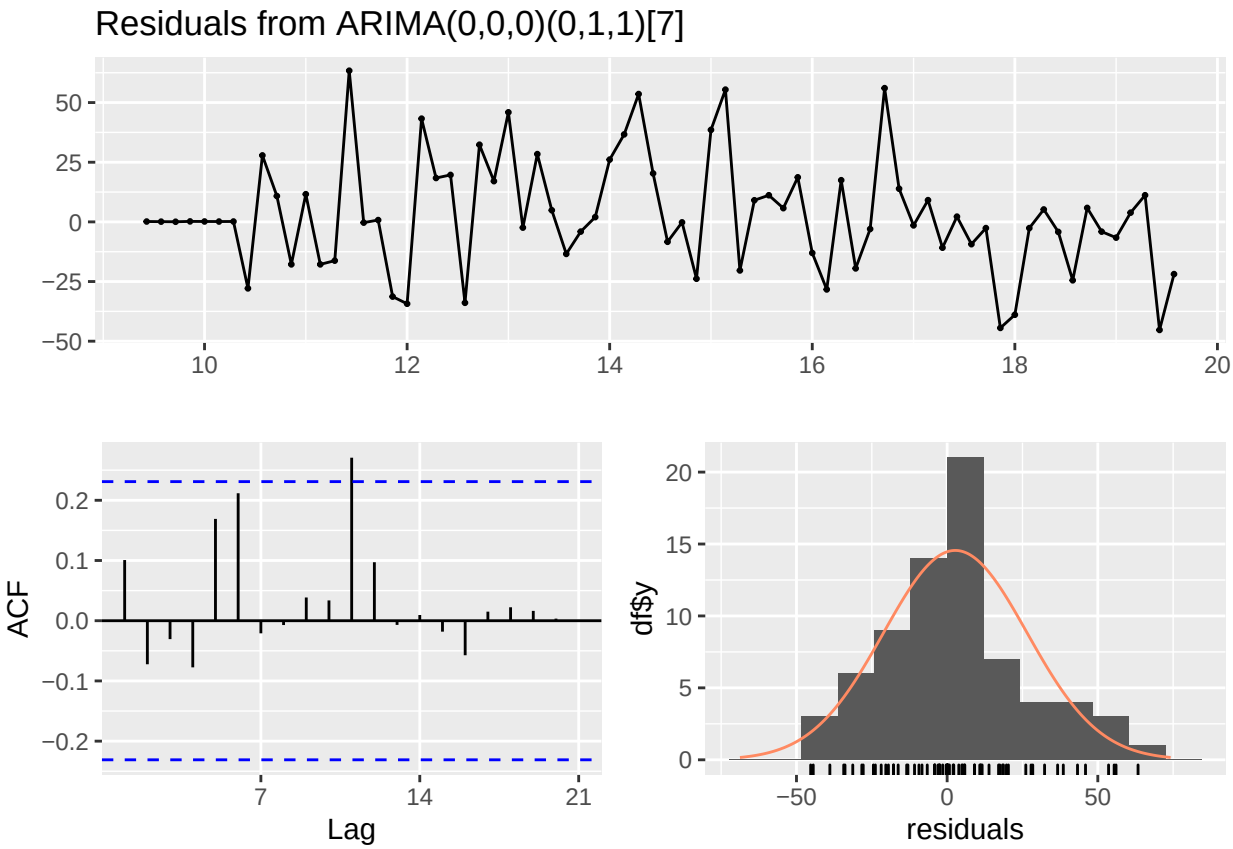
4.3 Verification

The last step is to check how well the model fits the data using residual analysis. The checkresiduals() function results are displayed below for the two selected ARIMA models.

4.3.1 ARIMA(0,0,0)(0,1,1)[7]

The residuals graph shows that the residuals are normally distributed (zero mean and constant variance) and the ACF plot indicates that they behave like white noise (autocorrelations within limits). There are a few observations outside the bounds, however, that is expected due to the small number of observations in the sample.
The results of the Ljung-Box test are also displayed below the charts. The large p-value of the Ljung-Box test confirms that there are no inter-temporal correlations in the residuals.
Given these points, the ARIMA(0,0,0)(0,1,1)[7] fits the data well.

```
# residual analysis ts.m1
checkresiduals(ts.m1)
```

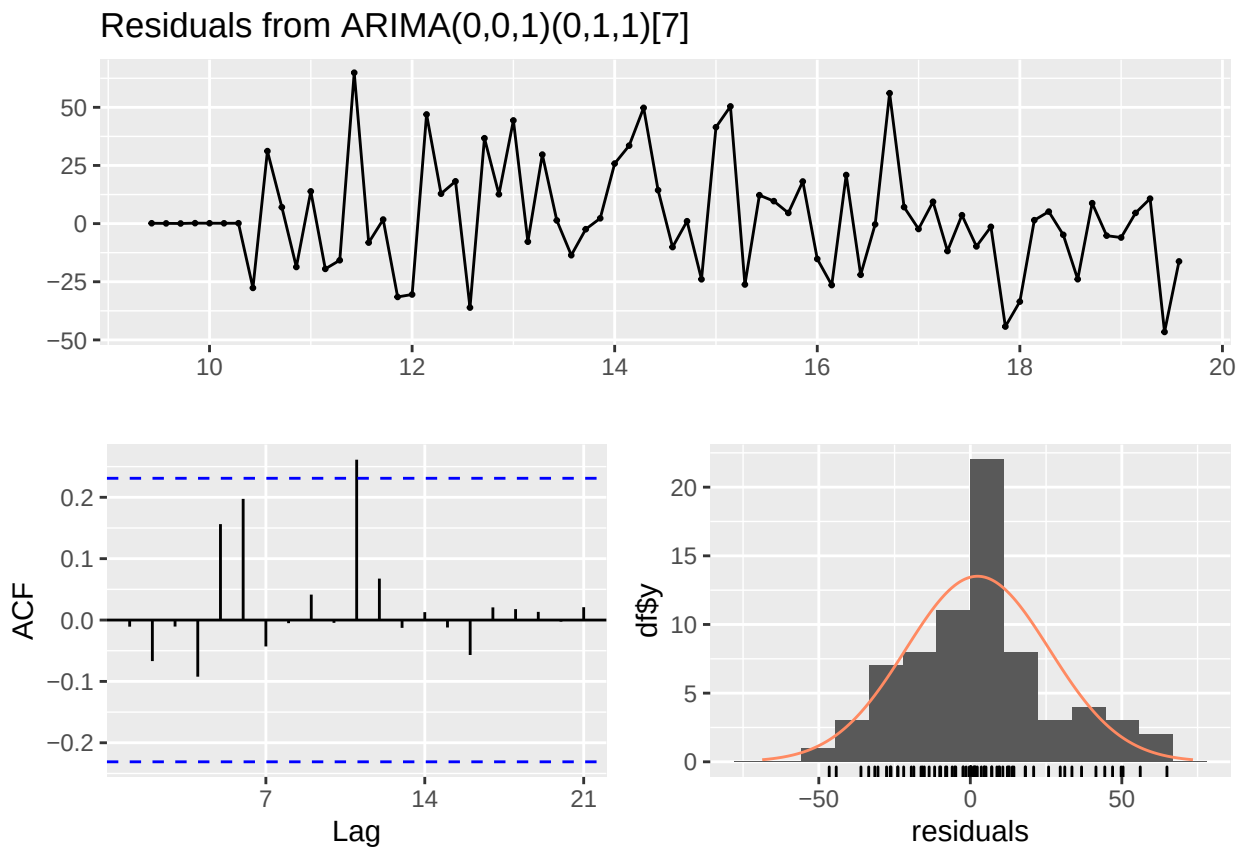



```
##
##  Ljung-Box test
##
## data:  Residuals from ARIMA(0,0,0)(0,1,1)[7]
## Q* = 15.105, df = 13, p-value = 0.3008
##
## Model df: 1.   Total lags used: 14
```

4.3.2 ARIMA(0,0,1)(0,1,1)[7]

The residual analysis for the second model, ARIMA(0,0,1)(0,1,1)[7] leads to similar observations to the ones described above. In short, the residuals in this model appear to be random too and do not display any inter-temporal correlations.

```
# residual analysis ts.m2
checkresiduals(ts.m2)
```




```
##
##  Ljung-Box test
##
## data:  Residuals from ARIMA(0,0,1)(0,1,1)[7]
## Q* = 12.811, df = 12, p-value = 0.3829
##
## Model df: 2.   Total lags used: 14
```

In conclusion, there are no warning messages that would make the two models unfit for forecasting.

5 ARIMA vs Holt-Winters

In this section, I compare the performance of the selected models by checking how accurately they forecast 31 days after 05-15-2015.

The below tables display the accuracy of the forecasts using the models specified in the previous sections. As can be seen, the obtained accuracy values of the three models are very close, with Holt-Winters having the highest accuracy based on the test set RMSE.

```
# ARIMA(0,0,0)(0,1,1)[7]
accuracy(forecast(ts.m1, h = 31), ts.test)
```

```
##                      ME     RMSE      MAE          MPE     MAPE      MASE
## Training set  2.704904 23.80928 17.50230  -0.06571051 12.06784 0.6618089
```

```
## Test set     -7.672889 32.63252 23.88794 -10.69108897 19.59120 0.9032671
##                   ACF1 Theil's U
## Training set 0.1007488        NA
## Test set     0.1593810 0.4893133
```

```
# ARIMA(0,0,1)(0,1,1)[7]
accuracy(forecast(ts.m2, h = 31), ts.test)
```

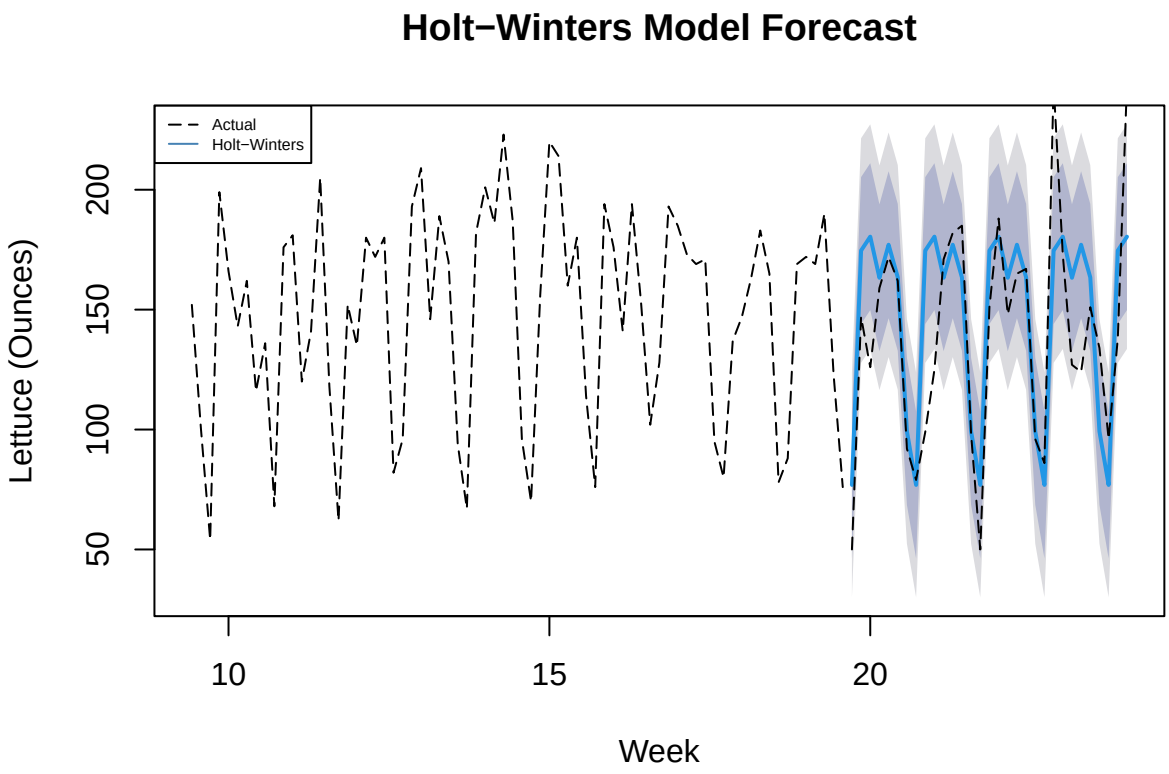
```
##                     ME     RMSE      MAE         MPE     MAPE      MASE
## Training set  2.390223 23.64983 17.43245  -0.1615379 12.01939 0.6591675
## Test set     -7.557267 32.55300 23.81573 -10.4938847 19.44368 0.9005367
##                     ACF1 Theil's U
## Training set -0.01077003        NA
## Test set      0.15955747  0.489141
```

```
# Holt-Winters
accuracy(forecast(ts.ets, h = 31), ts.test)
```

```
##                     ME     RMSE      MAE       MPE     MAPE      MASE      ACF1
## Training set  0.429331 22.34886 17.75836  -2.22342 12.97225 0.6714912 0.1043380
## Test set     -7.833417 32.04200 23.53357 -10.14445 19.08573 0.8898675 0.1545304
##              Theil's U
## Training set        NA
## Test set     0.4940561
```

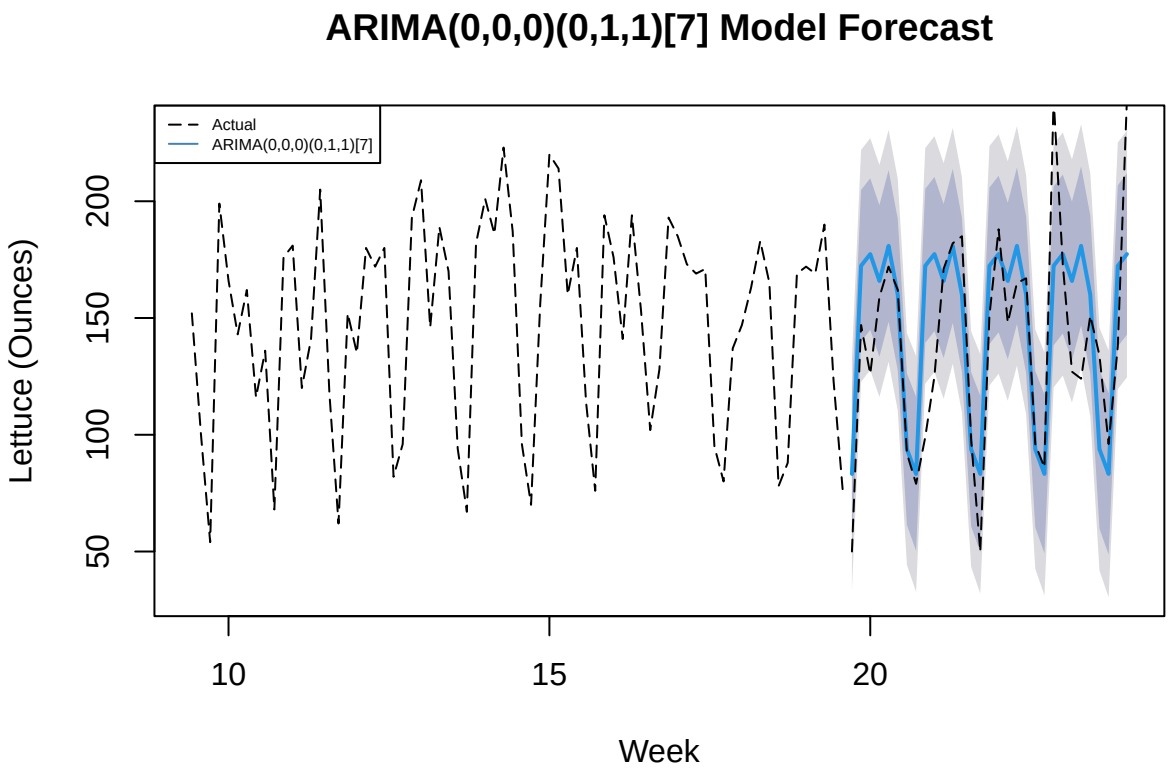
The below plot shows the Holt-Winters Model forecast obtained using the test set vs the actual time series.

```
plot(forecast(ts.ets, h = 31), main = "Holt-Winters Model Forecast", xlab = "Week",
    ylab = "Lettuce (Ounces)", lty = 5, col = "black")
lines(ts.test, col = "black", lty = 5)
legend("topleft", legend = c("Actual", "Holt-Winters"), col = c("black", "steel blue"),
    box.lty = 1, lty = c(5, 1), cex = 0.5)
```



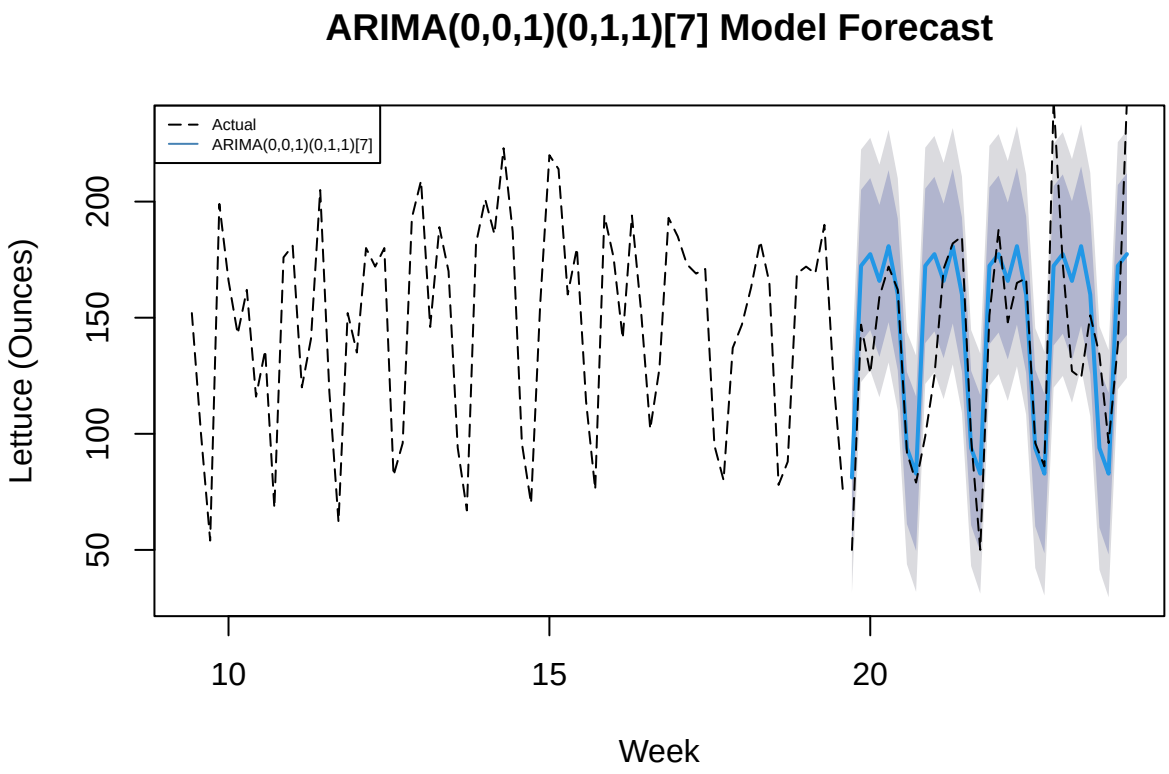
The below plot shows the ARIMA(0,0,0)(0,1,1)[7] Model forecast obtained using the test set vs the actual time series.

```
plot(forecast(ts.m1, h = 31), main = "ARIMA(0,0,0)(0,1,1)[7] Model Forecast", xlab = "Week",
    ylab = "Lettuce (Ounces)", lty = 5, col = "black")
lines(ts.test, col = "black", lty = 5)
legend("topleft", legend = c("Actual", "ARIMA(0,0,0)(0,1,1)[7]"), col = c("black",
    "steel blue"), box.lty = 1, lty = c(5, 1), cex = 0.5)
```



The below plot shows the ARIMA(0,0,1)(0,1,1)[7] Model forecast obtained using the test set vs the actual time series.

```
plot(forecast(ts.m2, h = 31), main = "ARIMA(0,0,1)(0,1,1)[7] Model Forecast", xlab = "Week",
    ylab = "Lettuce (Ounces)", lty = 5, col = "black")
lines(ts.test, col = "black", lty = 5)
legend("topleft", legend = c("Actual", "ARIMA(0,0,1)(0,1,1)[7]"), col = c("black",
    "steel blue"), box.lty = 1, lty = c(5, 1), cex = 0.5)
```



As can be seen, the forecasts of the three models are quite similar. Given that there are no significant differences in the accuracy of the forecasts, I will select the the model with the lowest RMSE, which is the Holt-Winters model.

6 Forecasting

In this section, I use the selected method to retrain the model using the whole time series and I create a forecast for the weeks (06/16/2015 to 06/29/2015) for the daily lettuce demand for the restaurant with StoreId: 46673.

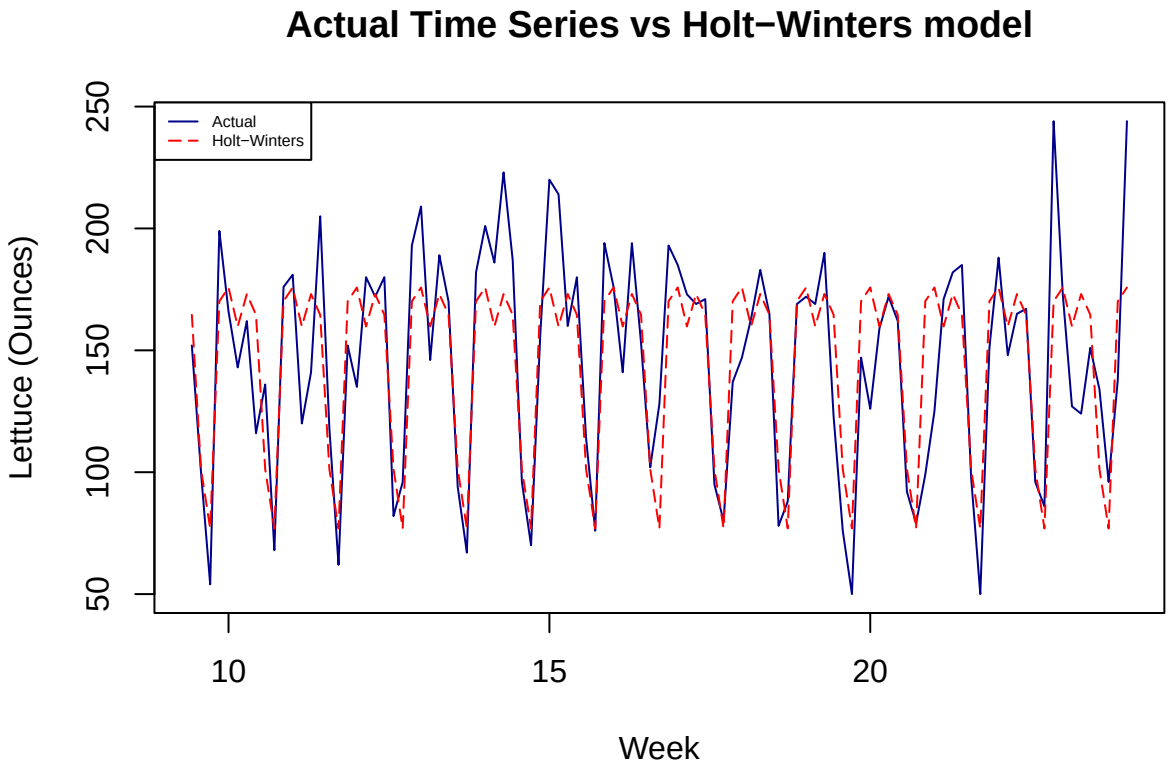
```
# retrain the model using the whole time series, ts and using the selected
# Holt-Winters model
ts.final <- ets(ts, model = "ANA")
ts.final
```

```
## ETS(A,N,A)
##
## Call:
##  ets(y = ts, model = "ANA")
##
##   Smoothing parameters:
##     alpha = 1e-04
##     gamma = 1e-04
##
```

```
##   Initial states:
##     l = 145.9345
##     s = 27.1516 13.8629 29.8353 24.1902 -69.0184 -44.6583
##           18.6367
##
##   sigma:  26.6343
##
##      AIC     AICc      BIC
## 1164.093 1166.484 1190.440
```

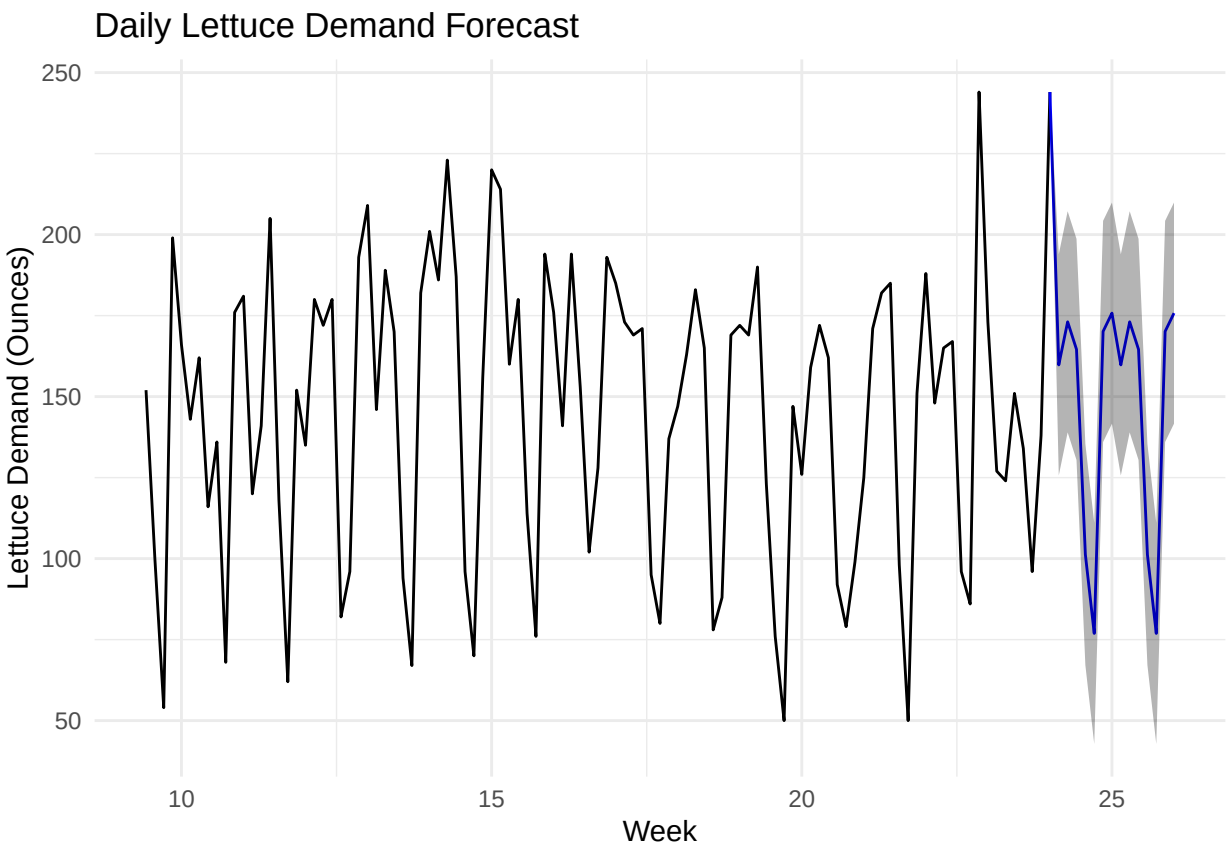
The below plot shows the Holt-Winters Model against the actual time series. As can be seen, the Holt-Winters model has a constant pattern that is repeated every 7 days, which is quite similar to the original seasonal data pattern. Even though, it does not perfectly follow the actual daily lettuce demand, it gives a very close approximation.

```
# plot the actual time series against the Holt-Winters model time series
plot(ts, main = "Actual Time Series vs Holt-Winters model", xlab = "Week", ylab = "Lettuce (Ounces)",
    lty = 1, col = "dark blue")
lines(fitted(ts.final), col = "red", lty = 5)
legend("topleft", legend = c("Actual", "Holt-Winters"), col = c("dark blue", "red"),
    box.lty = 1, lty = c(1, 5), cex = 0.5)
```



The below plot displays the forecast for the next 14 days.

```r
# forecast the demand for the next 14 days
f <- forecast(ts.final, h=14)
# plot of the forecast
ggplot2::autoplot(forecast(ts.final, h=14))+
  ggtitle("Daily Lettuce Demand Forecast")+
  xlab("Week")+
  ylab("Lettuce Demand (Ounces)")+
  theme_minimal()
```




```r
# create a dataframe with the daily lettuce demand for the next 14 days
dates = seq(from = as.Date("2015-06-16"), to = as.Date("2015-06-29"), by = "day")
forecast_df <- data.frame(date = dates, store_46673_ounces = f$mean)
forecast_df$date <- format(as.Date(forecast_df$date), "%d/%m/%Y")
forecast_df <- forecast_df %>%
    column_to_rownames(., var = "date")
```

The forecast for the next two weeks is displayed below:

```r
forecast_df
```

	store_46673_ounces
16/06/2015	159.79583
17/06/2015	173.08458
18/06/2015	164.56924

	store_46673_ounces
19/06/2015	101.27350
20/06/2015	76.91392
21/06/2015	170.12068
22/06/2015	175.76910
23/06/2015	159.79583
24/06/2015	173.08458
25/06/2015	164.56924
26/06/2015	101.27350
27/06/2015	76.91392
28/06/2015	170.12068
29/06/2015	175.76910

```
# create a csv file of the forecast_df
write.csv(forecast_df, "01996178.csv")
```

7 Conclusions

The aim of this project was to produce a forecast for the daily lettuce demand for the period 06/16/2015 - 06/29/2015. In this report I described the process I followed to obtain this forecast. Firstly, I showed how the time series was generated. Then, I applied two different popular methods for time series forecasting, the Holt-Winters model and the ARIMA model and compared their performance on the test set. The two methods produced models with similar accuracy, so my selection of the final model was solely based on their test set RMSE values. Finally, I used the selected Holt-Winters model to produce a forecast.

8 References

Jiahua, W. (October) Logistics and Supply Chain Analytics [Module] Imperial College London
Hyndman, R. and Athanasopoulos, G. (2021). Forecasting: Principles and practice. Third edition. OTexts.