SEAS 6414 - Python Applications in Data Analytics

Homework 8

Due Date: March 9, 2024 (10:00am EST)

Instructions: To complete the following task using Python, please download an Integrated Development Environment (IDE) of your choice. Ensure that your solution includes both the written code (input) and its corresponding output. Once completed, upload your solution in PDF format or any other format you prefer. The questions are worth 50 points each.

Note

I have left in some paths I took that did not pan out. Normally, I take them all out. For this last assignment, I left a few in.

All of the imports

In [ ]:
```
import pandas as pd
import numpy as np
import datetime as dt
import matplotlib.pyplot as plt
from scipy import stats
from sklearn.model_selection import TimeSeriesSplit  # you have everything done for
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_squared_log_error
from sklearn.metrics import median_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.metrics import root_mean_squared_error
from sklearn.metrics import mean_absolute_percentage_error
from scipy.optimize import minimize  # for function minimization
from tqdm.notebook import tqdm
import statsmodels.api as sm
import statsmodels.formula.api as smf  # statistics and econometrics
import statsmodels.tsa.api as smt
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score
import warnings
warnings.filterwarnings("ignore")
%matplotlib inline
from plotly import graph_objs as go
from plotly.offline import init_notebook_mode, iplot
import logging
```

```
from prophet import Prophet
logging.getLogger().setLevel(logging.ERROR)
```

1 Data Preparation

- Analyze merchant transaction data from January 1, 2033, to January 1, 2035.
- Resample the data to obtain daily transaction sums (in Dollars).

In [ ]:
```
pd.options.display.float_format = '{:,.2f}'.format
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 2000)

HW8F1 = pd.read_csv('./homework8.csv')
```

In [ ]:
```
HW8F1.shape
```

Out[ ]: (1513719, 4)

In [ ]:
```
HW8F1.head()
```

Out[ ]:

	Unnamed: 0	merchant	time	amount_usd_in_cents
0	1	faa029c6b0	2034-06-17 23:34:14	6349
1	2	ed7a7d91aa	2034-12-27 00:40:38	3854
2	3	5608f200cf	2034-04-30 01:29:42	789
3	4	15b1a0d61e	2034-09-16 01:06:23	4452
4	5	4770051790	2034-07-22 16:21:42	20203

In [ ]:
```
HW8F1['merchant'].nunique()
```

Out[ ]: 14351

In [ ]:
```
# Rename the unnamed column
HW8F1 = HW8F1.rename(columns={'Unnamed: 0': 'ID'})
```

In [ ]:
```
# convert the cents to dollars
HW8F1 = HW8F1.rename(columns={'amount_usd_in_cents': 'amount_usd_in_dollars'})  # R
HW8F1 = HW8F1.assign(amount_usd_in_dollars=lambda x: x['amount_usd_in_dollars'] / 1
```

In [ ]:
```
#transform to datetime format
HW8F1['time'] = pd.to_datetime(HW8F1['time'])
```

In [ ]:
```
HW8F1.head()
```

Out[ ]:

	ID	merchant	time	amount_usd_in_dollars
0	1	faa029c6b0	2034-06-17 23:34:14	63.49
1	2	ed7a7d91aa	2034-12-27 00:40:38	38.54
2	3	5608f200cf	2034-04-30 01:29:42	7.89
3	4	15b1a0d61e	2034-09-16 01:06:23	44.52
4	5	4770051790	2034-07-22 16:21:42	202.03

In [ ]:
```
HW8F1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1513719 entries, 0 to 1513718
Data columns (total 4 columns):
 #   Column                 Non-Null Count    Dtype
---  ------                 --------------    -----
 0   ID                     1513719 non-null  int64
 1   merchant               1513719 non-null  object
 2   time                   1513719 non-null  datetime64[ns]
 3   amount_usd_in_dollars  1513719 non-null  float64
dtypes: datetime64[ns](1), float64(1), int64(1), object(1)
memory usage: 46.2+ MB
```

In [ ]:
```
# Resample by merchant, day on 'time' / Sum the amount_usd_in_cents for each group
daily_sales1 = HW8F1.groupby('merchant').resample('D', on='time',include_groups=Fal
```

In [ ]:
```
daily_sales1.shape
```

Out[ ]:
```
(2883395, 1)
```

In [ ]:
```
daily_sales1.head()
```

Out[ ]:

merchant	time	amount_usd_in_dollars
0002b63b92	**2033-05-16**	33.79
0002d07bba	**2034-10-11**	55.49
	2034-10-12	0.00
	2034-10-13	0.00
	2034-10-14	0.00

At first it seemed to me that I should group by merchant. But I realized that this would create too many rows and was not the goal. It is the total sales per day irrespective of the merchant. Grouping by merchant is not called for here. So, I will create a new dataframe without it.

In [ ]:
```
# Resample by day on 'time' / Sum the amount_usd_in_cents for each group / Reset in
daily_sales2 = HW8F1.resample('D', on='time').sum().reindex().drop(columns=['ID','m
```

In [ ]:
```
daily_sales2.shape
```

Out[ ]: (730, 1)

In [ ]:
```
daily_sales2.head()
```

Out[ ]:

	amount_usd_in_dollars
time	
2033-01-01	4,415.40
2033-01-02	4,758.62
2033-01-03	6,283.53
2033-01-04	4,851.11
2033-01-05	10,016.21

In [ ]:
```
daily_sales2.describe()
```

Out[ ]:

	amount_usd_in_dollars
count	730.00
mean	321,088.02
std	247,370.12
min	4,415.40
25%	134,365.02
50%	284,525.09
75%	473,967.63
max	2,663,536.30

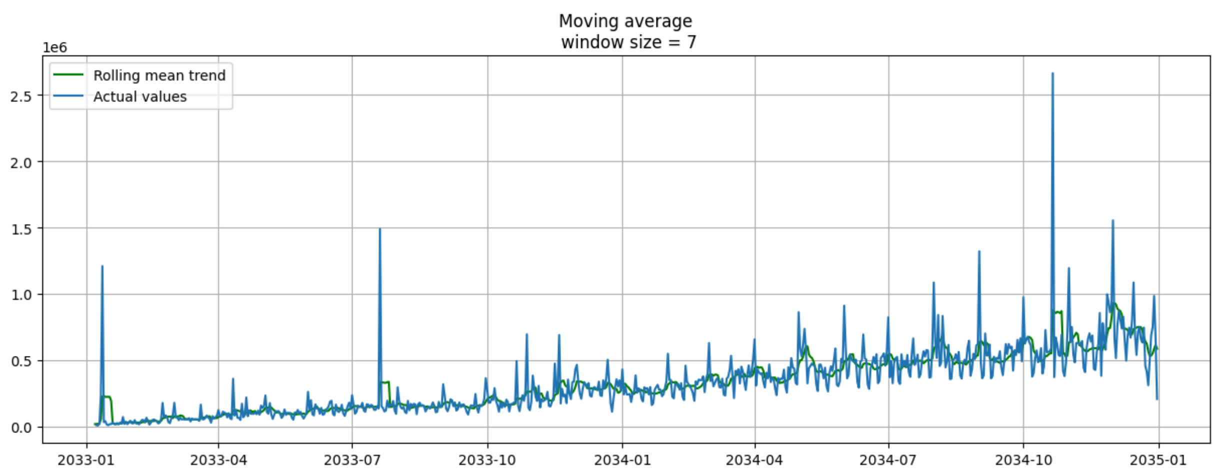
Lets look at the data to see what we have.

In [ ]:
```
def plotMovingAverage(
    series, window, plot_intervals=False, scale=1.96, plot_anomalies=False
):

    """
        series - dataframe with timeseries
        window - rolling window size
        plot_intervals - show confidence intervals
        plot_anomalies - show anomalies


```

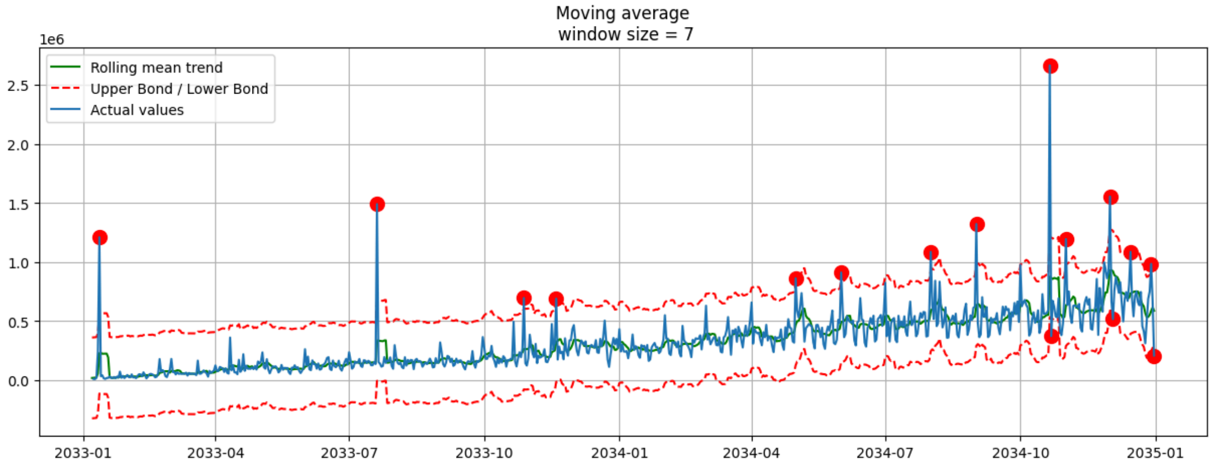
```
    """
    rolling_mean = series.rolling(window=window).mean()

    plt.figure(figsize=(15, 5))
    plt.title("Moving average\n window size = {}".format(window))
    plt.plot(rolling_mean, "g", label="Rolling mean trend")

    # Plot confidence intervals for smoothed values
    if plot_intervals:
        mae = mean_absolute_error(series[window:], rolling_mean[window:])
        deviation = np.std(series[window:] - rolling_mean[window:])
        lower_bond = rolling_mean - (mae + scale * deviation)
        upper_bond = rolling_mean + (mae + scale * deviation)
        plt.plot(upper_bond, "r--", label="Upper Bond / Lower Bond")
        plt.plot(lower_bond, "r--")

        # Having the intervals, find abnormal values
        if plot_anomalies:
            anomalies = pd.DataFrame(index=series.index, columns=series.columns)
            anomalies[series < lower_bond] = series[series < lower_bond]
            anomalies[series > upper_bond] = series[series > upper_bond]
            plt.plot(anomalies, "ro", markersize=10)

    plt.plot(series[window:], label="Actual values")
    plt.legend(loc="upper left")
    plt.grid(True)
```

A seven day moving average.

In [ ]:
```
plotMovingAverage(daily_sales2,7)
```



In [ ]:
```
plotMovingAverage(
    daily_sales2, 7, plot_intervals=True, plot_anomalies=True
)  # weekly smoothing
```



It looks like there are a few outliers. I count 16, three of which seem extreme. Let's do a simple process to remove them.

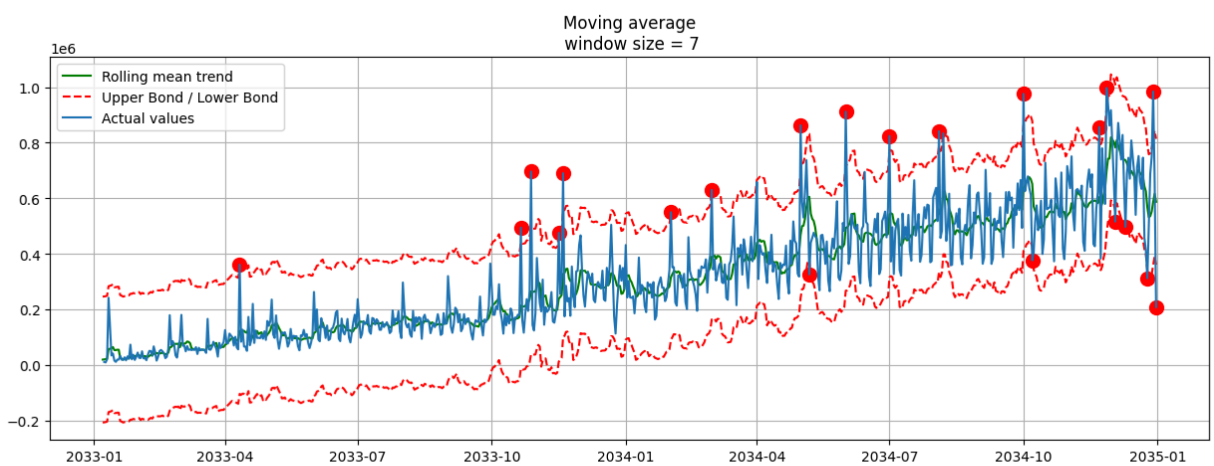
In [ ]:
```
daily_sales3 = daily_sales2[np.abs(stats.zscore(daily_sales2['amount_usd_in_dollars
```

In [ ]:
```
daily_sales3.shape
```

Out[ ]: (722, 1)

Went from 730 down to 722 so we removed 8 outliers. Lets see the result.

In [ ]:
```
plotMovingAverage(
    daily_sales3, 7, plot_intervals=True, plot_anomalies=True
)  # weekly smoothing
```



It went from 16 outliers to 21. But these are less severe. I am not sure that it is better.

This looks better. So, here is what I have:

- daily_sales1 - the data grouped by merchant and summed by day. Not really useful.
- daily_sales2 - the data summed by day.
- daily_sales3 - the data summed by day and removal of outliers based on z score.

After reviewing the graphs, I am not sure I can tell what to use for seasonality. My best guess is weekly or seven days. I am going to leave the outliers in for this. So, going forward I will be using daily_sales2.

I also note that the data is not stationary. There is a clear trend upwards. I think that the SARIMA section addressed this. Since our homework doe not require SARIMA, I will not address it.

2 Forecasting Methods

2.1 Holt-Winters (Triple Exponential Smoothing)

Apply Holt-Winters method to capture seasonality, trend, and level in the daily transaction data.

In [ ]:
```
class HoltWinters:

    """
    Holt-Winters model with the anomalies detection using Brutlag method

    # series - initial time series
    # slen - length of a season
    # alpha, beta, gamma - Holt-Winters model coefficients
    # n_preds - predictions horizon
    # scaling_factor - sets the width of the confidence interval by Brutlag (usuall

    """


    def __init__(self, series, slen, alpha, beta, gamma, n_preds, scaling_factor=1.
        self.series = series
        self.slen = slen
        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma
        self.n_preds = n_preds
        self.scaling_factor = scaling_factor

    def initial_trend(self):
        sum = 0.0
        for i in range(self.slen):
            sum += float(self.series[i + self.slen] - self.series[i]) / self.slen
        return sum / self.slen

    def initial_seasonal_components(self):
        seasonals = {}
        season_averages = []
        n_seasons = int(len(self.series) / self.slen)
        # let's calculate season averages
        for j in range(n_seasons):
            season_averages.append(
```

```
                sum(self.series[self.slen * j : self.slen * j + self.slen])
                / float(self.slen)
            )
        # let's calculate initial values
        for i in range(self.slen):
            sum_of_vals_over_avg = 0.0
            for j in range(n_seasons):
                sum_of_vals_over_avg += (
                    self.series[self.slen * j + i] - season_averages[j]
                )
            seasonals[i] = sum_of_vals_over_avg / n_seasons
        return seasonals

    def triple_exponential_smoothing(self):
        self.result = []
        self.Smooth = []
        self.Season = []
        self.Trend = []
        self.PredictedDeviation = []
        self.UpperBond = []
        self.LowerBond = []

        seasonals = self.initial_seasonal_components()

        for i in range(len(self.series) + self.n_preds):
            if i == 0:  # components initialization
                smooth = self.series[0]
                trend = self.initial_trend()
                self.result.append(self.series[0])
                self.Smooth.append(smooth)
                self.Trend.append(trend)
                self.Season.append(seasonals[i % self.slen])

                self.PredictedDeviation.append(0)

                self.UpperBond.append(
                    self.result[0] + self.scaling_factor * self.PredictedDeviation[
                )

                self.LowerBond.append(
                    self.result[0] - self.scaling_factor * self.PredictedDeviation[
                )
                continue

            if i >= len(self.series):  # predicting
                m = i - len(self.series) + 1
                self.result.append((smooth + m * trend) + seasonals[i % self.slen])

                # when predicting we increase uncertainty on each step
                self.PredictedDeviation.append(self.PredictedDeviation[-1] * 1.01)

            else:
                val = self.series[i]
                last_smooth, smooth = (
                    smooth,
                    self.alpha * (val - seasonals[i % self.slen])
```

```
                    + (1 - self.alpha) * (smooth + trend),
                )
                trend = self.beta * (smooth - last_smooth) + (1 - self.beta) * tren
                seasonals[i % self.slen] = (
                    self.gamma * (val - smooth)
                    + (1 - self.gamma) * seasonals[i % self.slen]
                )
                self.result.append(smooth + trend + seasonals[i % self.slen])

                # Deviation is calculated according to Brutlag algorithm.
                self.PredictedDeviation.append(
                    self.gamma * np.abs(self.series[i] - self.result[i])
                    + (1 - self.gamma) * self.PredictedDeviation[-1]
                )

            self.UpperBond.append(
                self.result[-1] + self.scaling_factor * self.PredictedDeviation[-1]
            )

            self.LowerBond.append(
                self.result[-1] - self.scaling_factor * self.PredictedDeviation[-1]
            )

            self.Smooth.append(smooth)
            self.Trend.append(trend)
            self.Season.append(seasonals[i % self.slen])
```

In [ ]:

```
def timeseriesCVscore(params, series, loss_function=mean_squared_error, slen=24):
    """
        Returns error on CV

        params - vector of parameters for optimization
        series - dataset with timeseries
        slen - season length for Holt-Winters model
    """
    # errors array
    errors = []

    values = series.values
    alpha, beta, gamma = params

    # set the number of folds for cross-validation
    tscv = TimeSeriesSplit(n_splits=3)

    # iterating over folds, train model on each, forecast and calculate error
    for train, test in tscv.split(values):

        model = HoltWinters(
            series=values[train],
            slen=slen,
            alpha=alpha,
            beta=beta,
            gamma=gamma,
            n_preds=len(test),
        )
        model.triple_exponential_smoothing()
```

```
        predictions = model.result[-len(test) :]
        actual = values[test]
        error = loss_function(predictions, actual)
        errors.append(error)

    return np.mean(np.array(errors))
```

In [ ]:
```
%%time
data = daily_sales2.amount_usd_in_dollars[:-20]  # leave some data for testing

# initializing model parameters alpha, beta and gamma
x = [0, 0, 0]

# Minimizing the loss function
opt = minimize(
    timeseriesCVscore,
    x0=x,
    args=(data, mean_squared_error), # changed from log because it complained about
    method="TNC",
    bounds=((0, 1), (0, 1), (0, 1)),
)

# Take optimal values...
alpha_final, beta_final, gamma_final = opt.x
print(alpha_final, beta_final, gamma_final)

# ...and train the model with them, forecasting for the next 50 hours
model = HoltWinters(
    data,
    slen=7,
    alpha=alpha_final,
    beta=beta_final,
    gamma=gamma_final,
    n_preds=50,
    scaling_factor=3,
)
model.triple_exponential_smoothing()
```

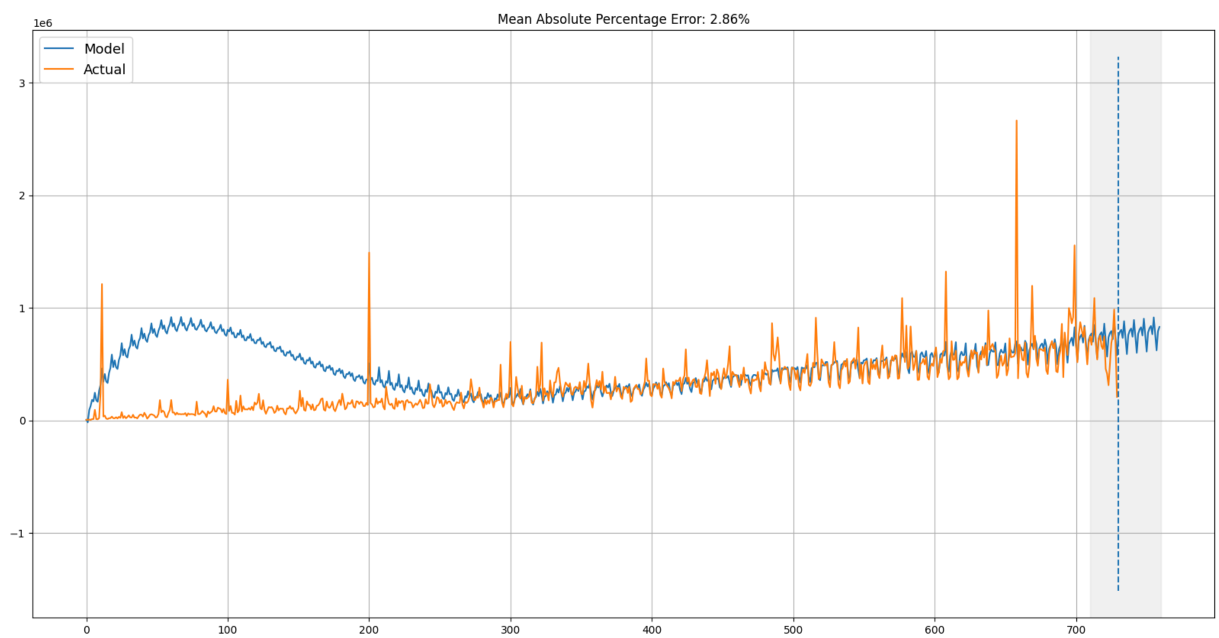
```
0.010009174709418145 0.024494780538362126 0.08887341741955923
CPU times: total: 1.11 s
Wall time: 2.18 s
```

In [ ]:
```
def plotHoltWinters(series, plot_intervals=False, plot_anomalies=False):
    """
        series - dataset with timeseries
        plot_intervals - show confidence intervals
        plot_anomalies - show anomalies
    """

    plt.figure(figsize=(20, 10))
    plt.plot(model.result, label="Model")
    plt.plot(series.values, label="Actual")
    error = mean_absolute_percentage_error(series.values, model.result[: len(series
    plt.title("Mean Absolute Percentage Error: {0:.2f}%".format(error))
```

```
    if plot_anomalies:
        anomalies = np.array([np.NaN] * len(series))
        anomalies[series.values < model.LowerBond[: len(series)]] = series.values[
            series.values < model.LowerBond[: len(series)]
        ]
        anomalies[series.values > model.UpperBond[: len(series)]] = series.values[
            series.values > model.UpperBond[: len(series)]
        ]
        plt.plot(anomalies, "o", markersize=10, label="Anomalies")

    if plot_intervals:
        plt.plot(model.UpperBond, "r--", alpha=0.5, label="Up/Low confidence")
        plt.plot(model.LowerBond, "r--", alpha=0.5)
        plt.fill_between(
            x=range(0, len(model.result)),
            y1=model.UpperBond,
            y2=model.LowerBond,
            alpha=0.2,
            color="grey",
        )

    plt.vlines(
        len(series),
        ymin=min(model.LowerBond),
        ymax=max(model.UpperBond),
        linestyles="dashed",
    )
    plt.axvspan(len(series) - 20, len(model.result), alpha=0.3, color="lightgrey")
    plt.grid(True)
    plt.axis("tight")
    plt.legend(loc="best", fontsize=13);
```

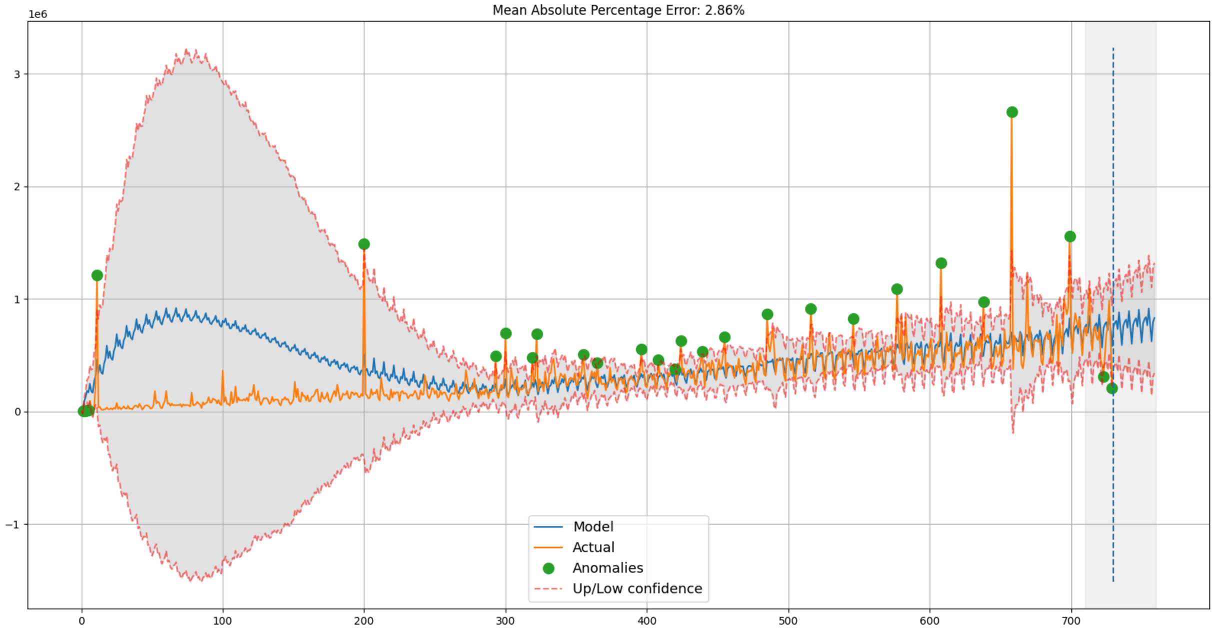
In [ ]:
```
MAPE_HW = mean_absolute_percentage_error(daily_sales2.amount_usd_in_dollars.values,
```

In [ ]:
```
plotHoltWinters(daily_sales2.amount_usd_in_dollars)
```



It starts off bad but then settles into a nice pattern that follows the actual data less some of the extremes. Are the extremes noise or should we be trying to map to them? I am not sure.

In [ ]:
```
plotHoltWinters(daily_sales2.amount_usd_in_dollars, plot_intervals=True, plot_anoma
```



Still a lot of values outside the confidence interval.

2.2 Time Series Cross-Validation

Use time series cross-validation for evaluating the performance of the models.

In [ ]:
```
# for time-series cross-validation set 5 folds
tscv = TimeSeriesSplit(n_splits=5)
```

2.3 Feature Engineering with Lags

Create new features based on lagged values of the time series for linear regression and random forest models.

In [ ]:
```
# Creating a copy of the initial dataframe to make various transformations
data = pd.DataFrame(daily_sales2.amount_usd_in_dollars.copy())
data.columns = ["y"]
```

In [ ]:
```
# Adding the lag of the target variable from 6 steps back up to 24
for i in range(6, 25):
    data["lag_{}".format(i)] = data.y.shift(i)
```

In [ ]:
```
# take a look at the new dataframe
data.tail(7)
```

Out[ ]:

time	y	lag_6	lag_7	lag_8	lag_9	lag_10	lag_11
2034-12-25	312,083.41	752,171.23	712,842.49	540,359.50	688,965.40	1,086,375.85	808,416.72
2034-12-26	499,564.87	659,891.45	752,171.23	712,842.49	540,359.50	688,965.40	1,086,375.85
2034-12-27	688,573.47	633,820.61	659,891.45	752,171.23	712,842.49	540,359.50	688,965.40
2034-12-28	750,020.80	746,183.26	633,820.61	659,891.45	752,171.23	712,842.49	540,359.50
2034-12-29	984,672.08	457,942.73	746,183.26	633,820.61	659,891.45	752,171.23	712,842.49
2034-12-30	668,559.95	412,629.69	457,942.73	746,183.26	633,820.61	659,891.45	752,171.23
2034-12-31	208,533.08	312,083.41	412,629.69	457,942.73	746,183.26	633,820.61	659,891.45

2.4 Linear Regression Model

Implement a linear regression model using the lagged features.

In [ ]:

```
def timeseries_train_test_split(X, y, test_size):
    """
        Perform train-test split with respect to time series structure
    """

    # get the index after which test set starts
    test_index = int(len(X) * (1 - test_size))

    X_train = X.iloc[:test_index]
    y_train = y.iloc[:test_index]
    X_test = X.iloc[test_index:]
    y_test = y.iloc[test_index:]

    return X_train, X_test, y_train, y_test
```

In [ ]:

```
y = data.dropna().y
X = data.dropna().drop(["y"], axis=1)

# reserve 30% of data for testing
X_train, X_test, y_train, y_test = timeseries_train_test_split(X, y, test_size=0.3)
```

In [ ]:

```
# machine learning in two lines
lr = LinearRegression()
lr.fit(X_train, y_train)
```

Out[ ]:

```
LinearRegression

LinearRegression()
```

Calculate and store the MAPE for LR.

In [ ]:

```
MAPE_LR = mean_absolute_percentage_error(lr.predict(X_test), y_test)
MAPE_LR
```

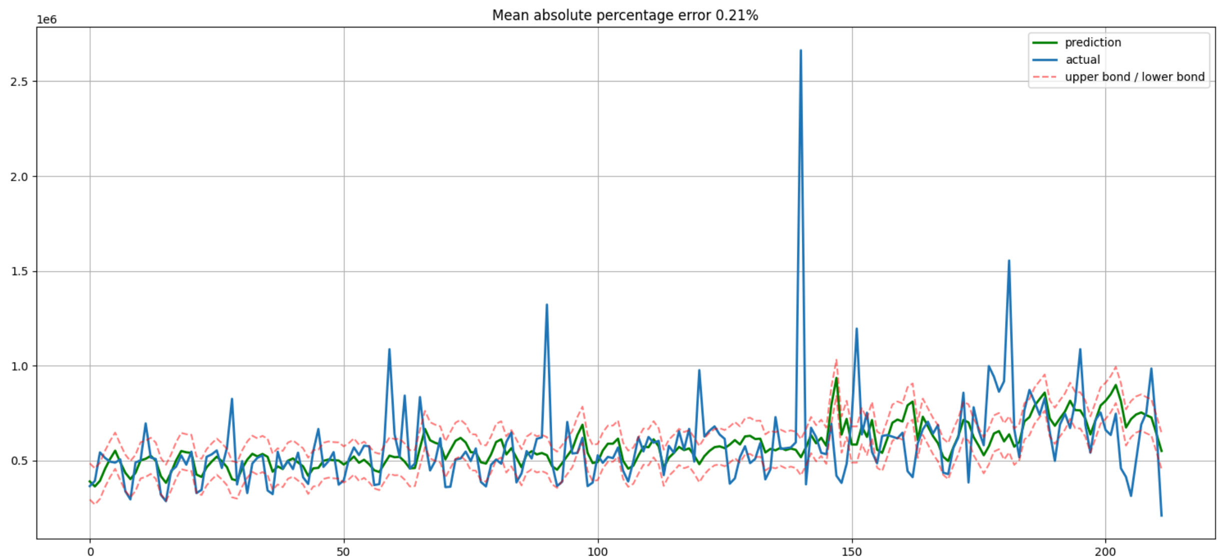
Out[ ]: 0.20939886824567627

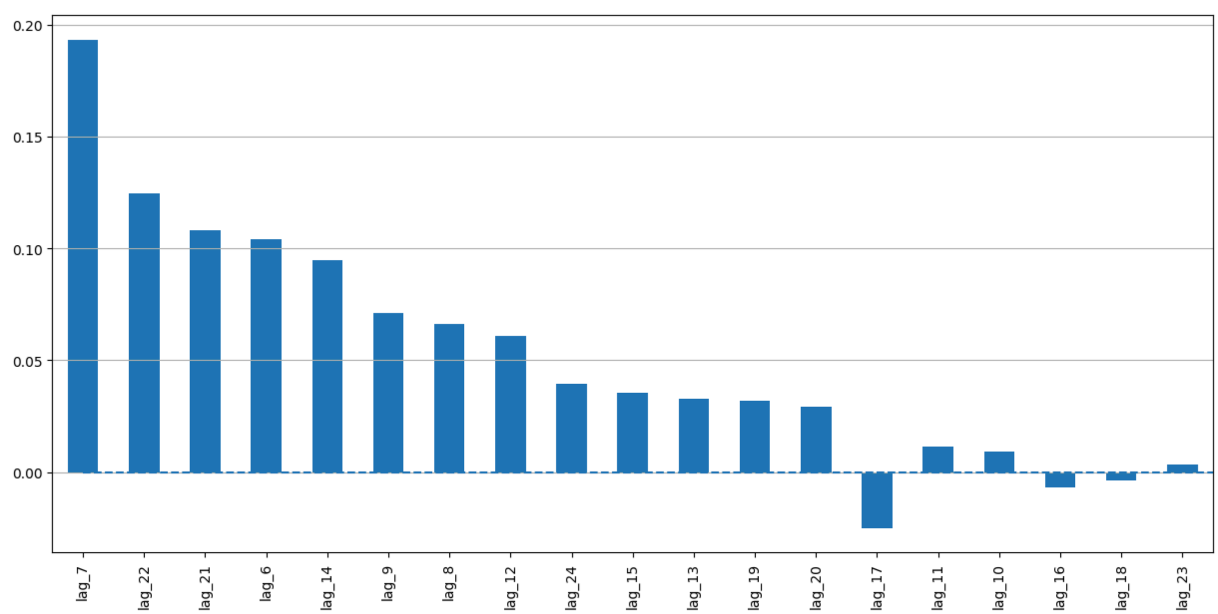
In [ ]:

```
def plotModelResults(
    model, X_train=X_train, X_test=X_test, plot_intervals=False, plot_anomalies=Fal
):
    """
        Plots modelled vs fact values, prediction intervals and anomalies

    """

    prediction = model.predict(X_test)

    plt.figure(figsize=(15, 7))
    plt.plot(prediction, "g", label="prediction", linewidth=2.0)
    plt.plot(y_test.values, label="actual", linewidth=2.0)

    if plot_intervals:
        cv = cross_val_score(
            model, X_train, y_train, cv=tscv, scoring="neg_mean_absolute_error"
        )
        mae = cv.mean() * (-1)
        deviation = cv.std()

        scale = 1.96
        lower = prediction - (mae + scale * deviation)
        upper = prediction + (mae + scale * deviation)

        plt.plot(lower, "r--", label="upper bond / lower bond", alpha=0.5)
        plt.plot(upper, "r--", alpha=0.5)

        if plot_anomalies:
            anomalies = np.array([np.NaN] * len(y_test))
            anomalies[y_test < lower] = y_test[y_test < lower]
            anomalies[y_test > upper] = y_test[y_test > upper]
            plt.plot(anomalies, "o", markersize=10, label="Anomalies")

    error = mean_absolute_percentage_error(prediction, y_test)
    plt.title("Mean absolute percentage error {0:.2f}%".format(error))
    plt.legend(loc="best")
    plt.tight_layout()
    plt.grid(True)


def plotCoefficients(model):
    """
```

```
        Plots sorted coefficient values of the model
    """

    coefs = pd.DataFrame(model.coef_, X_train.columns)
    coefs.columns = ["coef"]
    coefs["abs"] = coefs.coef.apply(np.abs)
    coefs = coefs.sort_values(by="abs", ascending=False).drop(["abs"], axis=1)

    plt.figure(figsize=(15, 7))
    coefs.coef.plot(kind="bar")
    plt.grid(True, axis="y")
    plt.hlines(y=0, xmin=0, xmax=len(coefs), linestyles="dashed");
```

In [ ]:
```
plotModelResults(lr, plot_intervals=True)
plotCoefficients(lr)
```





Add Weekday and Weekend to see if they can help the model.

In [ ]:
```
data_plus_time = data.copy(deep=True)
data_plus_time.index = pd.to_datetime(data.index)
```

```
data_plus_time["weekday"] = data_plus_time.index.weekday
data_plus_time["is_weekend"] = data_plus_time.weekday.isin([5, 6]) * 1
data_plus_time.tail()
```

Out[ ]:

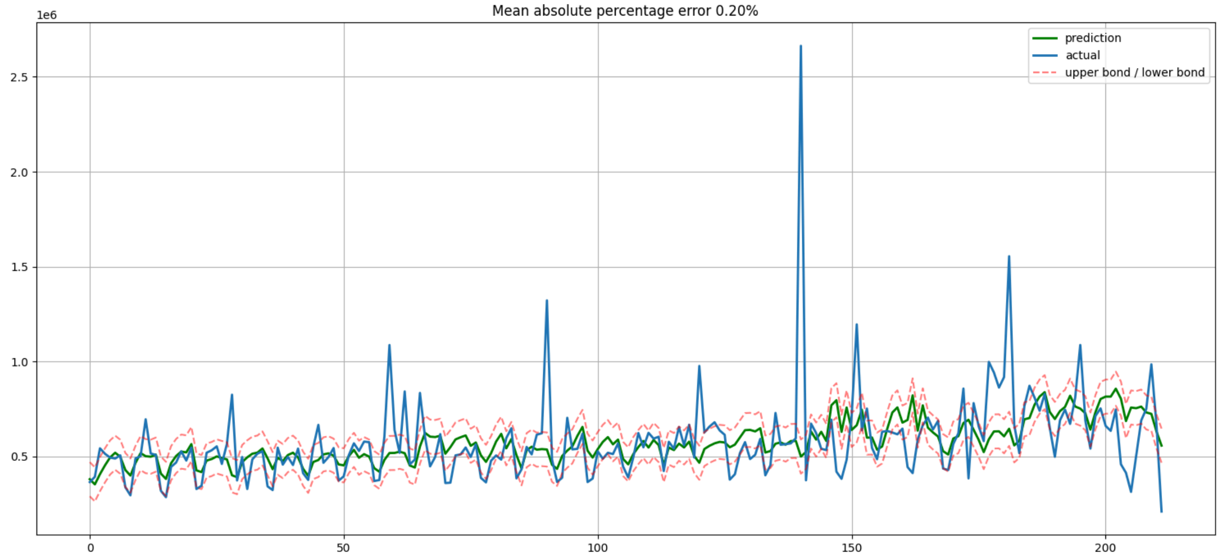
time	y	lag_6	lag_7	lag_8	lag_9	lag_10	lag_11	
2034-12-27	688,573.47	633,820.61	659,891.45	752,171.23	712,842.49	540,359.50	688,965.40	1,08
2034-12-28	750,020.80	746,183.26	633,820.61	659,891.45	752,171.23	712,842.49	540,359.50	68
2034-12-29	984,672.08	457,942.73	746,183.26	633,820.61	659,891.45	752,171.23	712,842.49	54
2034-12-30	668,559.95	412,629.69	457,942.73	746,183.26	633,820.61	659,891.45	752,171.23	71
2034-12-31	208,533.08	312,083.41	412,629.69	457,942.73	746,183.26	633,820.61	659,891.45	75

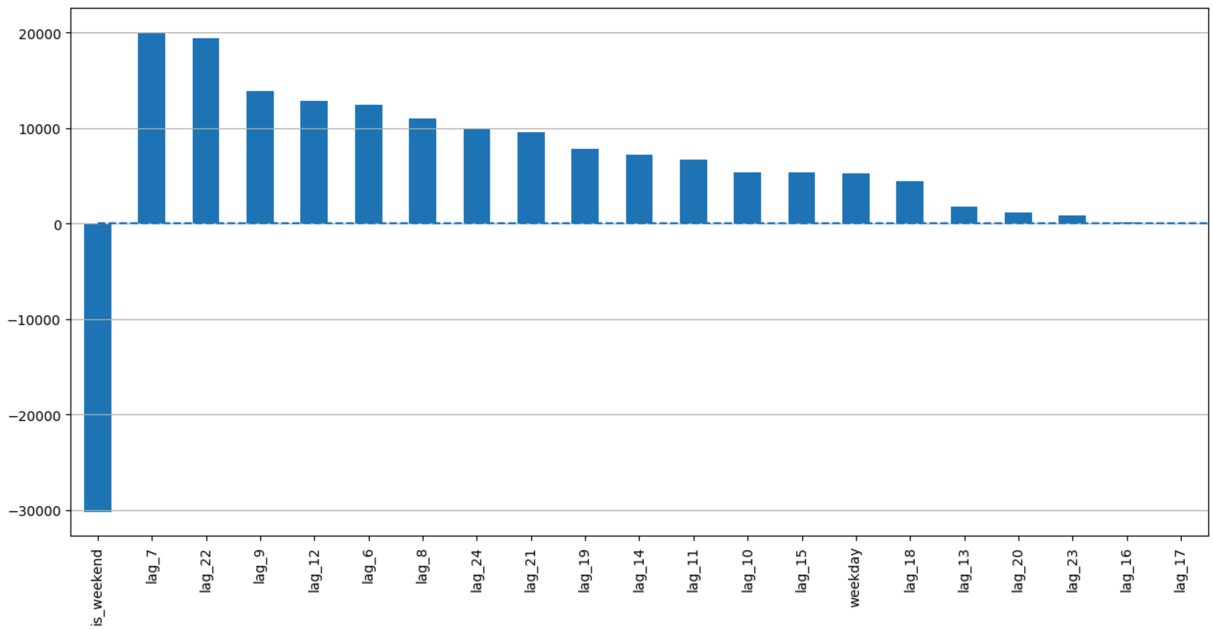
In [ ]:

```
scaler = StandardScaler()
```

In [ ]:

```
y = data_plus_time.dropna().y
X = data_plus_time.dropna().drop(["y"], axis=1)

X_train, X_test, y_train, y_test = timeseries_train_test_split(X, y, test_size=0.3)

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

lr = LinearRegression()
lr.fit(X_train_scaled, y_train)

plotModelResults(lr, X_train=X_train_scaled, X_test=X_test_scaled, plot_intervals=T
plotCoefficients(lr)
```





Calculate and store the MAPE for Linear Regression with Weekend and Weekday.

In [ ]:
```
MAPE_LRS = mean_absolute_percentage_error(lr.predict(X_test_scaled), y_test)
MAPE_LRS
```

Out[ ]: 0.20237243649488235

My interpretation of this is that it is not on a weekend and follows a week / month cycle.

2.5 Random Forest Model

Similarly, use the lagged features in a random forest model for forecasting.

In [ ]:
```
# Start fresh with the variables
del y, X, X_train, X_test, y_train, y_test
```

In [ ]:
```
y = data.dropna().y
X = data.dropna().drop(["y"], axis=1)

# reserve 30% of data for testing
X_train, X_test, y_train, y_test = timeseries_train_test_split(X, y, test_size=0.3)
```

In [ ]:
```
X_train.head()
```

Out[ ]:

time	lag_6	lag_7	lag_8	lag_9	lag_10	lag_11	lag_12	lag_13
2033-01-25	32,943.04	19,194.69	18,953.62	11,087.61	15,817.95	41,119.03	34,005.18	1,210,293.14
2033-01-26	18,733.76	32,943.04	19,194.69	18,953.62	11,087.61	15,817.95	41,119.03	34,005.18
2033-01-27	18,221.52	18,733.76	32,943.04	19,194.69	18,953.62	11,087.61	15,817.95	41,119.03
2033-01-28	27,234.13	18,221.52	18,733.76	32,943.04	19,194.69	18,953.62	11,087.61	15,817.95
2033-01-29	16,732.76	27,234.13	18,221.52	18,733.76	32,943.04	19,194.69	18,953.62	11,087.61

In [ ]:
```
param_grid = {'n_estimators': [1, 10, 50, 100, 200, 300, 400]}
grid = GridSearchCV(RandomForestRegressor(), param_grid, cv=7)
grid.fit(X_train, y_train)
grid.best_params_
```

Out[ ]:
```
{'n_estimators': 50}
```

Every time I run this, I get a different answer. I have received 300, 400, 1, and 50 as answers.

In [ ]:
```
rf = RandomForestRegressor(grid.best_params_['n_estimators'])
rf.fit(X_train, y_train)
y_train = rf.predict(X_train)
y_test = rf.predict(X_test)
```

In [ ]:
```
MAPE_RF = mean_absolute_percentage_error(rf.predict(X_test), y_test)
MAPE_RF
```

Out[ ]:
```
0.0
```

In [ ]:
```
print(y_test)
```

```
[377614.7172 317472.7    372410.5994 444011.5786 484880.2576 539458.6112
 536188.1186 395039.727  310315.3788 457455.0974 454124.5966 435499.7756
 465931.7232 506114.9784 395805.3546 319139.8392 408542.5462 460794.3628
 464899.1334 465903.928  525719.7828 418903.5588 393546.457  475425.7808
 456353.1398 484749.3524 521986.9988 564574.3698 408814.6634 337204.3016
 439993.7768 463996.71   485613.687  504637.653  543777.0022 418254.451
 347299.372  452706.3696 424912.3464 466728.037  483377.5258 494101.7964
 385590.9002 406435.6852 474322.2978 468496.9314 476370.1454 509013.9276
 512335.844  482991.7368 456912.9104 509490.141  444881.0992 457738.265
 481708.2056 465770.5276 424798.8656 391148.72   471280.7168 462901.488
 470197.8134 511022.1528 472105.7526 454592.0678 481111.9506 483123.7904
 481951.6358 485472.2258 512890.6096 498046.4936 494874.366  525805.8896
 499249.7686 454296.536  475940.0002 553612.6334 506116.3258 464869.2234
 469450.3876 485000.4904 470987.3692 466225.1386 498255.534  505382.233
 470964.3354 453825.8104 462964.0454 490741.7506 545254.47   513880.7442
 518826.7994 454785.5468 472933.232  472998.7832 488905.5776 519305.4196
 534511.8158 504350.2796 511309.8618 449744.0212 469556.7324 461259.5028
 491667.523  527341.536  483003.7104 462280.424  497073.5726 485981.0298
 454285.724  462171.3466 523196.4804 508118.5414 516384.822  450736.9674
 494344.2492 472610.7562 495878.6724 520073.512  544400.5604 490452.1536
 478498.4698 480292.5016 467686.7476 470956.796  518502.4466 485610.0488
 508760.9076 503466.899  495671.305  486309.1302 488457.3694 530564.0692
 483657.5522 469469.4206 497901.9906 485789.5218 484049.1116 459326.8728
 522025.169  500444.573  459249.671  498416.1104 497765.9614 497687.7498
 515763.9404 544100.5874 523290.4952 488472.0126 487945.5486 496273.2286
 469842.2326 468951.0298 503488.5044 483038.1274 464456.1458 491146.3872
 492171.505  476804.3468 509465.2788 518041.9282 517565.453  501007.9688
 538582.6524 490683.7466 498895.7702 509731.7902 518691.3724 500569.757
 477908.8494 534889.5126 490474.1254 483731.4406 488897.1202 514307.735
 501549.5726 479692.9008 490546.9362 519117.5152 494157.7494 491875.4158
 515729.5536 517973.8446 502330.7526 510457.1944 478547.4298 486529.2102
 492172.817  534640.6444 511931.5452 494592.7388 499867.2602 482563.4972
 479777.025  476390.9154 510421.8406 504545.097  488800.3578 476835.3348
 508311.5218 505990.9972 517001.5882 505246.7634 507691.676  491094.7222
 501132.9168 508311.5218 508563.2496 515435.2584 505729.0808 486605.0724
 479040.7186 474289.1074]
```

Something is wrong with the type and I ran out of time to figure out how to fix it. I am sure it is a simple fix.

In [ ]:
```
#plotModelResults(rf, plot_intervals=True)
#plotCoefficients(rf)
```

2.6 Facebook Prophet

Implement the Facebook Prophet model for forecasting.

Data

In [ ]:
```
# Create a copy of te data and sort it
daily_sales4 = daily_sales2.copy(deep = True)
daily_sales4.sort_values(by=["time"]).head(n=3)
```

Out[ ]:

time	amount_usd_in_dollars
2033-01-01	4,415.40
2033-01-02	4,758.62
2033-01-03	6,283.53

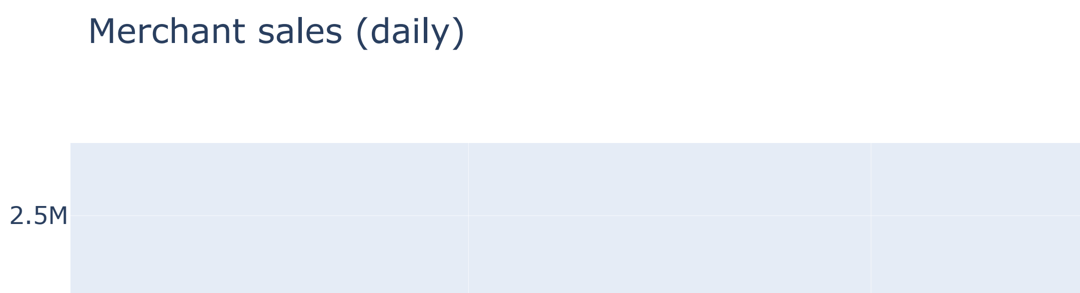
Exploratory visual analysis

Using plotly

In [ ]:
```
# Initialize plotly
init_notebook_mode(connected=True)
```

In [ ]:
```
def plotly_df(df, title=""):
    """Visualize all the dataframe columns as line plots."""
    common_kw = dict(x=df.index, mode="lines")
    data = [go.Scatter(y=df[c], name=c, **common_kw) for c in df.columns]
    layout = dict(title=title)
    fig = dict(data=data, layout=layout)
    iplot(fig, show_link=False)
```

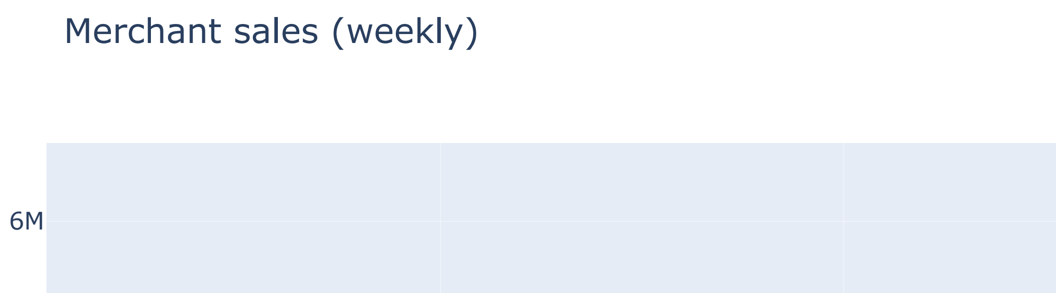
In [ ]:
```
plotly_df(daily_sales4, title="Merchant sales (daily)")
```



Going from daily to weekly.

```
In [ ]: weekly_sales1 = daily_sales4.resample("W").apply(sum)
```

```
In [ ]: plotly_df(weekly_sales1, title="Merchant sales (weekly)")
```



Making a forecast

In [ ]:
```
df = daily_sales4.reset_index()
df.columns = ["ds", "y"]
# converting timezones (issue https://github.com/facebook/prophet/issues/831)
# Caused an error, this data has not time, so there is no timezone.
# df["ds"] = df["ds"].dt.tz_convert(None)
df.tail(n=3)
```

Out[ ]:

	ds	y
727	2034-12-29	984,672.08
728	2034-12-30	668,559.95
729	2034-12-31	208,533.08

In [ ]:
```
prediction_size = 30
train_df = df[:-prediction_size]
train_df.tail(n=3)
```

Out[ ]:

	ds	y
697	2034-11-29	862,086.23
698	2034-11-30	917,309.40
699	2034-12-01	1,554,456.47

In [ ]:
```
m = Prophet()
m.fit(train_df)
```

```
14:55:47 - cmdstanpy - INFO - Chain [1] start processing
14:55:47 - cmdstanpy - INFO - Chain [1] done processing
```

Out[ ]: <prophet.forecaster.Prophet at 0x1382101ac30>

In [ ]:
```
future = m.make_future_dataframe(periods=prediction_size)
future.tail(n=3)
```

Out[ ]:

	ds
727	2034-12-29
728	2034-12-30
729	2034-12-31

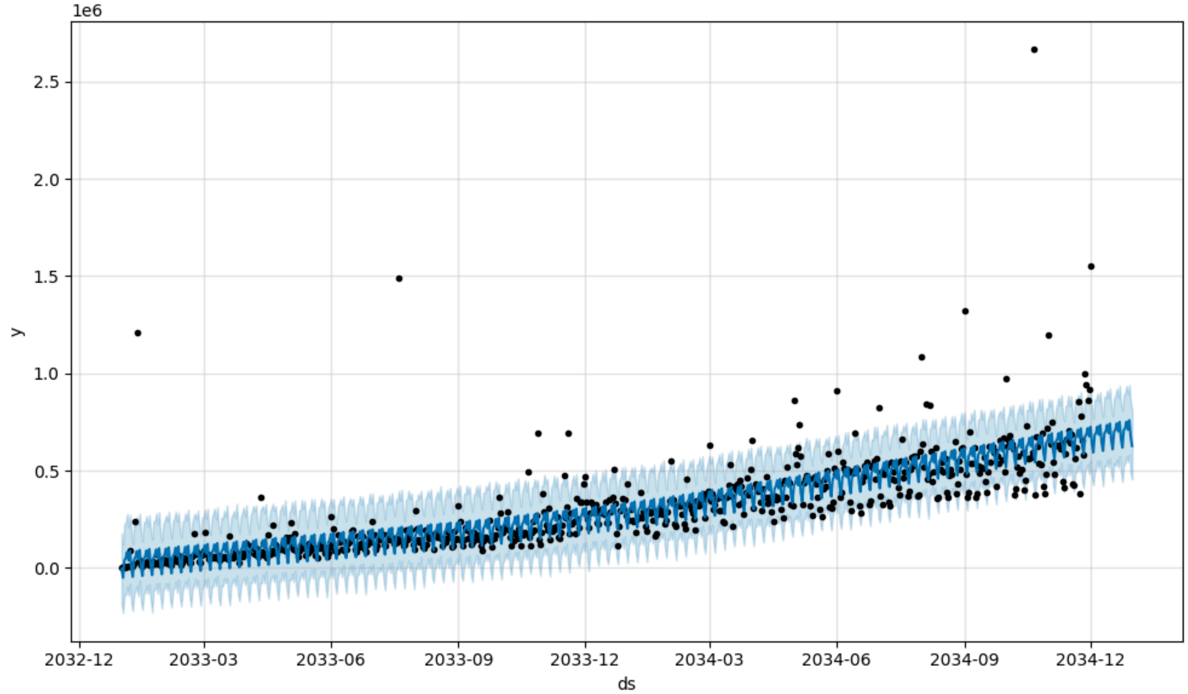
In [ ]:
```
forecast = m.predict(future)
forecast.tail(n=3)
```

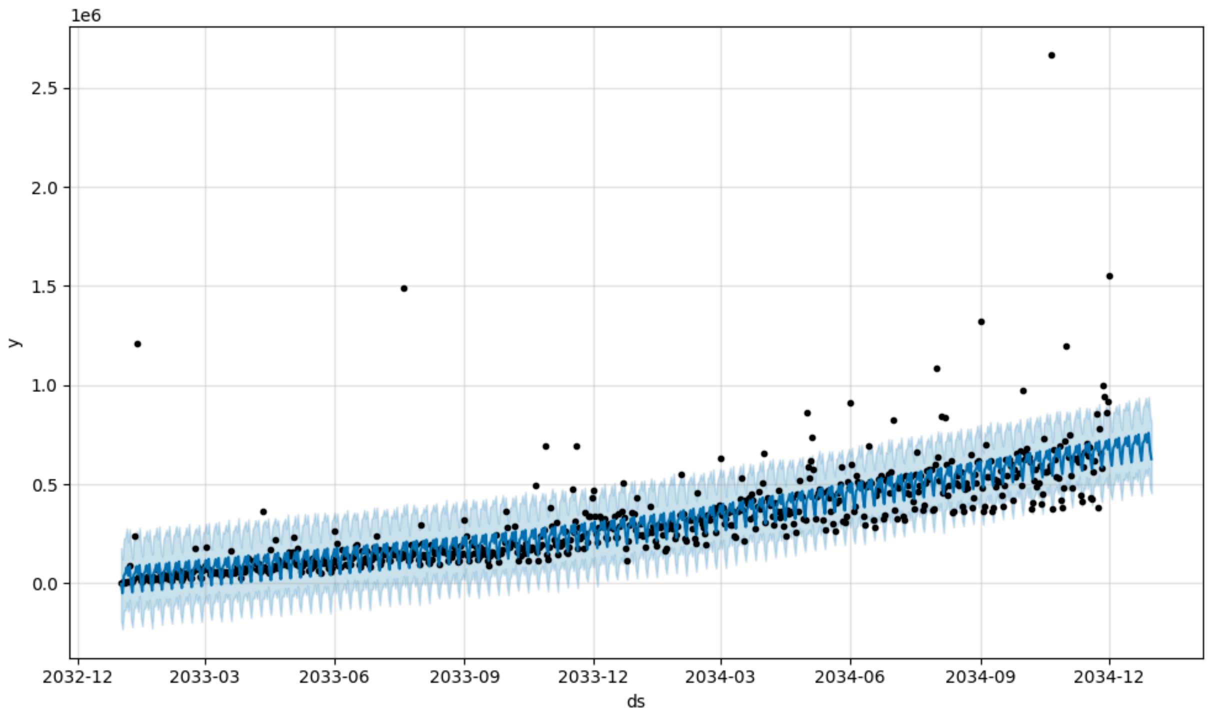
Out[ ]:

	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms
727	2034-12-29	708,074.62	582,106.00	939,706.39	707,675.52	708,588.90	50,741.13
728	2034-12-30	709,275.08	480,595.38	848,881.41	708,862.05	709,805.53	-36,958.02
729	2034-12-31	710,475.54	455,463.85	812,504.86	710,039.24	711,034.38	-83,405.13

In [ ]:
```
m.plot(forecast)
```

Out[ ]:



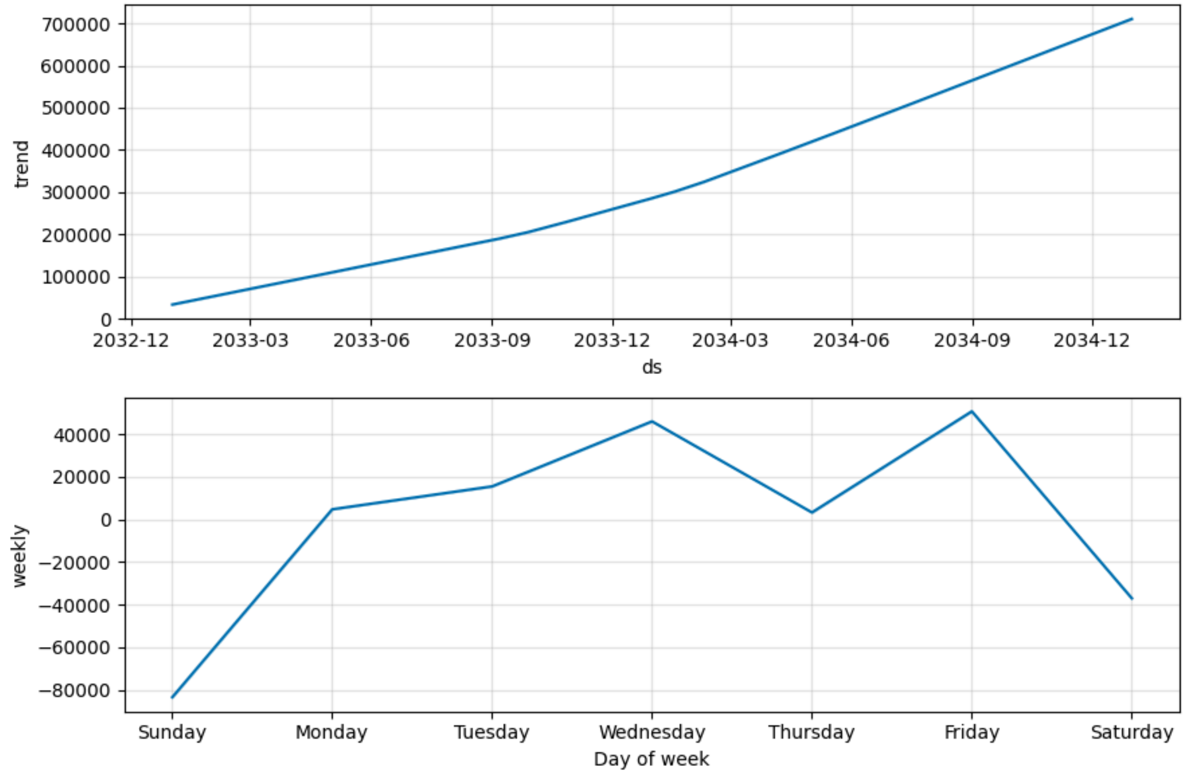


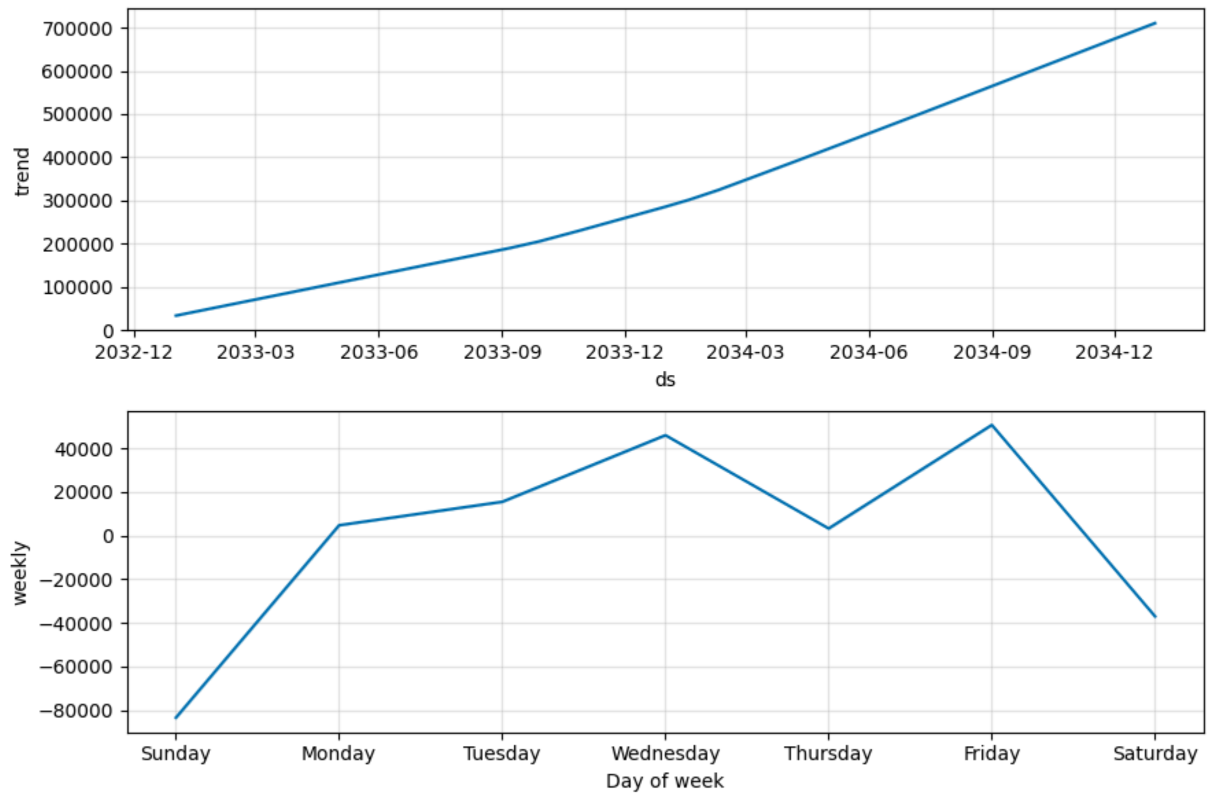
I am not sure why the graph is printing twice.

In [ ]:

```
m.plot_components(forecast)
```

Out[ ]:





Again, the graphs are printing twice but there is no monthly plot.

Forecast quality evaluation

In [ ]:

```
print(", ".join(forecast.columns))
```

```
ds, trend, yhat_lower, yhat_upper, trend_lower, trend_upper, additive_terms, additiv
e_terms_lower, additive_terms_upper, weekly, weekly_lower, weekly_upper, multiplicat
ive_terms, multiplicative_terms_lower, multiplicative_terms_upper, yhat
```

In [ ]:
```
def make_comparison_dataframe(historical, forecast):
    """Join the history with the forecast.

       The resulting dataset will contain columns 'yhat', 'yhat_lower', 'yhat_upper
    """
    return forecast.set_index("ds")[["yhat", "yhat_lower", "yhat_upper"]].join(
        historical.set_index("ds")
    )
```

In [ ]:
```
cmp_df = make_comparison_dataframe(df, forecast)
cmp_df.tail(n=3)
```

Out[ ]:

ds	yhat	yhat_lower	yhat_upper	y
2034-12-29	758,815.75	582,106.00	939,706.39	984,672.08
2034-12-30	672,317.07	480,595.38	848,881.41	668,559.95
2034-12-31	627,070.41	455,463.85	812,504.86	208,533.08

In [ ]:
```
def calculate_forecast_errors(df, prediction_size):
    """Calculate MAPE and MAE of the forecast.

       Args:
           df: joined dataset with 'y' and 'yhat' columns.
           prediction_size: number of days at the end to predict.
    """

    # Make a copy
    df = df.copy()

    # Now we calculate the values of e_i and p_i according to the formulas given in
    df["e"] = df["y"] - df["yhat"]
    df["p"] = 100 * df["e"] / df["y"]

    # Recall that we held out the values of the last `prediction_size` days
    # in order to predict them and measure the quality of the model.

    # Now cut out the part of the data which we made our prediction for.
    predicted_part = df[-prediction_size:]

    # Define the function that averages absolute error values over the predicted pa
    error_mean = lambda error_name: np.mean(np.abs(predicted_part[error_name]))

    # Now we can calculate MAPE and MAE and return the resulting dictionary of erro
    return {"MAPE": error_mean("p"), "MAE": error_mean("e")}
```

In [ ]:
```
for err_name, err_value in calculate_forecast_errors(cmp_df, prediction_size).items
    if err_name == "MAPE":
```

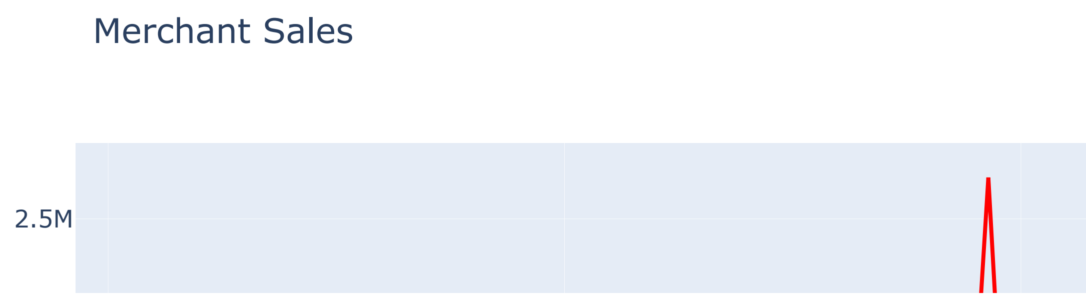
```
        MAPE_PR = err_value
    print(err_name, err_value)
```

```
MAPE 23.85228148556473
MAE 112080.80453769096
```

As a result, the relative error of our forecast (MAPE) is about 23.9%, and on average our model is wrong by ~$112,080 in sales (MAE).

Visualization

In [ ]:
```
def show_forecast(cmp_df, num_predictions, num_values, title):
    """Visualize the forecast."""

    def create_go(name, column, num, **kwargs):
        points = cmp_df.tail(num)
        args = dict(name=name, x=points.index, y=points[column], mode="lines")
        args.update(kwargs)
        return go.Scatter(**args)

    lower_bound = create_go(
        "Lower Bound",
        "yhat_lower",
        num_predictions,
        line=dict(width=0),
        marker=dict(color="gray"),
    )
    upper_bound = create_go(
        "Upper Bound",
        "yhat_upper",
        num_predictions,
        line=dict(width=0),
        marker=dict(color="gray"),
        fillcolor="rgba(68, 68, 68, 0.3)",
        fill="tonexty",
    )
    forecast = create_go(
        "Forecast", "yhat", num_predictions, line=dict(color="rgb(31, 119, 180)")
    )
    actual = create_go("Actual", "y", num_values, marker=dict(color="red"))

    # In this case the order of the series is important because of the filling
    data = [lower_bound, upper_bound, forecast, actual]

    layout = go.Layout(yaxis=dict(title="Posts"), title=title, showlegend=False)
    fig = go.Figure(data=data, layout=layout)
    iplot(fig, show_link=False)


show_forecast(cmp_df, prediction_size, 100, "Merchant Sales")
```



The prediction looks good. But there are a few outliers in that the actual data is outside the confidence interval for the prediction.

Box-Cox Transformation

In [ ]:
```
def inverse_boxcox(y, lambda_):
    return np.exp(y) if lambda_ == 0 else np.exp(np.log(lambda_ * y + 1) / lambda_)
```

In [ ]:
```
train_df2 = train_df.copy().set_index("ds")
```

In [ ]:
```
train_df2["y"], lambda_prophet = stats.boxcox(train_df2["y"])
train_df2.reset_index(inplace=True)
```

In [ ]:
```
m2 = Prophet()
m2.fit(train_df2)
future2 = m2.make_future_dataframe(periods=prediction_size)
forecast2 = m2.predict(future2)
```

```
14:55:49 - cmdstanpy - INFO - Chain [1] start processing
14:55:49 - cmdstanpy - INFO - Chain [1] done processing
```

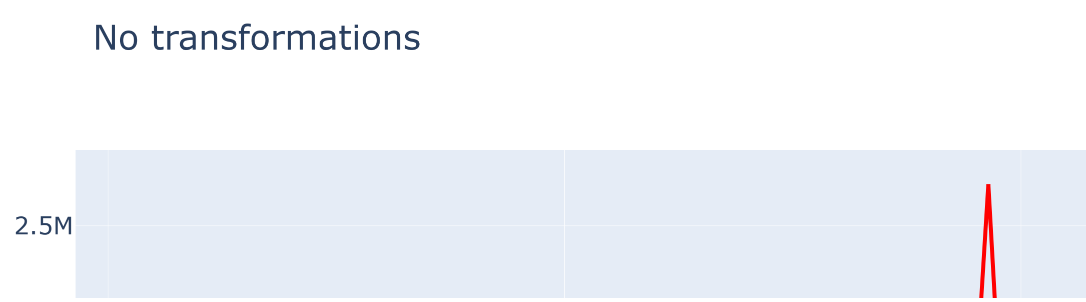
In [ ]:
```
for column in ["yhat", "yhat_lower", "yhat_upper"]:
    forecast2[column] = inverse_boxcox(forecast2[column], lambda_prophet)
```

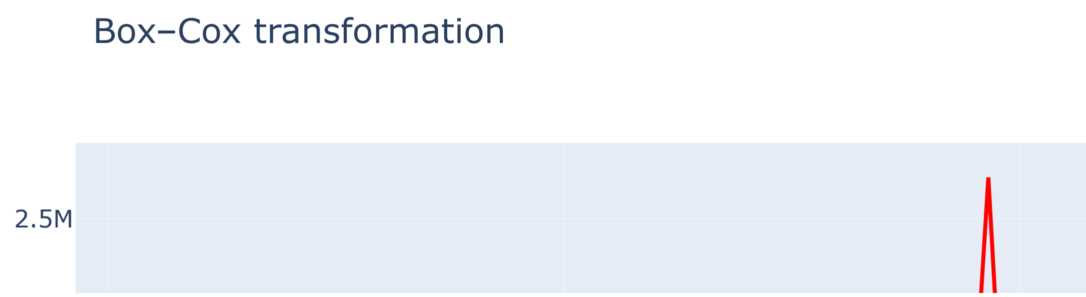
In [ ]:
```
cmp_df2 = make_comparison_dataframe(df, forecast2)
for err_name, err_value in calculate_forecast_errors(cmp_df2, prediction_size).item
    if err_name == "MAPE":
        MAPE_PRBC = err_value
    print(err_name, err_value)
```

```
MAPE 22.437301513183776
MAE 106060.68851355634
```

As a result, the relative error of our forecast (MAPE) is about 22.4%, and on average our model is wrong by ~ $106,060 in sales (MAE). This is slightly better than without the BOX-COX transforma$
112,080 in sales (MAE).

In [ ]:
```
show_forecast(cmp_df, prediction_size, 100, "No transformations")
show_forecast(cmp_df2, prediction_size, 100, "Box-Cox transformation")
```





There is definitely improvement but not a tremendous improvement.

3 Model Validation

3.1 Mean Absolute Percentage Error (MAPE)

Calculate and compare the MAPE for each model to assess their accuracy.

In [ ]:
```
print(f"The MAPE for Holt Winters is {MAPE_HW}.")
print(f"The MAPE for Linear Regression is {MAPE_LR}.")
print(f"The MAPE for Linear Regression including weekend / weekday is {MAPE_LRS}.")
print(f"The MAPE for Random Forest is {MAPE_RF}.")
print(f"The MAPE for Facebook Prophet is {MAPE_PR}.")
print(f"The MAPE for Facebook Profit with Box-Cox Transformation is {MAPE_PRBC}.")
```

```
The MAPE for Holt Winters is 2.8595375728115715.
The MAPE for Linear Regression is 0.20939886824567627.
The MAPE for Linear Regression including weekend / weekday is 0.20237243649488235.
The MAPE for Random Forest is 0.0.
The MAPE for Facebook Prophet is 23.85228148556473.
The MAPE for Facebook Profit with Box-Cox Transformation is 22.437301513183776.
```

I do not believe that the Random Forest result is correct. Of the rest it would seem that the Linear Regression including weekend / weekday is the best. But it only seems to be a slight margin. Without anything else, I would use it.

3.2 Confidence Intervals and Anomaly Detection

Plot the confidence intervals for each model's forecasts and identify any anomalies

I have included these above. I ran into an issue with the Random Forest and could not solve it.

4 Report Writing

- Summarize the findings from each model, including their performance metrics and insights.
- Discuss the implications of the results and suggest the most suitable model for forecasting merchant transactions.

I have included comments with each model. It seems to me that the Linear Regression including weekend and weekday is the best model but only by a small amount.