SARIMAX Forecasting

of a

Boarder Crossing Data Set

Inhaltsverzeichnis

1. Data set description

Context

The Bureau of Transportation Statistics (BTS) Border Crossing Data provide summary statistics for inbound crossings at the U.S.-Canada and the U.S.-Mexico border at the port level. Data are available for trucks, trains, containers, buses, personal vehicles, passengers, and pedestrians. Border crossing data are collected at ports of entry by U.S. Customs and Border Protection (CBP). The data reflect the number of vehicles, containers, passengers or pedestrians entering the United States. CBP does not collect comparable data on outbound crossings. Users seeking data on outbound counts may therefore want to review data from individual bridge operators, border state governments, or the Mexican and Canadian governments.

The data set is accessible at: https://www.kaggle.com/datasets/akhilv11/border-crossing-entry-data.

Content

COVERAGE: Incoming vehicle, container, passenger, and pedestrian counts at U.S.-Mexico and U.S.-Canada land border ports.

DEFINITIONS:

1. Bus Crossings: Number of arriving buses at a particular port, whether or not they are carrying passengers.

2. Container: A Container is defined as any conveyance entering the U.S. used for commercial purposes, either full or empty. Includes containers moving in-bond for the port initiating the bonded movements.

3. Types of Containers: The following are examples of a Container: Stakebed truck, truck with a car carrier, van, pickup truck/car, flatbed truck, piggyback truck with two linked trailers/containers = 2 containers, straight truck, bobtail truck, railcar, rail flatbed car stacked with four containers = 4 containers (on each rail car if there is multiple box containers count each container and the flatbed car.), and tri-level boxcar with multiple containers inside = 3 containers

4. Passengers Crossing In Buses: Number of persons arriving by bus requiring U.S. Customs and Border Protection (CBP) processing.

5. Passengers Crossing In Privately Owned Vehicles: Persons entering the United States at a particular port by private automobiles, pick-up trucks, motorcycles, recreational vehicles, taxis, ambulances, hearses, tractors, snowmobiles and other motorized private ground vehicles.

6. Passengers Crossing In Trains: Number of passengers and crew arriving by train and requiring CBP processing.

7. Pedestrian Crossings: The number of persons arriving on foot or by certain conveyance (such as bicycles, mopeds, or wheel chairs) requiring CBP processing.

8. Privately Owned Vehicle Crossings: Number of privately owned vehicles (POVs) arriving at a particular port. Includes pick-up trucks, motorcycles, recreational vehicles, taxis, snowmobiles, ambulances, hearses, and other motorized private ground vehicles.

9. Rail Container Crossings (loaded and empty): A container is any conveyance entering the U.S. used for commercial purposes, full or empty. In this case, it is the number of full or empty rail containers arriving at a port. This series includes containers moving as inbound shipments.

10. Train Crossings: Number of arriving trains at a particular port.

11. Truck Container Crossings (loaded and empty): A container is any conveyance entering the U.S. used for commercial purposes, full or empty. In this case, it is the number of full or empty truck containers arriving at a port. This series includes containers moving as inbound shipments.

12. Truck Crossings: Number of arriving trucks; does not include privately owned pick-up trucks.

Notes

Canada:
The ports of entry at Noyes, Minnesota and Whitetail, Montana closed in June 2006 and January 2013, respectively.

1. Incoming Trucks, Incoming PVs, PV Passengers, Incoming Buses, Bus Passengers, and Incoming Pedestrians

o Bar Harbor and Portland, Maine (ferry crossing) - Ferries arrived from May to September. The Bar Harbor, Maine to Yarmouth, Nova Scotia ferry is no longer in operation.

o Anacortes and Friday Harbor - The international ferries that connect Anacortes and Friday Harbor, Washington with Sidney, British Columbia do not run in February. Truck Containers (Loaded) and Truck Containers (Unloaded)

o Passenger vehicle and passengers in personal vehicles data for Cape Vincent, New York (ferry) are available beginning in 2007. The ferry between Wolfe Island (Canada) and Cape Vincent does not operate in the winter.

2. Incoming Train Passengers

o Includes both passengers and crew.

o Starting with November 2017, Maine officials restrict international bridge traffic to passenger vehicles only.

Mexico:
Data for the port of Calexico are reported as a combined total with Calexico East.

1. Incoming Trucks:
o Data represent the number of truck crossings, not the number of unique vehicles, and include both loaded and unloaded trucks.

2. Incoming Train Passengers:
o Includes train crew. BTS is not aware of any passenger service currently operating across the U.S.-Mexico Border.

o CBP has indicated to BTS that since 2009 train crew are being exchanged at the Texas-Mexico border, and thus do not enter the United States.

2. Data preprocessing

Main question: Forecast the monthly number of boarder crossings between USA and Canada.

STEP1: We extract via DuckDB the relevant portion of the data set by means of the following list of SQL queries:

```
query1 = """
    CREATE TABLE IF NOT EXISTS AllData AS
    SELECT * FROM read_csv('Border_Crossing_Entry_Data.csv')
    """
# Select the first 7 columns of the AllData table and store them into a new table "RelevantData"
query2 = """
    CREATE TABLE IF NOT EXISTS RelevantData AS
    SELECT #1, #2, #3, #4, #5, #6, #7
    FROM AllData
    """

# Select the first 7 columns of the "RelevantData" table and store them into a new table
"US_Canada_Data"
query3 = """
    CREATE TABLE IF NOT EXISTS US_Canada_Data AS
    SELECT #1, #2, #3, #4, #5, #6, #7
    FROM RelevantData
    WHERE Border = 'US-Canada Border'
    """

# Select the first 7 columns of the "RelevantData" table and store them into a new table
"US_Mexico_Data"
query4 = """
    CREATE TABLE IF NOT EXISTS US_Mexico_Data AS
    SELECT #1, #2, #3, #4, #5, #6, #7
    FROM RelevantData
    WHERE Border = 'US-Mexico Border'
```

```
    """
# Describe the schema of the US_Canada_Data table
query5 = "DESCRIBE US_Canada_Data"

# Convert a string date column to date format with a CTE
query6 = """
CREATE TABLE IF NOT EXISTS US_Canada_Data_Altered AS
SELECT
    *,
    strptime(concat('01 ', Date), '%d %b %Y') AS converted_date
FROM
    US_Canada_Data
"""

# Select the entire altered table US_Canada_Data_Altered
query7 = "SELECT * FROM US_Canada_Data_Altered"

# Describe the altered table US_Canada_Data_Altered
query8 = "DESCRIBE US_Canada_Data_Altered"

# Order the table US_Canada_Data_Altered based on the converted_date
# column in ascending manner
query9 = """
SELECT *
FROM US_Canada_Data_Altered
ORDER BY converted_date ASC
"""

# Ctreate a US_Canada_AggregatedData table
query10 = """
-- Step 1: Drop the table if it exists
DROP TABLE IF EXISTS US_Canada_AggregatedData;

-- Step 2: Create the new table and populate it with aggregated data
CREATE TABLE IF NOT EXISTS US_Canada_AggregatedData AS
SELECT
    converted_date,
    AVG(Value) AS avg_value_month,
    SUM(Value) AS total_value_month
FROM
    US_Canada_Data_Altered
GROUP BY
    converted_date
ORDER BY
    converted_date ASC;
"""

# select the aggregated table:
query11 = """
SELECT *
```

FROM US_Canada_AggregatedData
"""

These queries lead to the following table structure:

The aggregated time stamp ordered US_Canada_AggregatedData table:

converted_date timestamp	avg_value_month double	total_value_month int128
1996-01-01 00:00:00	10141.905982905982	9492824
1996-02-01 00:00:00	10653.472222222223	9971650
1996-03-01 00:00:00	11533.17628205128 1	10795053
1996-04-01 00:00:00	12576.271367521367	11771390
1996-05-01 00:00:00	14216.83226495726 6	13306955
1996-06-01 00:00:00	15745.551282051281	14737836
1996-07-01 00:00:00	17928.80876068376	16781365
1996-08-01 00:00:00	19217.96153846154	17988012
1996-09-01 00:00:00	14730.805555555555	13788034
1996-10-01 00:00:00	13659.291666666666	12785097
·	·	·
·	·	·
·	·	·
2024-03-01 00:00:00	12206.027675276753	6615667
2024-04-01 00:00:00	11444.026737967915	6420099
2024-05-01 00:00:00	13015.939609236235	7327974
2024-06-01 00:00:00	13756.374564459931	7896159
2024-07-01 00:00:00	16412.471304347826	9437171
2024-08-01 00:00:00	17274.27719298245 6	9846338
2024-09-01 00:00:00	12909.131715771231	7448569
2024-10-01 00:00:00	12736.114235500878	7246849
2024-11-01 00:00:00	11614.872072072072	6446254
2024-12-01 00:00:00	11762.11743772242	6610310
348 rows (20 shown)		3 columns

We convert the SQL table into a data frame via

„US_Canada_Data_Aggregated_df = db.sql("SELECT * FROM US_Canada_AggregatedData").df()“,

leading to:

First ten entries of the US_Canada_AggregatedData_df:

	converted_date	avg_value_month	total_value_month
0	1996-01-01	10141.905983	9492824.0
1	1996-02-01	10653.472222	9971650.0
2	1996-03-01	11533.176282	10795053.0
3	1996-04-01	12576.271368	11771390.0
4	1996-05-01	14216.832265	13306955.0
5	1996-06-01	15745.551282	14737836.0
6	1996-07-01	17928.808761	16781365.0
7	1996-08-01	19217.961538	17988012.0
8	1996-09-01	14730.805556	13788034.0
9	1996-10-01	13659.291667	12785097.0

STEP 2: We create an aggregate dictionare from the extracted pandas DataFrame containing averaged numbers of boarder crossings for each month between January 1996 and December 2024 via:

```
AggregateDictionary = dict()
for iterYear in UniqueYears:
    MeasuresYear = []
    for iterMonth in UniqueMonths:
        condition1 = US_Canada_Data_df['Year'] == iterYear
        condition2 = US_Canada_Data_df['Month'] == iterMonth
        Temp_df = US_Canada_Data_df[condition1 & condition2]
        MeasuresYear.append(Temp_df['Value'].sum())
        AggregateDictionary[iterYear] = MeasuresYear

# sort dictionary according to its keys
myKeys = list(AggregateDictionary.keys())
myKeys.sort()

# Sorted Dictionary
sd = {i: AggregateDictionary[i] for i in myKeys}
AggregateDictionary = sd
print(AggregateDictionary)
```

This leads us to the following Data Frame structure:

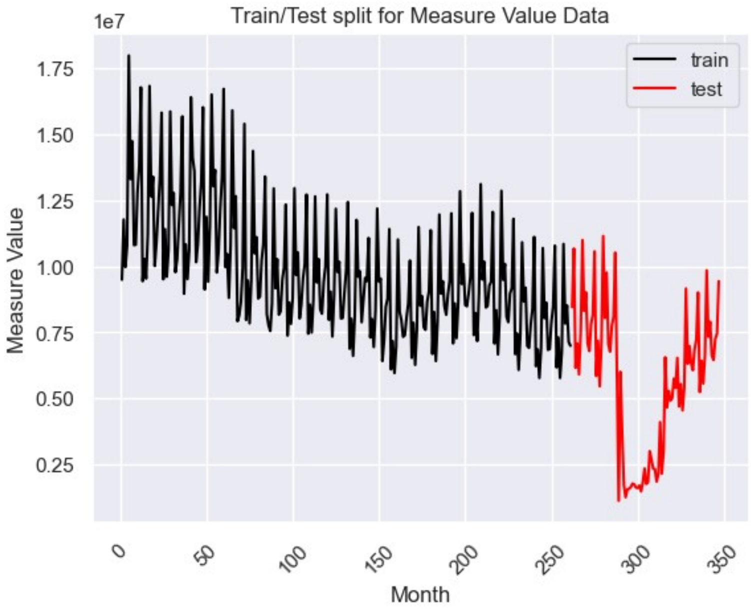
c

```
AggregateDF = pd.DataFrame.from_dict(AggregateDictionary)
print(AggregateDF)
```

```
        1996      1997      1998      1999      2000      2001      2002  \
0    9492824   9436325   9509603   8963032   9120445   9968711   7967046
1   11771390  10277874  11408750  10827952  11872809  10463283   9458774
2    9971650   9533652   9606589   9515913   9417181   8803012   7839425
3   10795053  11213563  10622465  10607585  11557273  10788933   9484491
4   17988012  16834126  15867936  16408819  16509750  15909159  14371954
5   13306955  12645355  12321480  14126950  13035044  11449120  10487686
6   14737836  13397632  12780630  13600774  13653957  12655972  11102595
7   10801524  10014597   9776730  10157514   9766290   7919578   8780240
8   10819163  10903931  10270735  10842100  10567692   8136859   8850342
9   12785097  12086771  11607099  12209136  12196150   8656311  10238366
10  13788034  13345769  12836779  13211576  13369583   9942374  10848047
11  16781365  15823115  15682994  16025977  16723499  15399079  13403729

        2003      2004      2005  ...      2015      2016      2017      2018  \
0    8186226   7373490   7447288  ...   6671807   6215665   6168767   6151874
1    7766297   8626328   8544541  ...   7455737   6847059   7300505   7073413
2    7549888   7821507   7492452  ...   6074264   5767485   5768250   5901475
3    8578623   8799951   8724428  ...   7421462   6913898   6749117   7561801
4   12952628  12960841  12649601  ...  10908827  10692018  10846607  10993925
5    9168311   9669448   9386894  ...   8664906   8033685   7830360   8307472
6   10273715  10525294  10277161  ...   9188601   8624151   8515363   9015529
7    8167744   8023030   8364507  ...   6981127   6833933   7131453   7070154
8    8295872   8550079   8205193  ...   6909179   6864575   6987074   6788073
9    9499606   9547455   9409051  ...   7799362   7890198   7843460   7813441
10   9959044  10080151  10019526  ...   8369654   8487012   8454019   8172138
11  12340460  12718048  12723059  ...  11111636  10782099  10673299  10564665

        2019     2020     2021     2022     2023     2024
0    5841881  5895552  1596316  2126121  4685610  5235491
1    7175976  1110625  1688595  4091393  5534787  6420099
2    5451377  6000979  1465675  2140893  4536450  5553414
3    7098470  3703717  1818274  2992770  5373629  6615667
4   11146471  1714174  2333125  6550339  9156150  9846338
5    8039300  1242037  1748625  4648165  6305656  7327974
6    9762713  1540712  1825042  5273546  6978141  7896159
7    7058135  1568631  2989342  4908379  6324285  6610310
8    6763885  1624869  2674041  4999946  6059394  6446254
9    7733413  1762684  2327083  5741597  6855550  7246849
10   8070939  1730368  2312496  5388875  7226765  7448569
11  10519310  1612690  1838438  6527388  9007752  9437171

[12 rows x 29 columns]
```

STEP 3: We perform an 80-20 train-test split of the aggregated DataFrame via:

```
# create data set lists for training and testing
train = DataSet[:int(0.75*len(DataSet))]
test = DataSet[int(0.75*len(DataSet))+1:]

months1 = [(month+1) for month in range(len(train))]
months2 = [(month+1) for month in range(len(train),len(train)+len(test))]

plt.plot(months1,train, color = "black")
plt.plot(months2,test, color = "red")
plt.ylabel('Measure Value')
plt.xlabel('Month')
plt.xticks(rotation=45)
plt.title("Train/Test split for Measure Value Data")
plt.legend(['train', 'test'])
plt.show()
```



The test portion of the data set present a Covid-induced time anomaly in boarder crossings. This implies that we may have to expect our algorithm to perform suboptimally, however this split allows us to test the customized and automated forecasting performance of the cross-validated SARIMAX model under more „strained“ conditions.

3. SARIMAX evaluation

We trigger a customized Python function „SARIMAX_grid_search“ and obtain the following optimal SARIMAX parameters (seasonality s is set to 12) via

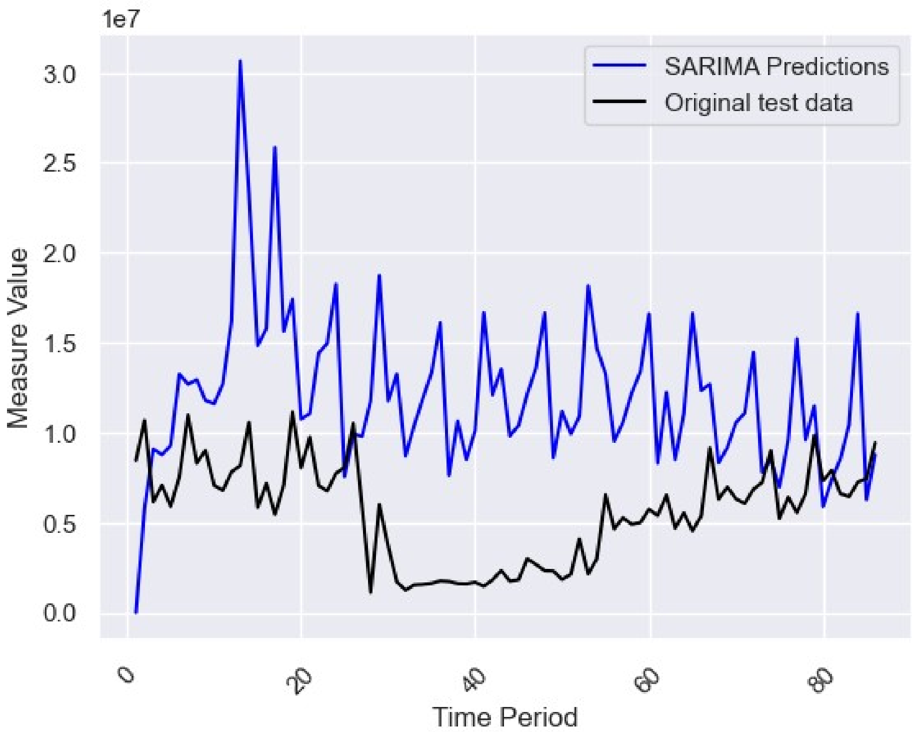
```python
# start the grid search
start_time = time.time()
best_params = SARIMAX_grid_search(y, 12)
print("--- Duration: %s seconds ---" % (time.time() - start_time))
```

```
Tuned SARIMAX Parameters: (2, 1, 2, 2, 2, 2, 12)
Best average rmse score is 3622051.7420007405
   Time Period  True Values  Predicted Values  Confidence Lower  \
0            0      7690676      7.876369e+06      4.669138e+06
1            1      7594110      7.503923e+06      4.259713e+06
2            2      8669030      7.609453e+06      4.243794e+06
3            3      9048099      8.223683e+06      4.826908e+06
4            4     11366681      8.724421e+06      5.278799e+06
5            5      6677975      1.023144e+07      6.749724e+06
6            6      8265991      6.450796e+06      2.929420e+06
7            7      6409080      8.117489e+06      4.558953e+06
8            8      7938922      6.341962e+06      2.745789e+06
9            9     11961504      7.836648e+06      4.203646e+06

   Confidence Upper
0      1.108360e+07
1      1.074813e+07
2      1.097511e+07
3      1.162046e+07
4      1.217004e+07
5      1.371316e+07
6      9.972171e+06
7      1.167602e+07
8      9.938135e+06
9      1.146965e+07
--- Duration: 190.60557293891907 seconds ---
```

```python
# print the optimal grid search parameters
print(best_params)
```

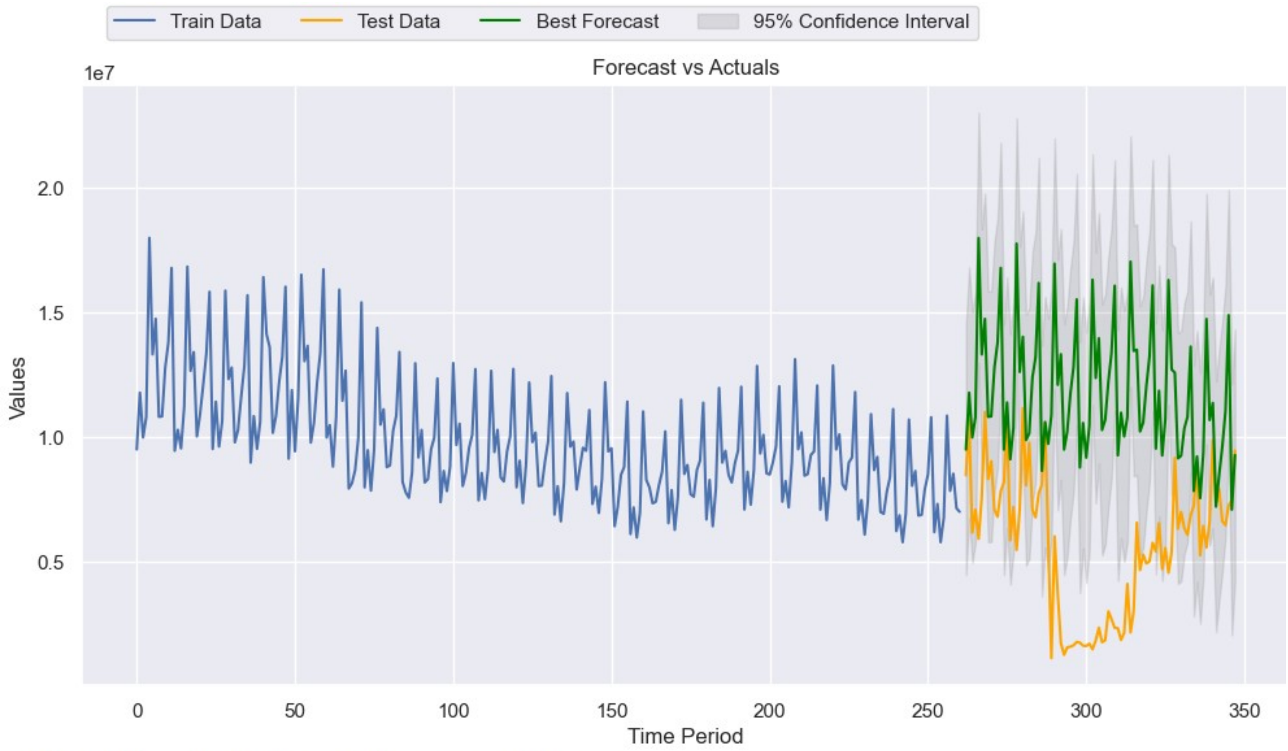
```
(2, 1, 2, 2, 2, 2, 12)
```

The average RMSE score indicates a lot of room for improvement, however due to the pronounced time series anomaly of the test portion within the extracted DataFrame it may be regarded as a solid „first guess“, as indicated by the corresponding forecast plot:



In the above plot „Measure Value“ denotes an aggregated number of monthly boarder crossings, whereas „Time Period“ refers to the number of months passed since the last training month.

Now we perform an automated cross-validated SARIMAX grid search by means of the Pythonic function „AutomatedSarimaxGridSearch“ and obtain the following results:



	Time Period	Predicted Values	Confidence Lower	Confidence Upper
0	0	9.487343e+06	4.452919e+06	1.452177e+07
1	1	1.176717e+07	6.732749e+06	1.680160e+07
2	2	9.970411e+06	4.935986e+06	1.500484e+07
3	3	1.079340e+07	5.758979e+06	1.582783e+07
4	4	1.798019e+07	1.294577e+07	2.301461e+07
5	5	1.330475e+07	8.270328e+06	1.833918e+07
6	6	1.473462e+07	9.700193e+06	1.976904e+07
7	7	1.080241e+07	5.767983e+06	1.583683e+07
8	8	1.081993e+07	5.785507e+06	1.585436e+07
9	9	1.278392e+07	7.749494e+06	1.781834e+07

--- Duration: 260.2104489803314 seconds ---

The ideal SARIMAX parameter set amounts to

\`p=4.0\`, \`d=1.0\`, \`q=2.0\`, \`P=2.0\`, \`D=2.0\`,\`Q=2.0\`,\`s=12\`

and leads to Mean Absolute Percentage Error (MAPE) of about 0.06 which indicates a significant forecasting improvement compared to the previous customized SARIMAX grid search.

Apparently, Pythonic SARIMAX modeling allows for robust and reliable trend predictions even in cases of highly anomalous time series trend patterns.