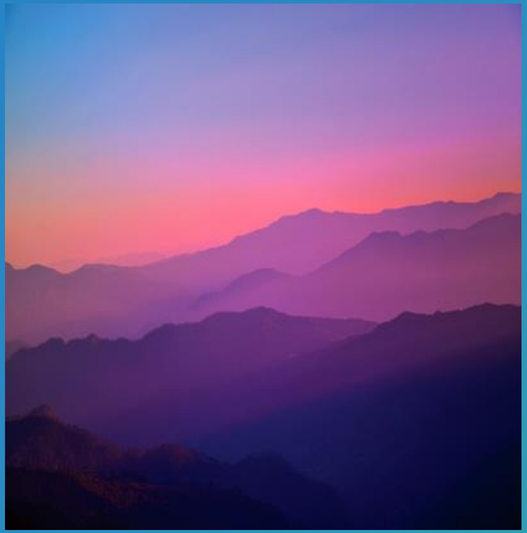
DATA SCIENCE FOR SUPPLY CHAIN FORECASTING

by Diego Beteta



Intro

Pasado

Empresas tecnológicas proporcionaron software de pronóstico estadístico, basados en suavización exponencial, que permitieron a las empresas utilizarlas como el pilar de sus procesos S&OP.

Presente

Debido al aumento de la potencia computacional, grandes conjuntos de datos, mejores modelos de pronóstico y disponibilidad de herramientas gratuitas, las empresas pueden ser cada vez más competitivas.

Con modelos de **Machine Learning (ML)** construidos en Python, es posible aportar a cualquier negocio más valor que cualquier software de pronóstico disponible en el mercado.

Futuro

Los 'demand planners' tendrán que aprender a trabajar con modelos de pronóstico avanzados basados en ML y podrán agregarles valor a medida que comprendan las deficiencias de ML.

Excel vs Python

Es necesario un cambio de paradigma para pasar aproximaciones manuales realizadas en Excel a modelos potentes automatizados en Python.

Python permite realizar cálculos en grandes conjuntos de datos de forma rápida y automatizada. Viene con muchas bibliotecas:

- Análisis de datos (pandas)
- Cálculos científicos (Numpy/SciPy)
- Machine Learning (scikit-learn)

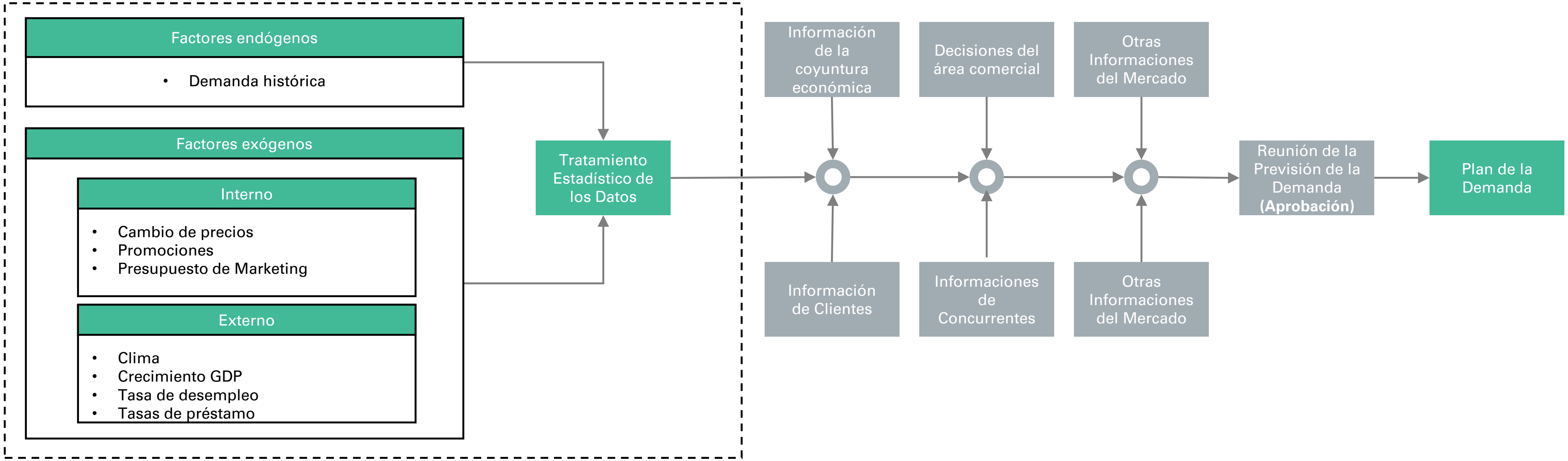
Beneficios de aplicar ‘Data Science’ en el pronóstico de la demanda:

- Escalabilidad
- Automatización
- Rentabilidad

“Si Excel es una navaja suiza, Python es un ejército completo de máquinas de construcción que esperan instrucciones de cualquier científico de datos.”

CONTENT

Application of Data Science in Demand Management



Old-School Statistics vs Machine Learning

Statistical Forecasting

- Permite ver y entender los patrones de la demanda (variación aleatoria, tendencia y estacionalidad)
- Sólo permite procesar un producto a la vez
- No permite evaluar la relación de datos externos con datos históricos de la demanda

Modelos de pronóstico

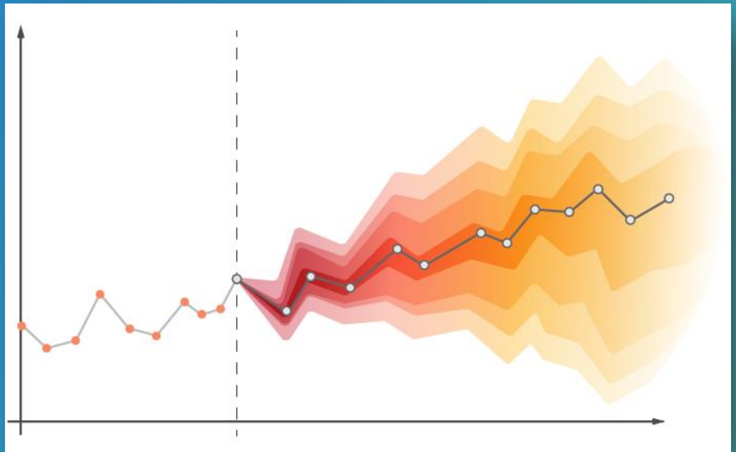
1. Moving Average
2. Exponential Smoothing
3. Double Exponential Smoothing
4. Double Exponential Smoothing with Damped Trend
5. Triple Exponential Smoothing
6. Tripple Additive Exponential Smoothing

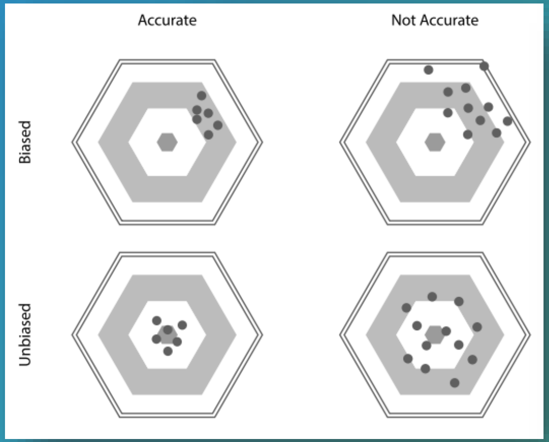
Machine Learning

- No proporciona ninguna explicación ni comprensión de los diferentes patrones de demanda
- Sólo se enfoca en obtener la respuesta correcta
- Permite procesar una gran cantidad de productos a la vez
- Permite evaluar la relación de datos externos con datos históricos de la demanda

Modelos de pronóstico

1. Linear Regression
2. Tree
3. Forest
4. Extremely Randomized Trees
5. Adaptative Boosting
6. Extreme Gradient Boosting





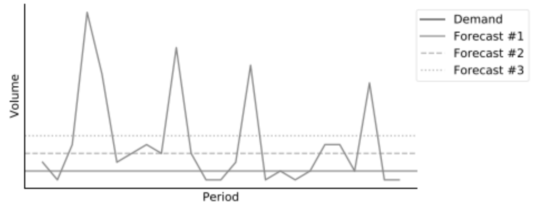


Figure 2.6: Demand and forecasts.

Table 2.1: KPI comparison.

	Forecast #1	Forecast #2	Forecast #3
Bias	−3.9	−1.9	**0.1**
MAPE	**64%**	109%	180%
MAE	4.4	**4.1**	4.8
RMSE	7.1	6.2	**5.9**

Forecast KPI

Definir la calidad del pronóstico mediante la evaluación de sus KPI es importante porque nos permite:

- Dimensionar la confiabilidad de cada técnica de pronóstico
- Mejorar la calidad de cada técnica mediante la optimización de sus parámetros
- Comparar diferentes técnicas optimizadas y definir las más adecuada para nuestros productos.
- Supervisar constantemente las técnicas utilizadas

Adaptabilidad

No existe un modelo de pronóstico perfecto que pueda vencer a cualquier otro modelo para cualquier tipo de negocio.

Adaptar los modelos de pronóstico a su conjunto de datos de demanda permitirá lograr un mejor nivel de precisión que mediante el uso de herramientas de caja negra.

Los modelos de Machine Learning deben adaptarse a los patrones de demanda de sus productos.

Forecast KPI

BIAS

Average Error

- El sesgo promedio de un pronóstico se define como su error promedio.
- Nos indica si, en promedio, subestimamos o sobrestimamos la demanda.
- Un valor relativamente bajo (%) indica que la técnica no tiene sesgo, no sobrestima ni subestima consistentemente la demanda.

MAPE

Mean Absolute Percentage Error

- Es especialmente útil cuando los valores de la demanda real son muy grandes.
- MAPE divide cada error individualmente por la demanda, por lo que está sesgado: un pronóstico extremadamente bajo sólo puede resultar en un error máximo de 100%, mientras que cualquier pronóstico demasiado alto no se limitará a un porcentaje de error específico (>100%)
- Debido a esto, la optimización de MAPE dará como resultado un pronóstico que muy probablemente subestime la demanda. ¡Evita usar MAPE!

MAE

Mean Absolute Error

- Mide la precisión del pronóstico al promediar las magnitudes de los errores absolutos del pronóstico de cada periodo de tiempo.
- Nos indica qué tan lejos, en promedio, están nuestros pronósticos de la demanda mediana.
- Un valor relativamente alto (%) indican que se tiene una pésima calidad de pronóstico.

RMSE

Root Mean Square Error

- RMSE le da más importancia a los errores más grandes, mientras que MAE le da la misma importancia a cada error.
- Nos indica qué tan lejos, en promedio, están nuestros pronósticos de la demanda promedio.
- Un gran error es suficiente para obtener un RMSE muy malo (alto %).

HANDS-ON

IMPORT, TRANSFORM AND FILTER DATA

dataset.csv

```
Year,Month,Product,Demand
2007,1,Product 1,2884
2007,1,Product 2,2521
2007,1,Product 3,1029
2007,1,Product 4,870
2007,1,Product 5,693
2007,1,Product 6,665
2007,1,Product 7,622
2007,1,Product 8,599
2007,1,Product 9,423
2007,1,Product 10,362
2007,1,Product 11,352
2007,1,Product 12,263
2007,1,Product 13,258
2007,1,Product 14,191
2007,1,Product 15,169
2007,1,Product 16,168
2007,1,Product 17,136
2007,1,Product 18,127
2007,1,Product 19,97
2007,1,Product 20,55
2007,1,Product 21,33
2007,1,Product 22,26
2007,1,Product 23,26
2007,1,Product 24,22
2007,1,Product 25,20
2007,1,Product 26,16
2007,1,Product 27,15
2007,1,Product 28,14
2007,1,Product 29,9
2007,1,Product 30,4
2007,1,Product 31,4
2007,1,Product 32,3
2007,1,Product 33,2
2007,1,Product 34,2
2007,1,Product 35,2
2007,1,Product 36,1
2007,1,Product 37,1
2007,1,Product 38,1
2007,2,Product 1,1885
2007,2,Product 2,1517
2007,2,Product 4,686
2007,2,Product 3,621
2007,2,Product 5,570
2007,2,Product 7,551
2007,2,Product 8,498
```

```python
# Import dataset to dataframe
def import_data():
    data = pd.read_csv('dataset.csv')
    return data

data = import_data()
display(data)
print('There are a total of', data['Product'].nunique(),
      'unique products in the dataset')
```

[166] ✓ 0.2s

	Year	Month	Product	Demand
0	2007	1	Product 1	2884
1	2007	1	Product 2	2521
2	2007	1	Product 3	1029
3	2007	1	Product 4	870
4	2007	1	Product 5	693
...	...	...	...	...
4372	2017	1	Product 34	3
4373	2017	1	Product 46	2
4374	2017	1	Product 48	1
4375	2017	1	Product 36	1
4376	2017	1	Product 37	1

4377 rows × 4 columns

There are a total of 66 unique products in the dataset

```python
# To filter the product
product = 'Product 40'
dataset = data[data['Product'] == product]
display(dataset)
```

[177] ✓ 0.6s

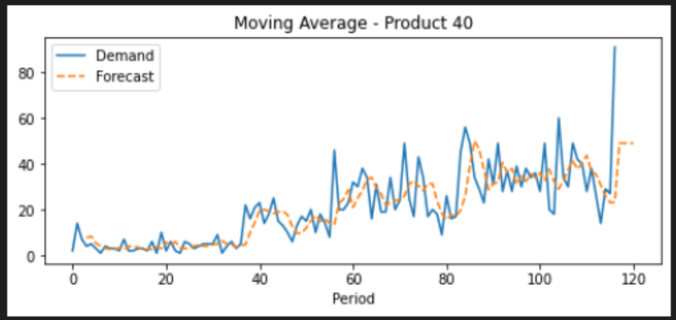
	Year	Month	Product	Demand
74	2007	2	Product 40	2
105	2007	3	Product 40	14
145	2007	4	Product 40	7
184	2007	5	Product 40	4
222	2007	6	Product 40	5
...	...	...	...	...
4229	2016	9	Product 40	26
4263	2016	10	Product 40	14
4299	2016	11	Product 40	29
4335	2016	12	Product 40	27
4363	2017	1	Product 40	91

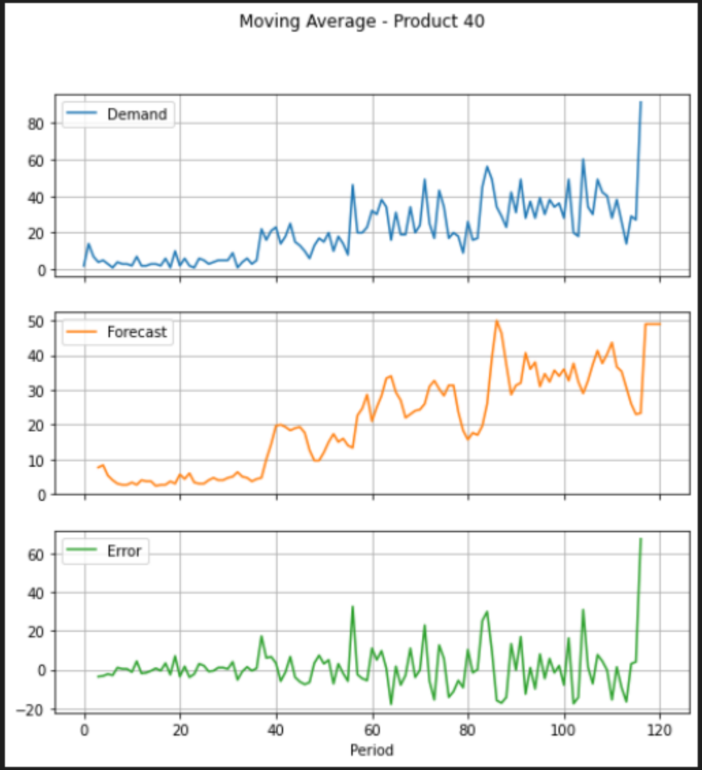
117 rows × 4 columns

HANDS-ON
1 MOVING AVERAGE

```
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = moving_average(d, extra_periods=4, n=3) # Call our new function
```

```
Moving Average Forecast - Product 40
        Demand   Forecast      Error
Period
0          2.0        NaN        NaN
1         14.0        NaN        NaN
2          7.0        NaN        NaN
3          4.0   7.666667  -3.666667
4          5.0   8.333333  -3.333333
...        ...        ...        ...
116       91.0  23.333333  67.666667
117        NaN  49.000000        NaN
118        NaN  49.000000        NaN
119        NaN  49.000000        NaN
120        NaN  49.000000        NaN

[121 rows x 3 columns]
Moving Average KPI - Product 40
Bias: 0.89, 4.27%
MAPE: 50.79%
MAE: 7.28, 34.87%
RMSE: 11.45, 54.88%
```



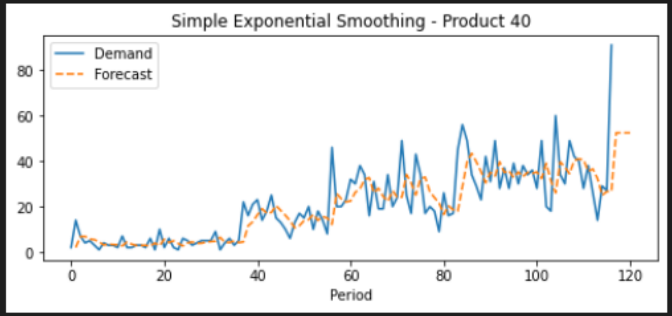


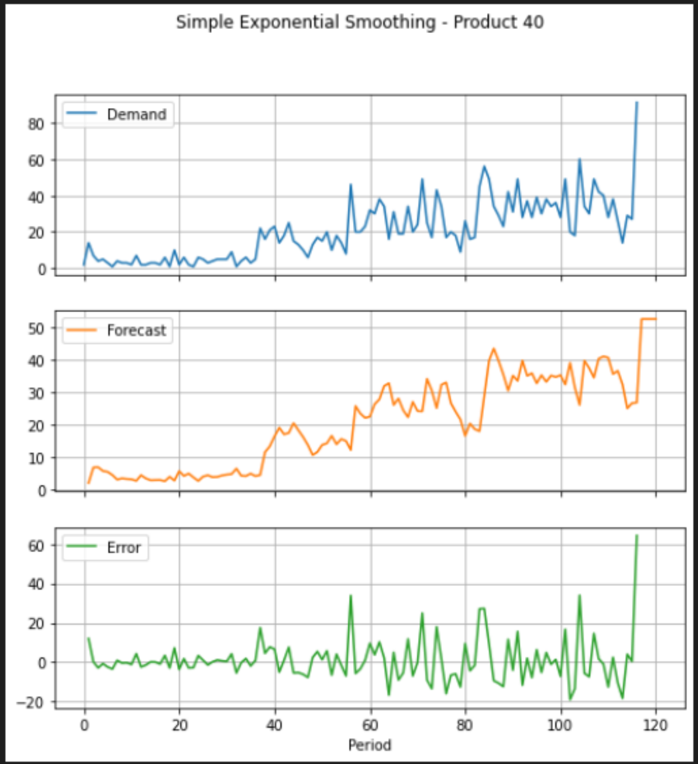
HANDS-ON

2 SIMPLE EXPONENTIAL SMOOTHING


```
d = dataset['Demand'].tolist() # Convert dataframe column to list
df = simple_exp_smooth(d, extra_periods=4, alpha=0.4)
```

```
Simple Exponential Smoothing Forecast - Product 40
        Demand   Forecast      Error
Period
0          2.0        NaN        NaN
1         14.0   2.000000  12.000000
2          7.0   6.800000   0.200000
3          4.0   6.880000  -2.880000
4          5.0   5.728000  -0.728000
...        ...        ...        ...
116       91.0  26.756136  64.243864
117        NaN  52.453682        NaN
118        NaN  52.453682        NaN
119        NaN  52.453682        NaN
120        NaN  52.453682        NaN

[121 rows x 3 columns]
Simple Exponential Smoothing KPI - Product 40
Bias: 1.09, 5.26%
MAPE: 50.94%
MAE: 7.23, 34.93%
RMSE: 11.27, 54.45%
```



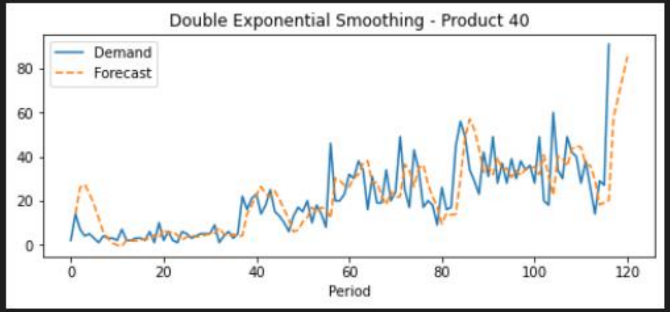


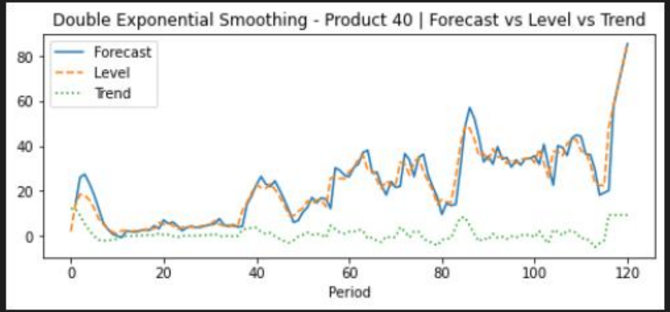
HANDS-ON

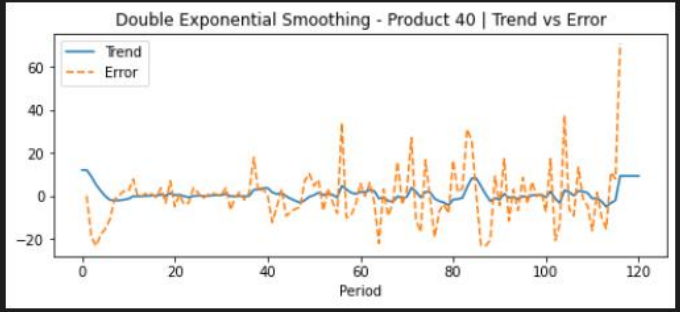
3 DOUBLE EXPONENTIAL SMOOTHING

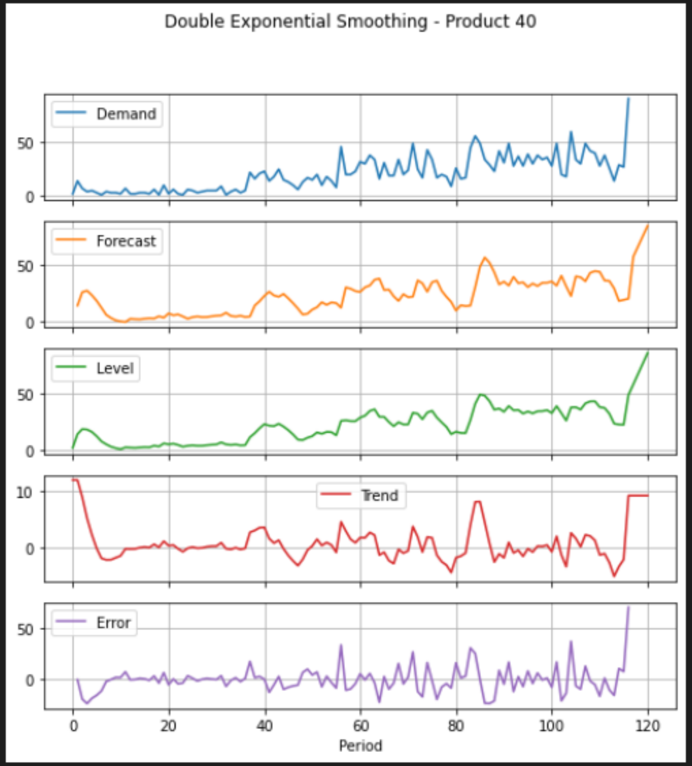
```
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = double_exp_smooth(d, extra_periods=4, alpha=0.4,
                       beta=0.4)
```

```
Double Exponential Smoothing Forecast - Product 40
        Demand   Forecast      Level      Trend      Error
Period
0          2.0        NaN   2.000000  12.000000        NaN
1         14.0  14.000000  14.000000  12.000000   0.000000
2          7.0  26.000000  18.400000   8.960000 -19.000000
3          4.0  27.360000  18.016000   5.222400 -23.360000
4          5.0  23.238400  15.943040   2.304256 -18.238400
...        ...        ...        ...        ...        ...
116       91.0  20.161385  48.496831   9.235834  70.838615
117        NaN  57.732665  57.732665   9.235834        NaN
118        NaN  66.968500  66.968500   9.235834        NaN
119        NaN  76.204334  76.204334   9.235834        NaN
120        NaN  85.440169  85.440169   9.235834        NaN

[121 rows x 5 columns]
Double Exponential Smoothing KPI - Product 40
Bias: -0.15, -0.72%
MAPE: 76.64%
MAE: 8.66, 41.86%
RMSE: 13.07, 63.19%
```









Level: es el valor medio en torno al cual varía la demanda a lo largo del tiempo.

Trend: es la variación promedio del nivel de la serie temporal entre dos periodos consecutivos.

Relación tendencia-error: la tendencia disminuye cuando el error es positivo y la tendencia aumenta cuando el error es negativo. La intuición es que nuestro modelo aprende de sus errores.

Si el modelo subestima la última demanda, aumentará la tendencia. Si sobrestima la última demanda, disminuirá la tendencia.

DATA SCIENCE FOR SUPPLY CHAIN FORECASTING

HANDS-ON

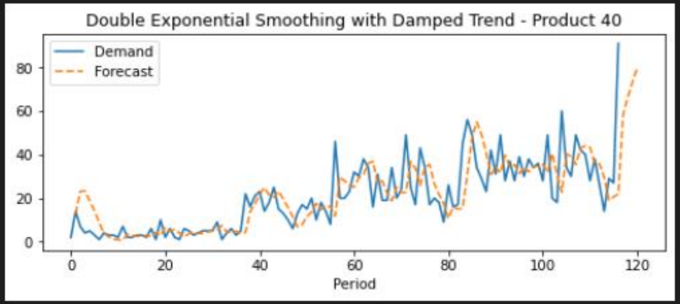
4 DOUBLE EXPONENTIAL SMOOTHING WITH DAMPED TREND

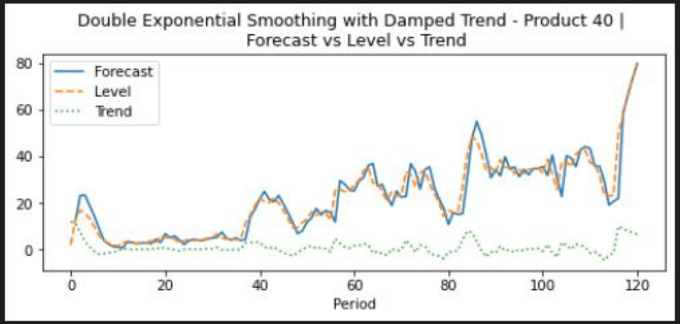
```
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = double_exp_smooth_damped(d, extra_periods=4, alpha=0.4,
                              beta=0.4, phi=0.9)
```

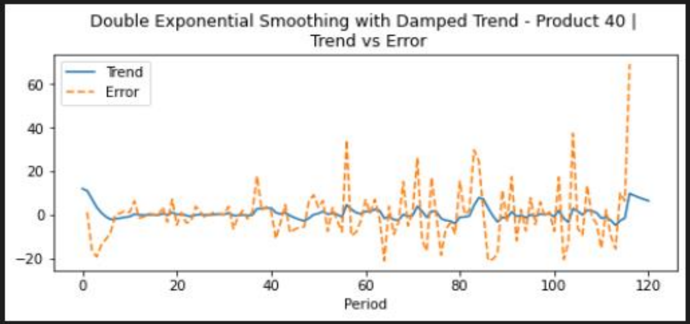
```
Double Exponential Smoothing with Damped Trend Forecast - Product 40
        Demand   Forecast      Level      Trend      Error
Period
0          2.0        NaN   2.000000  12.000000        NaN
1         14.0  12.800000  13.280000  10.992000   1.200000
2          7.0  23.172800  16.703680   7.305152 -16.172800
3          4.0  23.278317  15.566990   3.490106 -19.278317
4          5.0  18.708086  13.224851   0.947802 -13.708086
...        ...        ...        ...        ...        ...
116       91.0  21.910738  49.546443   9.797628  69.089262
117        NaN  58.364308  58.364308   8.817865        NaN
118        NaN  66.300387  66.300387   7.936079        NaN
119        NaN  73.442858  73.442858   7.142471        NaN
120        NaN  79.871082  79.871082   6.428224        NaN

[121 rows x 5 columns]
Double Exponential Smoothing with Damped Trend KPI - Product 40
Bias: 0.13, 0.63%
MAPE: 68.53%
MAE: 8.23, 39.76%
RMSE: 12.51, 60.48%
```

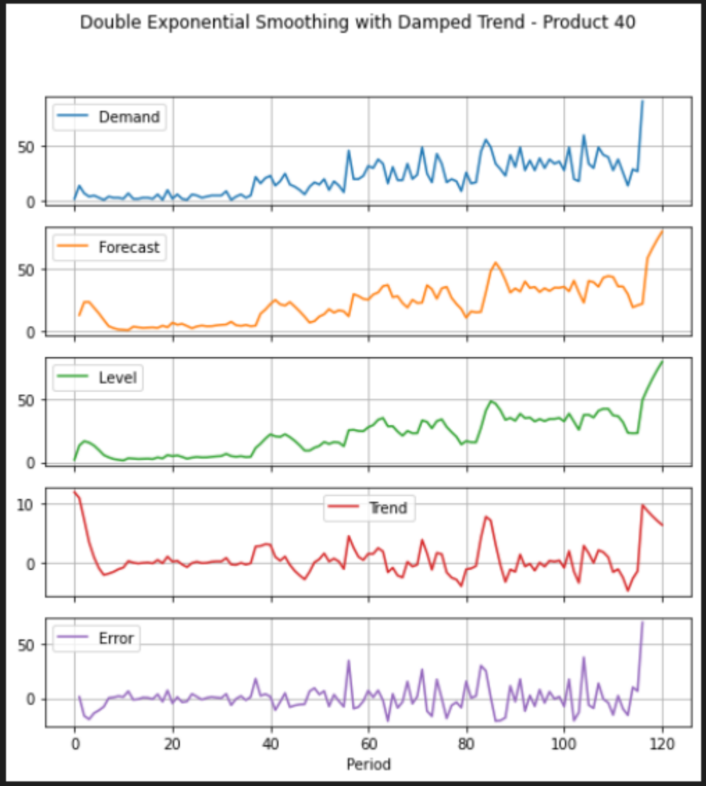
Factor de amortiguamiento (phi):

Reducirá exponencialmente la tendencia a lo largo del tiempo. Este nuevo modelo se olvida de la tendencia con el paso del tiempo. O que el modelo recuerde solo una fracción (phi) de la tendencia estimada anterior.









HANDS-ON

5 TRIPLE EXPONENTIAL SMOOTHING


```python
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = triple_exp_smooth_mul(d, slen=12, extra_periods=4, alpha=0.4,
                           beta=0.4, phi=0.9, gamma=0.2)
```

Triple Exponential Smoothing Forecast - Product 40

Period	Demand	Forecast	Level	Trend	Season	Error
0	2.0	NaN	1.924187	10.336338	1.039400	NaN
1	14.0	12.819718	11.640345	9.468086	1.141876	1.180282
2	7.0	21.743069	14.693320	6.333956	1.078438	-14.743069
3	4.0	22.192582	13.706649	3.025668	1.088198	-18.192582
4	5.0	12.426984	12.502056	1.152023	0.756371	-7.426984
...	...	...	...	...	...	...
116	91.0	36.021133	46.968460	7.473446	1.375607	54.978867
117	NaN	45.102712	53.694561	6.726101	0.839987	NaN
118	NaN	44.534721	59.748052	6.053491	0.745375	NaN
119	NaN	66.744360	65.196194	5.448142	1.023746	NaN
120	NaN	71.466505	70.099522	4.903328	1.019501	NaN

[121 rows x 6 columns]

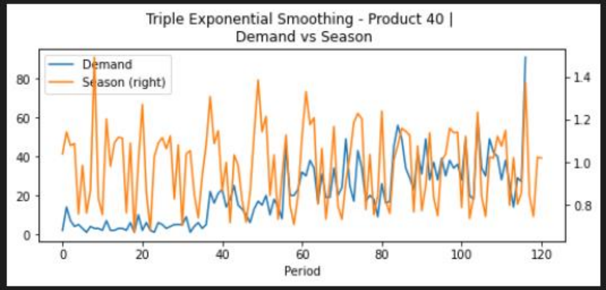
Triple Exponential Smoothing KPI - Product 40

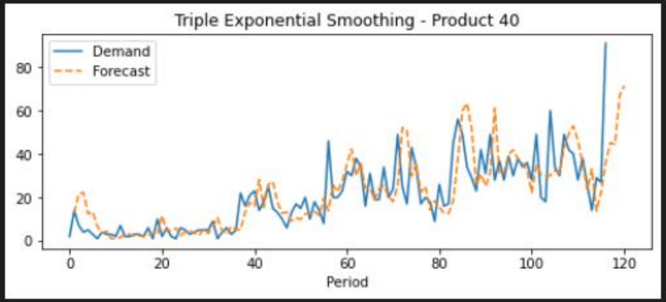
Bias: -0.24, -1.17%

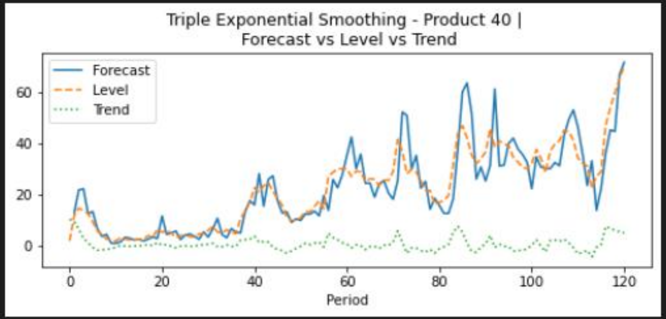
MAPE: 70.45%

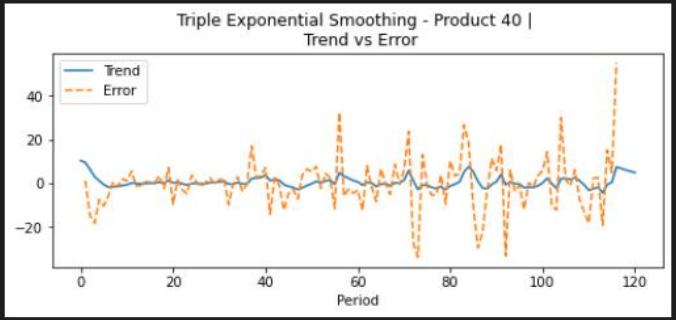
MAE: 7.96, 38.47%

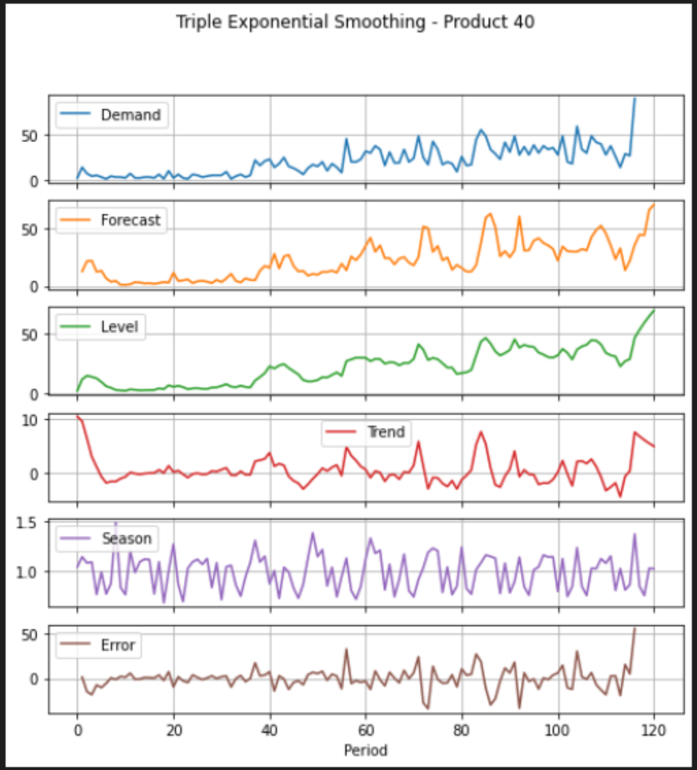
RMSE: 11.99, 57.94%









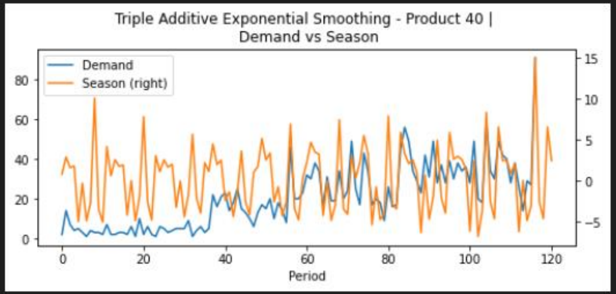


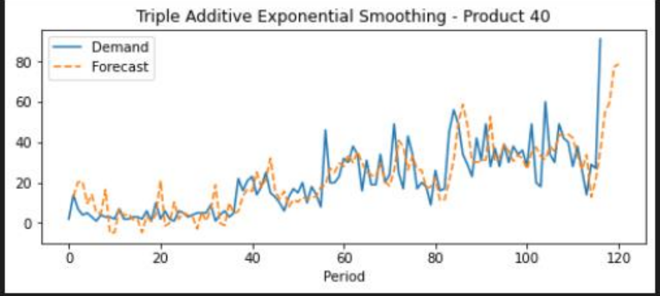
HANDS-ON

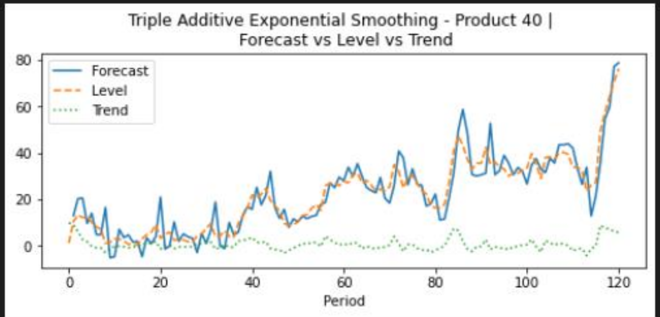
6 TRIPLE ADDITIVE EXPONENTIAL SMOOTHING

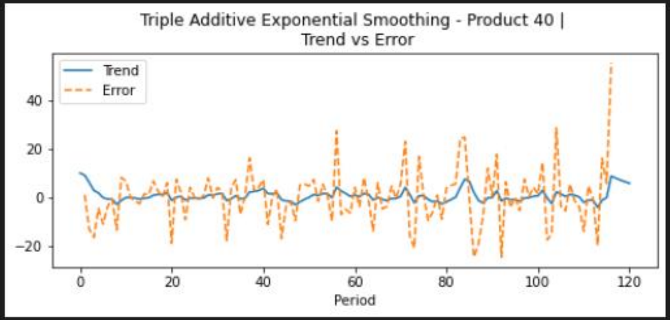

```
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = triple_exp_smooth_add(d, slen=12, extra_periods=4, alpha=0.4,
                           beta=0.4, phi=0.9, gamma=0.2)
```

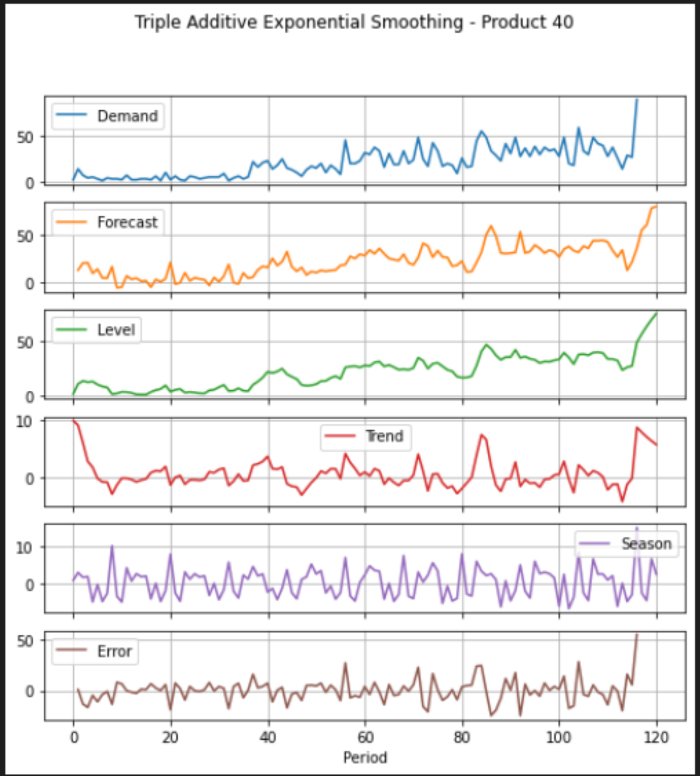
```
Triple Additive Exponential Smooting Forecast - Product 40
        Demand   Forecast      Level     Trend     Season      Error
Period
0          2.0        NaN   1.192593  9.900000   0.807407        NaN
1         14.0  13.010000  10.498593  9.068400   2.907407   0.990000
2          7.0  20.267560  13.353129  6.038750   1.607407 -13.267560
3          4.0  20.595411  12.149839  2.779610   1.807407 -16.595411
4          5.0   9.658895  12.787930  1.756225  -4.992593  -4.658895
...        ...        ...        ...       ...        ...        ...
116       91.0  35.505386  49.340805  8.712197  15.021781  55.494614
117        NaN  54.623637  57.181782  7.840977  -2.558145        NaN
118        NaN  59.611517  64.238661  7.056880  -4.627144        NaN
119        NaN  77.161796  70.589853  6.351192   6.571944        NaN
120        NaN  78.760716  76.305925  5.716072   2.454791        NaN

[121 rows x 6 columns]
Triple Additive Exponential Smooting KPI - Product 40
Bias: 0.18, 0.85%
MAPE: 94.63%
MAE: 8.02, 38.74%
RMSE: 11.50, 55.58%
```











HANDS-ON

DEFINE THE BEST MODEL FOR OUR PRODUCT
OPTIMAL RMSE


```python
def exp_smooth_opti_rmse(d, extra_periods=6, slen=12):
    params = [] # contains all the different parameter sets
    dfs = [] # contains all the DataFrames returned by the different models
    KPIs = [] # contains the results of each model

    for alpha in [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6]:
        df = simple_exp_smooth(d, extra_periods=extra_periods, alpha=alpha)
        params.append(f'Simple Exponential Smoothing, alpha: {alpha}')
        dfs.append(df)
        RMSE = np.sqrt((df['Error']**2).mean())
        KPIs.append(RMSE)

        for beta in [0.05, 0.1, 0.2, 0.3, 0.4]:
            df = double_exp_smooth(d, extra_periods=extra_periods, alpha=alpha, beta=beta)
            params.append(f'Double Exponential Smoothing, alpha: {alpha}, beta: {beta}')
            dfs.append(df)
            RMSE = np.sqrt((df['Error']**2).mean())
            KPIs.append(RMSE)

            for phi in [0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.00]: # To forget the trend over time
                df = double_exp_smooth_damped(d, extra_periods=extra_periods, alpha=alpha, beta=beta, phi=phi)
                params.append(f'Double Exponential Smoothing with Damped Trend, alpha: {alpha}, beta: {beta}, phi: {phi}')
                dfs.append(df)
                RMSE = np.sqrt((df['Error']**2).mean())
                KPIs.append(RMSE)

                for gamma in [0.05, 0.1, 0.2, 0.3]:
                    df = triple_exp_smooth_mul(d, slen=slen, extra_periods=extra_periods, alpha=alpha, beta=beta, phi=phi, gamma=gamma)
                    params.append(f'Triple Exponential Smoothing, alpha: {alpha}, beta: {beta}, phi: {phi}, gamma: {gamma}')
                    dfs.append(df)
                    RMSE = np.sqrt((df['Error']**2).mean())
                    KPIs.append(RMSE)

                    df = triple_exp_smooth_add(d, slen=slen, extra_periods=extra_periods, alpha=alpha, beta=beta, phi=phi, gamma=gamma)
                    params.append(f'Triple Additive Exponential Smoothing, alpga: {alpha}, beta: {beta}, gamma: {gamma}')
                    dfs.append(df)
                    RMSE = np.sqrt((df['Error']**2).mean())
                    KPIs.append(RMSE)

                mini = np.argmin(KPIs)
        # np.argmin() returns the location of the minimum value in an array.
        # Remember that, in Python, the index (location) of the first element in an array is 0.
    print(f'Best solution found: {params[mini]}. RMSE:', round(KPIs[mini], 2))
    return dfs[mini]
```

```python
# Forecast
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = exp_smooth_opti_rmse(d, extra_periods=6, slen=12)
df.index.name = 'Period'
print(df)
```

```
... Best solution found: Triple Exponential Smoothing, alpha: 0.3, beta: 0.05, phi: 0.7, gamma: 0.05. RMSE: 10.01
```

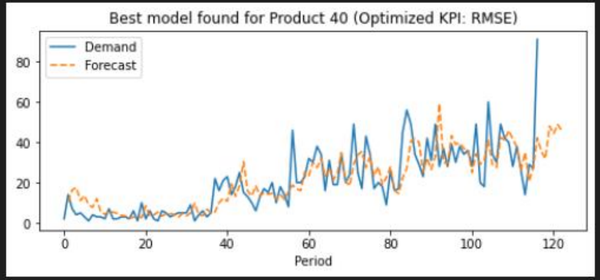
```
        Demand   Forecast      Level      Trend    Season      Error
Period
0          2.0        NaN   1.924187  10.336338  1.039400        NaN
1         14.0  10.459155  10.089894   7.281950  1.141876   3.540845
2          7.0  16.378525  12.578342   4.966919  1.078438  -9.378525
3          4.0  17.471222  12.341370   3.291153  1.088198 -13.471222
4          5.0  11.077185  12.234778   2.183287  0.756371  -6.077185
...        ...        ...        ...        ...       ...        ...
118        NaN  31.881295  41.689297   0.234740  0.764736        NaN
119        NaN  48.024932  41.853615   0.164318  1.147450        NaN
120        NaN  44.270222  41.968638   0.115023  1.054841        NaN
121        NaN  48.782631  42.049153   0.080516  1.160133        NaN
122        NaN  45.687254  42.105514   0.056361  1.085066        NaN

[123 rows x 6 columns]
```

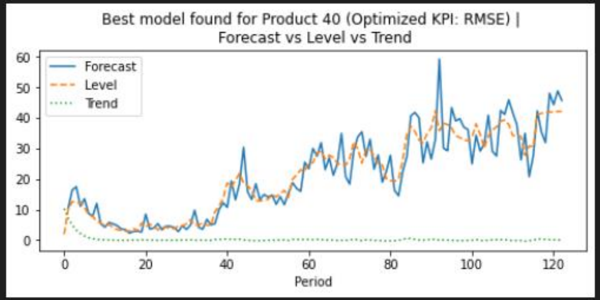
HANDS-ON

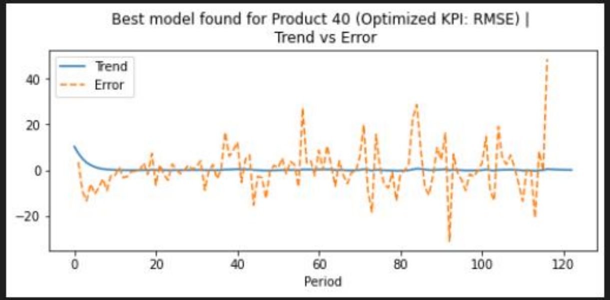
DEFINE THE BEST MODEL FOR OUR PRODUCT OPTIMAL RMSE

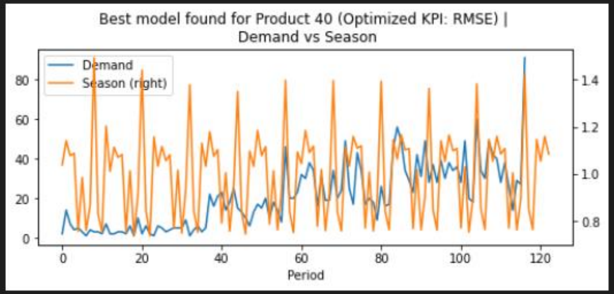

```
...  Best solution found: Triple Exponential Smoothing, alpha: 0.3, beta: 0.05, phi: 0.7, gamma: 0.05. RMSE: 10.01
```

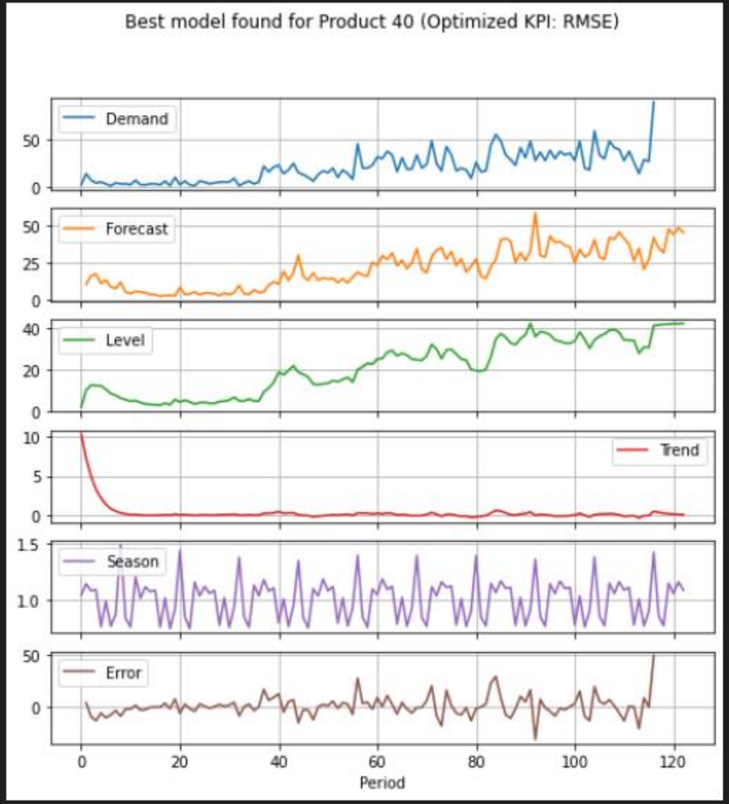
```
Triple Exponential Smoothing Forecast - Product 40
        Demand   Forecast      Level      Trend    Season      Error
Period
0          2.0        NaN   1.924187  10.336338  1.039400        NaN
1         14.0  10.459155  10.089894   7.281950  1.141876   3.540845
2          7.0  16.378525  12.578342   4.966919  1.078438  -9.378525
3          4.0  17.471222  12.341370   3.291153  1.088198 -13.471222
4          5.0  11.077185  12.234778   2.183287  0.756371  -6.077185
...        ...        ...        ...        ...       ...        ...
116       91.0  42.237217  41.119215   0.479061  1.424462  48.762783
117        NaN  35.363314  41.454557   0.335343  0.853062        NaN
118        NaN  31.881295  41.689297   0.234740  0.764736        NaN
119        NaN  48.024932  41.853615   0.164318  1.147450        NaN
120        NaN  44.270222  41.968638   0.115023  1.054841        NaN

[121 rows x 6 columns]
Triple Exponential Smoothing KPI - Product 40
Bias: 0.42, 2.03%
MAPE: 67.47%
MAE: 6.67, 32.23%
RMSE: 10.01, 48.40%
```











HANDS-ON

DEFINE THE BEST MODEL FOR OUR PRODUCT
OPTIMAL MAE


```python
def exp_smooth_opti_mae(d, extra_periods=6, slen=12):
    params = [] # contains all the different parameter sets
    dfs = [] # contains all the DataFrames returned by the different models
    KPIs = [] # contains the results of each model

    for alpha in [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6]:
        df = simple_exp_smooth(d, extra_periods=extra_periods, alpha=alpha)
        params.append(f'Simple Exponential Smoothing, alpha: {alpha}')
        dfs.append(df)
        MAE = df['Error'].abs().mean()
        KPIs.append(MAE)

        for beta in [0.05, 0.1, 0.2, 0.3, 0.4]:
            df = double_exp_smooth(d, extra_periods=extra_periods, alpha=alpha, beta=beta)
            params.append(f'Double Exponential Smoothing, alpha: {alpha}, beta: {beta}')
            dfs.append(df)
            MAE = df['Error'].abs().mean()
            KPIs.append(MAE)

            for phi in [0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.00]: # To forget the trend over time
                df = double_exp_smooth_damped(d, extra_periods=extra_periods, alpha=alpha, beta=beta, phi=phi)
                params.append(f'Double Exponential Smoothing with Damped Trend, alpha: {alpha}, beta: {beta}, phi: {phi}')
                dfs.append(df)
                MAE = df['Error'].abs().mean()
                KPIs.append(MAE)

                for gamma in [0.05, 0.1, 0.2, 0.3]:
                    df = triple_exp_smooth_mul(d, slen=slen, extra_periods=extra_periods, alpha=alpha, beta=beta, phi=phi, gamma=gamma)
                    params.append(f'Triple Exponential Smoothing, alpha: {alpha}, beta: {beta}, phi: {phi}, gamma: {gamma}')
                    dfs.append(df)
                    MAE = df['Error'].abs().mean()
                    KPIs.append(MAE)

                    df = triple_exp_smooth_add(d, slen=slen, extra_periods=extra_periods, alpha=alpha, beta=beta, phi=phi, gamma=gamma)
                    params.append(f'Triple Additive Exponential Smoothing, alpga: {alpha}, beta: {beta}, gamma: {gamma}')
                    dfs.append(df)
                    MAE = df['Error'].abs().mean()
                    KPIs.append(MAE)

                mini = np.argmin(KPIs)
        # np.argmin() returns the location of the minimum value in an array.
        # Remember that, in Python, the index (location) of the first element in an array is 0.
    print(f'Best solution found: {params[mini]}. MAE:', round(KPIs[mini], 2))
    return dfs[mini]
```

```python
# Forecast
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = exp_smooth_opti_mae(d, extra_periods=6, slen=12)
df.index.name = 'Period'
print(df)
```

```
...  Best solution found: Triple Exponential Smoothing, alpha: 0.4, beta: 0.05, phi: 0.7, gamma: 0.05. MAE: 6.59
```

```
        Demand   Forecast      Level      Trend    Season      Error
Period
0          2.0        NaN   1.924187  10.336338  1.039400        NaN
1         14.0  10.459155  10.399985   7.297455  1.141876   3.540845
2          7.0  16.724643  11.901268   4.927872  1.078438  -9.724643
3          4.0  16.704688  10.680788   3.216011  1.088198 -12.704688
4          5.0   9.781384  10.403403   2.124778  0.756371  -4.781384
...        ...        ...        ...        ...       ...        ...
118        NaN  34.669020  45.594792   0.334064  0.760372        NaN
119        NaN  52.226430  45.828637   0.233845  1.139603        NaN
120        NaN  48.388994  45.992329   0.163691  1.052110        NaN
121        NaN  53.054483  46.106912   0.114584  1.150684        NaN
122        NaN  49.982213  46.187121   0.080209  1.082168        NaN

[123 rows x 6 columns]
```

HANDS-ON

DEFINE THE BEST MODEL FOR OUR PRODUCT OPTIMAL MAE

DATA SCIENCE FOR SUPPLY CHAIN FORECASTING

```
Best solution found: Triple Exponential Smoothing, alpha: 0.4, beta: 0.05, phi: 0.7, gamma: 0.05. MAE: 6.59
```

Triple Exponential Smoothing Forecast - Product 40

Period	Demand	Forecast	Level	Trend	Season	Error
0	2.0	NaN	1.924187	10.336338	1.039400	NaN
1	14.0	10.459155	10.399985	7.297455	1.141876	3.540845
2	7.0	16.724643	11.901268	4.927872	1.078438	-9.724643
3	4.0	16.704688	10.680788	3.216011	1.088198	-12.704688
4	5.0	9.781384	10.403403	2.124778	0.756371	-4.781384
...	...	...	...	...	...	...
116	91.0	42.262323	44.783494	0.681763	1.411670	48.737677
117	NaN	38.460452	45.260728	0.477234	0.849753	NaN
118	NaN	34.669020	45.594792	0.334064	0.760372	NaN
119	NaN	52.226430	45.828637	0.233845	1.139603	NaN
120	NaN	48.388994	45.992329	0.163691	1.052110	NaN

[121 rows x 6 columns]

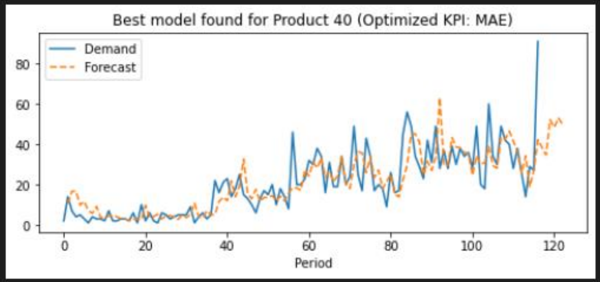
Triple Exponential Smoothing KPI - Product 40

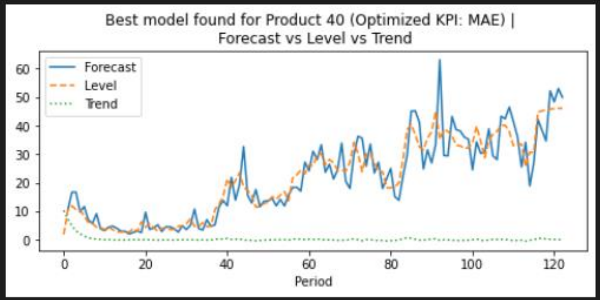
Bias: 0.38, 1.86%

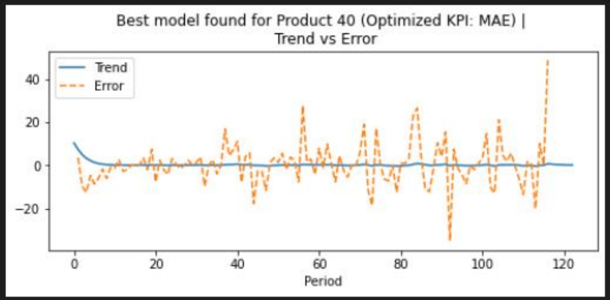
MAPE: 64.03%

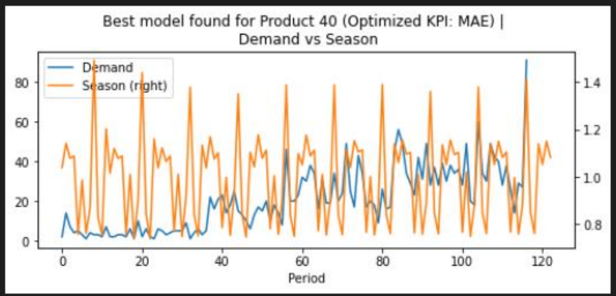
MAE: 6.59, 31.87%

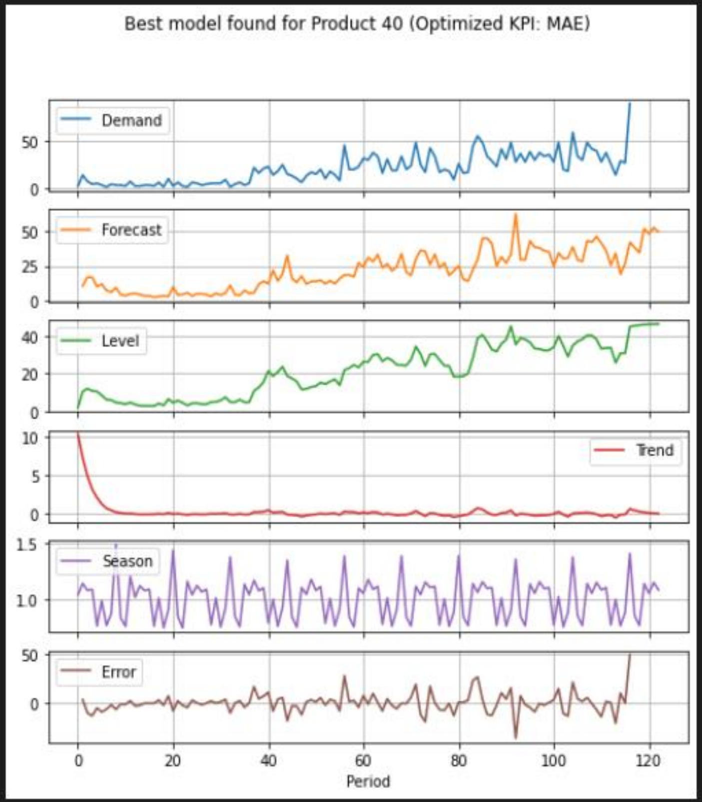
RMSE: 10.10, 48.79%











HANDS-ON

DEAL WITH OUTLIERS (AUTOMATICALLY)

La detección de valores atípicos es un asunto serio. Estos valores atípicos aparecen todo el tiempo en las cadenas de suministro modernas.

En su mayoría se deben a errores o una demanda excepcional. Si puede detectar valores atípicos y suavizarlos, hará un mejor pronóstico.

Marcar valores atípicos manualmente es un proceso que requiere mucho tiempo, es propenso a errores y no es gratificante; pocos planificadores de la demanda se tomarán el tiempo necesario para revisarlos.

Por lo tanto, cuanto mayor sea el conjunto de datos, más importante es automatizar esta detección y limpieza.

Con el método de detección más inteligente, analizando la desviación estándar del error de pronóstico, podremos marcar los valores atípicos con mucha más precisión y reemplazarlos con una cantidad plausible.

Se tomará dos desviaciones estándar como límites para este método.

```python
# Import norm to compute probabilities
from scipy.stats import norm

# Compute the error mean and error standard deviation
m = df['Error'].mean()
s = df['Error'].std()

# Get the percentage of the normal distribution function of each error with norm.cdf
prob = norm.cdf(df['Error'], m, s) # Normal cumulative distribution function

# Define probabilities ranges based on 2 standard deviation
outliers = (prob > 0.99) | (prob < 0.01)

# Re-compute the error mean and standard deviation, but this time, exclude the outliers. '~' useful for exclude.
m2 = df.loc[~outliers, 'Error'].mean()
s2 = df.loc[~outliers, 'Error'].std()

# Update the lower and upper acceptable thresholds based on these new m2 and s2 values
limit_high = norm.ppf(0.99, m2, s2) + df['Forecast'] # Percent point function
limit_low = norm.ppf(0.01, m2, s2) + df['Forecast'] # Percent point function

# Forecast with updated outliers values
df['Updated'] = df['Demand'].clip(lower=limit_low, upper=limit_high)
display(df)

# Replace the updated values in the main dataset
updated_values = df.loc[df['Updated'].notnull(),'Updated']
dataset['Demand'] = dataset['Demand'].replace(dataset['Demand'].tolist(), updated_values)
print('Dataset with Demand updated')
print(dataset)
```

[14] ✓ 0.3s

	Demand	Forecast	Level	Trend	Season	Error	Updated
Period							
113	14.0	34.239264	25.610577	-0.461364	0.997103	-20.239264	15.899767
114	29.0	18.919780	30.676793	-0.053496	0.758041	10.080220	29.000000
115	27.0	26.928939	30.671687	-0.035830	0.878970	0.071061	27.000000
116	91.0	42.262323	44.783494	0.681763	1.411670	48.737677	60.189113
117	NaN	38.460452	45.260728	0.477234	0.849753	NaN	NaN
118	NaN	34.669020	45.594792	0.334064	0.760372	NaN	NaN
119	NaN	52.226430	45.828637	0.233845	1.139603	NaN	NaN
120	NaN	48.388994	45.992329	0.163691	1.052110	NaN	NaN
121	NaN	53.054483	46.106912	0.114584	1.150684	NaN	NaN
122	NaN	49.982213	46.187121	0.080209	1.082168	NaN	NaN

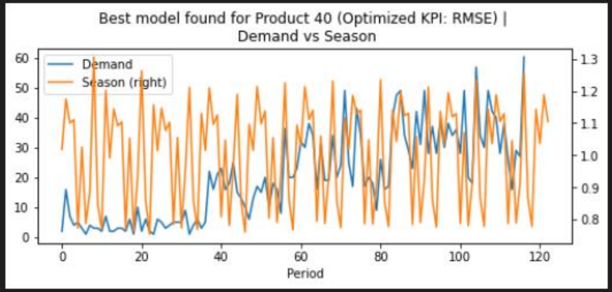
```
Dataset with Demand updated
      Year  Month     Product     Demand
74    2007      2  Product 40   2.000000
105   2007      3  Product 40  15.899767
145   2007      4  Product 40   7.000000
184   2007      5  Product 40   4.000000
222   2007      6  Product 40   5.000000
...    ...    ...         ...        ...
4229  2016      9  Product 40  26.000000
4263  2016     10  Product 40  15.899767
4299  2016     11  Product 40  29.000000
4335  2016     12  Product 40  27.000000
4363  2017      1  Product 40  60.189113
```

HANDS-ON

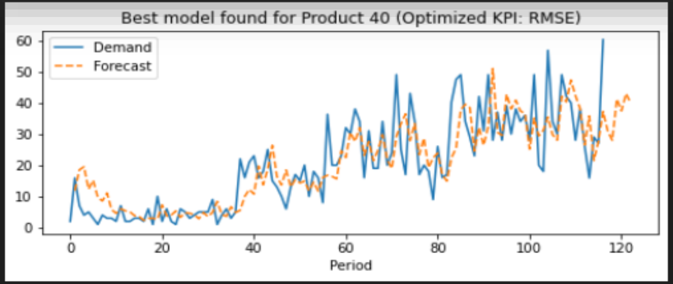
DEFINE THE BEST MODEL FOR OUR PRODUCT
OPTIMAL RMSE (WITHOUT OUTLIERS)

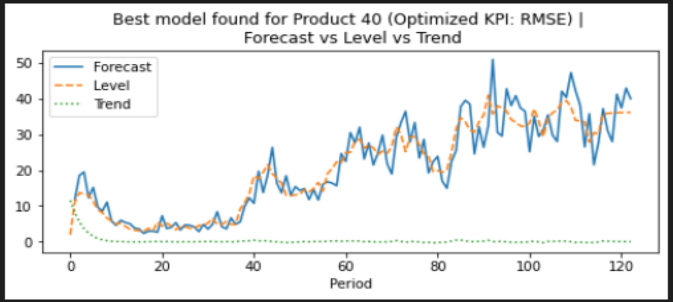

```
# Forecast
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = exp_smooth_opti_rmse(d, extra_periods=6, slen=12)
df.index.name = 'Period'
print(df)
```

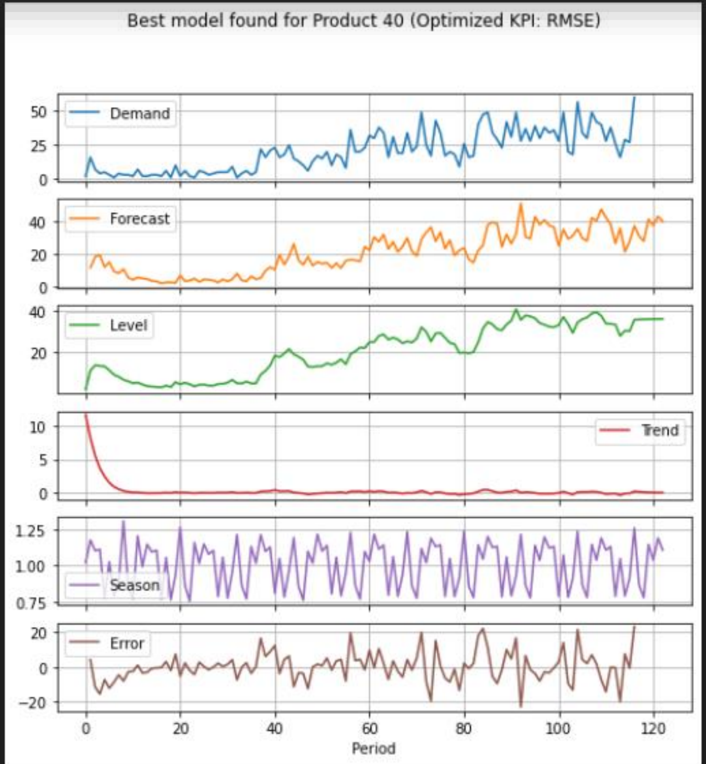
```
Triple Exponential Smoothing Forecast - Product 40
           Demand   Forecast      Level      Trend    Season      Error
Period
0        2.000000        NaN   1.963651  11.568077  1.018511        NaN
1       15.899767  11.822023  11.102432   8.149710  1.174999   4.077744
2        7.000000  18.501136  13.672791   5.548075  1.100784 -11.501136
3        4.000000  19.500754  13.369865   3.674324  1.110746 -15.500754
4        5.000000  12.307832  13.102220   2.430043  0.772043  -7.307832
...           ...        ...        ...        ...       ...        ...
116     60.189113  37.178948  35.699665   0.222749  1.257361  23.010165
117           NaN  31.061854  35.855589   0.155924  0.866304        NaN
118           NaN  27.967106  35.964736   0.109147  0.777626        NaN
119           NaN  41.218501  36.041139   0.076403  1.143651        NaN
120           NaN  37.405204  36.094621   0.053482  1.036310        NaN

[121 rows x 6 columns]
Triple Exponential Smoothing KPI - Product 40
Bias: 0.17, 0.84%
MAPE: 67.38%
MAE: 6.28, 31.02%
RMSE: 8.58, 42.36%
```




```
Best solution found: Triple Exponential Smoothing, alpha: 0.3, beta: 0.05, phi: 0.7, gamma: 0.05. RMSE: 8.58
```





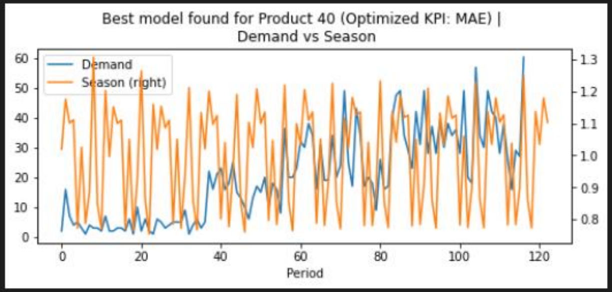


HANDS-ON

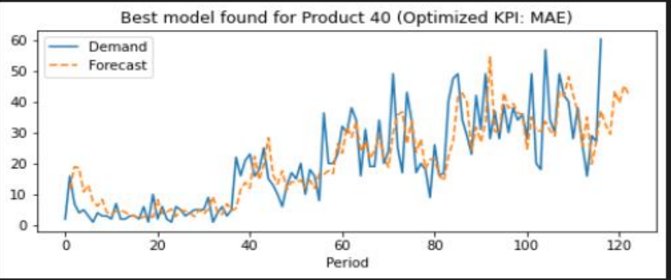
DEFINE THE BEST MODEL FOR OUR PRODUCT OPTIMAL MAE (WITHOUT OUTLIERS)

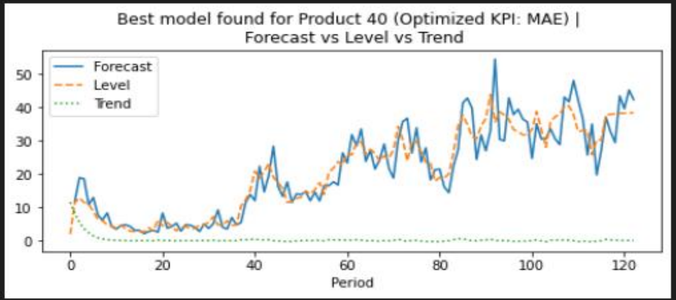

```python
# Forecast
d = dataset['Demand'].tolist() # Convert dataframe column to a list
df = exp_smooth_opti_mae(d, extra_periods=6, slen=12)
df.index.name = 'Period'
print(df)
```

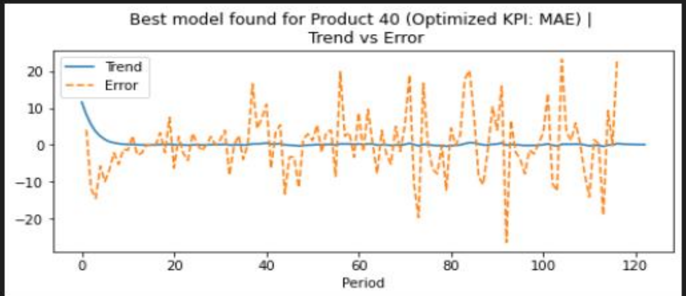
```
...  Triple Exponential Smoothing Forecast - Product 40
          Demand   Forecast      Level      Trend    Season      Error
Period
0       2.000000        NaN   1.963651  11.568077  1.018511        NaN
1      15.899767  11.822023  11.449474   8.167062  1.174999   4.077744
2       7.000000  18.896525  12.843491   5.500797  1.100784 -11.896525
3       4.000000  18.542853  11.456902   3.588701  1.110746 -14.542853
4       5.000000  10.784668  10.971924   2.362237  0.772043  -5.784668
...          ...        ...        ...        ...       ...        ...
116    60.189113  37.055260  37.663412   0.342569  1.247970  23.133853
117          NaN  32.675162  37.903211   0.239799  0.862068        NaN
118          NaN  29.434172  38.071070   0.167859  0.773138        NaN
119          NaN  43.411720  38.188571   0.117501  1.136773        NaN
120          NaN  39.573321  38.270822   0.082251  1.034034        NaN

[121 rows x 6 columns]
Triple Exponential Smoothing KPI - Product 40
Bias: 0.16, 0.78%
MAPE: 63.06%
MAE: 6.21, 30.63%
RMSE: 8.60, 42.42%
```

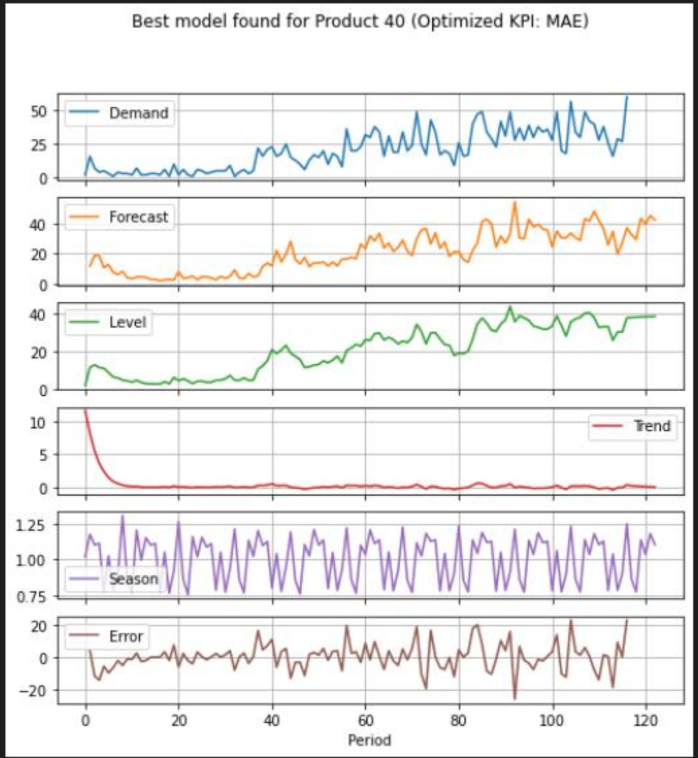



```
... Best solution found: Triple Exponential Smoothing, alpha: 0.4, beta: 0.05, phi: 0.7, gamma: 0.05. MAE: 6.21
```









WRAP-UP

THE ANALYSIS

```
...   Best solution found: Triple Exponential Smoothing, alpha: 0.3, beta: 0.05, phi: 0.7, gamma: 0.05. RMSE: 8.58
...   Best solution found: Triple Exponential Smoothing, alpha: 0.4, beta: 0.05, phi: 0.7, gamma: 0.05. MAE: 6.21
```

De los 65 únicos productos del conjunto de datos, se escogió aleatoriamente el producto # 40.

Detectando y reemplazando con cantidades plausibles a las demandas atípicas del producto, podemos mejorar nuestro pronóstico.

Si nuestro objetivo es ajustar nuestro pronóstico a la demanda promedio, debemos enfocarnos en reducir lo máximo posible el KPI: RMSE (%)

Para el producto #40, con demandas atípicas ajustadas, el algoritmo de optimización sugiere que la técnica de pronóstico estadístico 'Triple Exponential Smoothing' es la más adecuada con los parámetros:

- alpha: 0.3
- beta: 0.05
- phi: 0.7
- gamma: 0.05

Dando un RMSE de 8.58 unidades (42.36%)

El gráfico 'Demand vs Season' indica que el producto #40 es estacional.

Si nuestro objetivo es ajusta nuestro pronóstico a la demanda mediana, debemos enfocarnos en reducir lo máximo posible el KPI: MAE (%)

Para el producto #40, con demandas atípicas ajustadas, el algoritmo de optimización sugiere que la técnica de pronóstico estadístico 'Triple Exponential Smoothing' es la más adecuada con los parámetros:

- alpha: 0.4
- beta: 0.05
- phi: 0.7
- gamma: 0.05

Dando un MAE de 6.21 unidades (30.63%)

El gráfico 'Demand vs Season' indica que el producto #40 es estacional.

Para el producto #40, el pronóstico de los próximos 04 meses son 31.06, 27.97, 47.21 y 37.41, respectivamente.

Sin embargo, ¿cómo podemos pronosticar la demanda del siguiente periodo de los más de 60 productos en nuestro conjunto de datos?

¿Cómo podemos relacionar la demanda histórica de nuestros productos con sus respectivos cambios de precios, clima, crecimiento del GDP o tas de desempleo?

¡Esto será posible aplicando modelos de Machine Learning!

MACHINE LEARNING

IMPORT , TRANSFORM DATA AND BUILT MAIN FUNCTIONS

dataset.csv ×

dataset.csv

```
Year,Month,Product,Demand
2007,1,Product 1,2884
2007,1,Product 2,2521
2007,1,Product 3,1029
2007,1,Product 4,870
2007,1,Product 5,693
2007,1,Product 6,665
2007,1,Product 7,622
2007,1,Product 8,599
2007,1,Product 9,423
2007,1,Product 10,362
2007,1,Product 11,352
2007,1,Product 12,263
2007,1,Product 13,258
2007,1,Product 14,191
2007,1,Product 15,169
2007,1,Product 16,168
2007,1,Product 17,136
2007,1,Product 18,127
2007,1,Product 19,97
2007,1,Product 20,55
2007,1,Product 21,33
2007,1,Product 22,26
2007,1,Product 23,26
2007,1,Product 24,22
2007,1,Product 25,20
2007,1,Product 26,16
2007,1,Product 27,15
2007,1,Product 28,14
2007,1,Product 29,9
2007,1,Product 30,4
2007,1,Product 31,4
2007,1,Product 32,3
2007,1,Product 33,2
2007,1,Product 34,2
2007,1,Product 35,2
2007,1,Product 36,1
2007,1,Product 37,1
2007,1,Product 38,1
2007,2,Product 1,1885
2007,2,Product 2,1517
2007,2,Product 4,686
2007,2,Product 3,621
2007,2,Product 5,570
2007,2,Product 7,551
2007,2,Product 8,498
```

```python
# Import dataset
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def import_data():
    data = pd.read_csv('dataset.csv')
    # z.fill(x) adds leading zeros to a string until it reaches a length of x.
    data['Period'] = data['Year'].astype(str) + '-' + data['Month'].astype(str).str.zfill(2)
    df = pd.pivot_table(data=data, values='Demand', index='Product', columns='Period',
                        aggfunc='sum', fill_value=0)
    return df
```

```python
# Build an appropriate dataframe for our machine learning
def datasets(df, x_len=12, y_len=1, test_loops=12):
    D = df.values
    rows, periods = D.shape

    # Training set creation
    loops = periods + 1 - x_len - y_len
    train = []
    for col in range(loops):
        train.append(D[:, col:col+x_len+y_len])
    train = np.vstack(train)
    X_train, Y_train = np.split(train, [-y_len], axis=1)

    # Test set creation
    # np.split(array, indices, axis): cuts an array at specific indices along an axis
    if test_loops > 0:
        X_train, X_test = np.split(X_train, [-rows*test_loops], axis=0)
        Y_train, Y_test = np.split(Y_train, [-rows*test_loops], axis=0)

    # No test set
    # X_test is used to generate the future forecast
    else:
        X_test = D[:, -x_len:]
        Y_test = np.full((X_test.shape[0], y_len), np.nan)

    # Formatting required for scikit-learn
    # array.ravel(): reduces the dimension of a Numpy array to 1D.
    if y_len == 1:
        Y_train = Y_train.ravel()
        Y_test = Y_test.ravel()

    return X_train, Y_train, X_test, Y_test
```

```python
# We can now easily call our new functions
df = import_data()
X_train, Y_train, X_test, Y_test = datasets(df, x_len=12, y_len=1, test_loops=12)
print(df)

# Note that we set test_loops as 12 periods.
# That means that we will test our algorithm over 12 different loops.
# We will predict 12 times the following period [y_len=1] based on the last 12 periods [x_len=12]).

# We obtain the datasets we need to traing our machine learning algorithm (X_train and Y_train)
# and the datasets we need to test it (X_test and Y_test)
```

Period	2007-01	2007-02	2007-03	2007-04	2007-05	2007-06	2007-07 \
Product							
Product 1	2884	1885	1833	1300	1866	1620	1901
Product 10	362	410	387	387	422	421	469
Product 11	352	335	365	360	431	477	403
Product 12	263	247	239	179	223	277	281
Product 13	258	264	333	347	420	262	296
...	...	...	...	...	...	...	...
Product 65	0	0	0	0	0	0	0
Product 66	0	0	0	0	0	0	0
Product 7	622	551	578	534	771	683	685
Product 8	599	498	682	556	630	498	562
Product 9	423	356	399	351	520	624	401

```python
#KPI Function
def kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name=''):
    df = pd.DataFrame(columns = ['MAE', 'RMSE', 'Bias'], index=['Train', 'Test'])
    df.index.name = name
    df.loc['Train','MAE'] =   100*np.mean(abs(Y_train - Y_train_pred))/np.mean(Y_train)
    df.loc['Train', 'RMSE'] = 100*np.sqrt(np.mean((Y_train - Y_train_pred)**2))/np.mean(Y_train)
    df.loc['Train', 'Bias'] = 100*np.mean((Y_train - Y_train_pred))/np.mean(Y_train)
    df.loc['Test', 'MAE'] =   100*np.mean(abs(Y_test - Y_test_pred))/np.mean(Y_test)
    df.loc['Test', 'RMSE'] =  100*np.sqrt(np.mean((Y_test - Y_test_pred)**2))/np.mean(Y_test)
    df.loc['Test', 'Bias'] =  100*np.mean((Y_test - Y_test_pred))/np.mean(Y_test)
    df = df.astype(float).round(1) # Round number for display
    print(df)
```

MACHINE LEARNING

1 LINER REGRESSION


```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()

# Fit the linear regression to the training data
reg = reg.fit(X_train, Y_train)

# Create predictions for the training and test sets
Y_train_pred = reg.predict(X_train)
Y_test_pred = reg.predict(X_test)

# Measure its accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='Linear Regression')
```

✓ 1m 4.9s

	MAE	RMSE	Bias
Linear Regression			
Train	17.9	44.2	-0.0
Test	17.8	44.0	1.6

```
# Linear Regression Forecast for the next month
df = import_data()
X_train, Y_train, X_test, Y_test = datasets(df, x_len=12, y_len=1, test_loops=0)

reg = LinearRegression()
reg = reg.fit(X_train, Y_train)
forecast = pd.DataFrame(data=reg.predict(X_test), index=df.index)
print('Linear Regression Forecast for the next period')
display(forecast)
```

[6] ✓ 5.9s

Linear Regression Forecast for the next period

	0
Product	
Product 1	1457.102925
Product 10	794.012107
Product 11	1265.040299
Product 12	158.945172
Product 13	292.543366
...	...
Product 65	1.114935
Product 66	4.216304
Product 7	259.025417
Product 8	646.563604
Product 9	123.639567

66 rows × 1 columns

MACHINE LEARNING

2 TREE

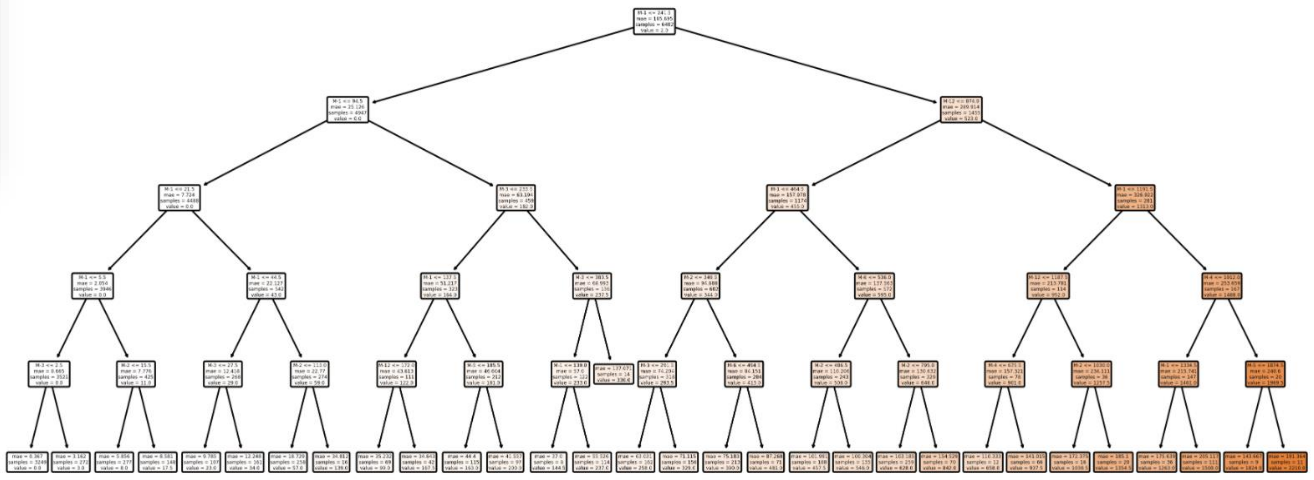

```python
from sklearn.tree import DecisionTreeRegressor
tree = DecisionTreeRegressor(max_depth=5, min_samples_split=15, min_samples_leaf=5)

# Fit the tree to the training data
tree = tree.fit(X_train, Y_train)

# Create predictions for the training and test sets
Y_train_pred = tree.predict(X_train)
Y_test_pred = tree.predict(X_test)

# Measure its accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='Tree')
```

✓ 23.6s

	MAE	RMSE	Bias
Tree			
Train	18.1	43.7	-0.0
Test	21.1	53.0	3.2

```python
# Plotting our tree
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(15,6), dpi=300)
ax = fig.gca()

plot_tree(tree, fontsize=3, feature_names=[f'M{x-12}' for x in range(12)], rounded=True, filled=True, ax=ax)
fig.savefig('Regression Tree.PNG')
```

✓ 1m 8.4s



MACHINE LEARNING

2 OPTIMAL TREE


```python
# k-fold Cross-validation to avoid overfitting (cv parameter)
# Random Search to efficiently find a very good parameter set among different possibilites

max_depth = list(range(5,11)) + [None]
min_samples_split = range(5,20)
min_samples_leaf = range(2,20)

param_dist = {'max_depth': max_depth,
              'min_samples_split': min_samples_split,
              'min_samples_leaf': min_samples_leaf}

# Test all these different parameter sets against our training set
from sklearn.model_selection import RandomizedSearchCV
tree = DecisionTreeRegressor()
tree_cv = RandomizedSearchCV(tree, param_dist, n_jobs=-1, cv=10, verbose=1, n_iter=100, scoring='neg_mean_absolute_error')

# Fit the tree to the training data
tree_cv.fit(X_train, Y_train)
print('Tuned Regression Tree Parameters:', tree_cv.best_params_)


# Create predictions for the training and test sets
Y_train_pred = tree_cv.predict(X_train)
Y_test_pred = tree_cv.predict(X_test)

# Measure its accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='Tree Optimization')
```

[6] ✓ 1m 18.1s

```
Fitting 10 folds for each of 100 candidates, totalling 1000 fits
Tuned Regression Tree Parameters: {'min_samples_split': 8, 'min_samples_leaf': 14, 'max_depth': 6}
                     MAE   RMSE   Bias
Tree Optimization
Train               17.1   42.0    0.0
Test                19.3   48.1    2.9
```

```python
# DecisionTreeRegressor Optimized Forecast for the next month
df = import_data()
X_train, Y_train, X_test, Y_test = datasets(df, x_len=12, y_len=1, test_loops=0)

# Set with the optimized parameters found
# Tuned Regression Tree Parameters: {'min_samples_split': 17, 'min_samples_leaf': 18, 'max_depth': 7}
tree = DecisionTreeRegressor(max_depth=7, min_samples_split=17, min_samples_leaf=18)
tree = tree.fit(X_train, Y_train)
forecast = pd.DataFrame(data=tree.predict(X_test), index=df.index)
print('DecisionTreeRegressor Optimized Forecast for the next period')
display(forecast)
```

[15] ✓ 1.2s

```
DecisionTreeRegressor Optimized Forecast for the next period
```

	0
Product	
Product 1	1602.172414
Product 10	937.971429
Product 11	1216.391304
Product 12	193.126866
Product 13	265.626866
...	...
Product 65	0.162784
Product 66	10.811024
Product 7	339.521739
Product 8	591.570000
Product 9	103.406250

66 rows × 1 columns

MACHINE LEARNING

3 FOREST VS OPTIMAL FOREST


```
from sklearn.ensemble import RandomForestRegressor
forest = RandomForestRegressor(bootstrap=True, max_samples=0.95, max_features=11, min_samples_leaf=18, max_depth=7)

# Fit he forest to the training data
forest.fit(X_train, Y_train)

# Create predictions for the training and test sets
Y_train_pred = forest.predict(X_train)
Y_test_pred = forest.predict(X_test)

# Measure its accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='Forest')
```

[7] ✓ 8.6s

```
         MAE  RMSE  Bias
Forest
Train   15.7  40.3   0.0
Test    18.3  47.3   3.5
```

```
# k-fold Cross-validation to avoid overfitting (cv parameter)
# Random Search to efficiently find a very good parameter set among different possibilites

max_depth = list(range(5,11)) + [None]
min_samples_split = range (5,20)
min_samples_leaf = range (2,15)

# Bootstrap means that each tree will receive a random selection from the initial training dataset
# max_samples means to limit the amount of data ggiven to each tree
# max_features means to limit the maximum number of features that the algorithm can choose from each node and
# the features are choosen randomly each time, we will obtain different trees at each fitting.
bootstrap = [True] #We force bootstrap
max_samples = [.7, .8, .9, .95, 1]
max_features = range(3,8)

param_dist = {'max_depth': max_depth,
              'min_samples_split': min_samples_split,
              'min_samples_leaf': min_samples_leaf,
              'bootstrap': bootstrap,
              'max_samples': max_samples,
              'max_features': max_features}

# Test all these different parameter sets against our training set
from sklearn.model_selection import RandomizedSearchCV
forest = RandomForestRegressor(n_jobs=1, n_estimators=30)
forest_cv = RandomizedSearchCV(forest, param_dist, cv=6, n_jobs = -1, verbose=2, n_iter=400, scoring='neg_mean_absolute_error')

# Fit the forest to the training data
forest_cv.fit(X_train, Y_train)
print('Tuned Forest Parameters:', forest_cv.best_params_)

# Create predictions for the training and test sets
Y_train_pred = forest_cv.predict(X_train)
Y_test_pred = forest_cv.predict(X_test)

# Measure the accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='Forest Optimization')
```

[9] ✓ 23m 42.5s

```
Fitting 6 folds for each of 400 candidates, totalling 2400 fits
Tuned Forest Parameters: {'min_samples_split': 7, 'min_samples_leaf': 4, 'max_samples': 0.9, 'max_features': 6, 'max_depth': 10, 'bootstrap': True}

                      MAE  RMSE  Bias
Forest Optimization
Train                12.1  30.8   0.1
Test                 17.7  45.8   2.8
```

MACHINE LEARNING

3 OPTIMAL FORESTX200


```
# In order to get the best out of our forest, let's run a forest with
# our new optimal parmeters and n_estimators=200.
# We can easily allow ourselves 200 trees due to the dataset limited size
forest = RandomForestRegressor(n_jobs=-1, n_estimators=200, **forest_cv.best_params_)

# Fit the forest to the training data
forest = forest.fit(X_train, Y_train)

# Create predictions for the training and test sets
Y_train_pred = forest.predict(X_train)
Y_test_pred = forest.predict(X_test)

# Measure the accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='Forestx200')
```

[10] ✓ 12.9s

	MAE	RMSE	Bias
Forestx200			
Train	11.9	30.4	-0.0
Test	17.5	45.4	2.3

```
# Forest Optimized x 200 Forecast for the next month
df = import_data()
X_train, Y_train, X_test, Y_test = datasets(df, x_len=12, y_len=1, test_loops=0)

forest200 = RandomForestRegressor(n_jobs=-1, n_estimators=200, **forest_cv.best_params_)
forest200 = forest200.fit(X_train, Y_train)
forecast  = pd.DataFrame(data=forest200.predict(X_test),index=df.index)
print('Forest Optimized x 200 Forecast for the next period')
display(forecast)
```

[6] ✓ 11.6s

Forest Optimized x 200 Forecast for the next period

	0
Product	
Product 1	1525.167175
Product 10	804.875529
Product 11	1078.683406
Product 12	169.169076
Product 13	305.843355
...	...
Product 65	0.086483
Product 66	9.101335
Product 7	295.069647
Product 8	692.907534
Product 9	121.680588

66 rows × 1 columns

MACHINE LEARNING

4 EXTREMELY RANDOMIZED TREES VS OPTIMAL EXTREMELY RANDOMIZED TREES


```python
from sklearn.ensemble import ExtraTreesRegressor
ETR = ExtraTreesRegressor(n_jobs=-1, n_estimators=200, min_samples_split=15, min_samples_leaf=4, max_samples=0.95,
                          max_features=4, max_depth=8, bootstrap=True)

# Fit the ExtraTreesRegressor to the training data
ETR.fit(X_train,Y_train)

# Create predictions for the training and test sets
Y_train_pred = ETR.predict(X_train)
Y_test_pred = ETR.predict(X_test)

# Measure the accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='ETR')
```

[8] ✓ 7.1s

ETR	MAE	RMSE	Bias
Train	17.8	43.9	0.1
Test	18.8	47.5	3.3

```python
# k-fold Cross-validation to avoid overfitting (cv parameter)
# Random Search to efficiently find a very good parameter set among different possibilites

max_depth = list(range(6, 13)) + [None]
min_samples_split = range(7,16)
min_samples_leaf = range(2,13)

# Bootstrap means that each tree will receive a random selection from the initial training dataset
# max_samples means to limit the amount of data ggiven to each tree
# max_features means to limit the maximum number of features that the algorithm can choose from each node and
# the features are choosen randomly each time, we will obtain different trees at each fitting.
bootstrap = [True] # We force bootstrap
max_samples = [.7, .8, .9, .95, 1]
max_features = range(5,13)

param_dist = {'max_depth': max_depth,
              'min_samples_split': min_samples_split,
              'min_samples_leaf': min_samples_leaf,
              'bootstrap': bootstrap,
              'max_samples': max_samples,
              'max_features': max_features}

# Test all these different parameter sets against our training set
ETR = ExtraTreesRegressor(n_jobs=1, n_estimators=30)
ETR_cv = RandomizedSearchCV(ETR, param_dist, cv=5, verbose=2, n_jobs=-1, n_iter=400, scoring='neg_mean_absolute_error')

# Fit the forest to the training data
ETR_cv.fit(X_train, Y_train)
print('Tuned Forest Parameters', ETR_cv.best_params_)

# Create predictions for the training and test sets
Y_train_pred = ETR_cv.predict(X_train)
Y_test_pred = ETR_cv.predict(X_test)

# Measure the accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='ETR Optimization')
```

[10] ✓ 8m 50.9s

```
Fitting 5 folds for each of 400 candidates, totalling 2000 fits
Tuned Forest Parameters {'min_samples_split': 11, 'min_samples_leaf': 2, 'max_samples': 0.9, 'max_features': 10, 'max_depth': 11, 'bootstrap': True}
```

ETR optimized	MAE	RMSE	Bias
Train	14.4	36.2	0.1
Test	17.8	45.8	2.6

MACHINE LEARNING

4 OPTIMAL EXTREMELY RANDOMIZED TREESX200


```python
# In order to get the best out of our ETR, let's run a ETR with
# our new optimal parameters and n_estimators=200.
# We can easily allow ourselves 200 trees due to the dataset limited size
ETR = ExtraTreesRegressor(n_jobs=-1, n_estimators=200, **ETR_cv.best_params_)

# Fit the ETR to the training data
ETR = ETR.fit(X_train, Y_train)

# Create predictions for the training and test sets
Y_train_pred = ETR.predict(X_train)
Y_test_pred = ETR.predict(X_test)

# Measure the accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='ETRx200')
```

[11] ✓ 4.7s

ETRx200	MAE	RMSE	Bias
Train	14.1	35.6	-0.0
Test	17.6	44.9	2.4

```python
# ETR Optimized x 200 Forecast for the next month
df = import_data()
X_train, Y_train, X_test, Y_test = datasets(df, x_len=12, y_len=1, test_loops=0)

ETR = ExtraTreesRegressor(n_jobs=-1, n_estimators=200, **ETR_cv.best_params_)
ETR = ETR.fit(X_train, Y_train)
forecast  = pd.DataFrame(data=ETR.predict(X_test),index=df.index)
print('ETR Optimized x 200 Forecast for the next period')
display(forecast)
```

[12] ✓ 4.9s

ETR Optimized x 200 Forecast for the next period

Product	0
Product 1	1508.069742
Product 10	809.593151
Product 11	1085.162544
Product 12	170.495017
Product 13	288.021175
...	...
Product 65	0.233671
Product 66	8.463769
Product 7	284.940312
Product 8	663.120976
Product 9	141.181486

66 rows × 1 columns

MACHINE LEARNING

5 ADAPTATIVE BOOSTING VS OPTIMAL ADAPTATIVE BOOSTING


```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor
ada = AdaBoostRegressor(DecisionTreeRegressor(max_depth=8), n_estimators=100, learning_rate=0.25, loss='square')

# Fit the AdaBoost to the training data
ada = ada.fit(X_train, Y_train)

# Create predictions or the training and test sets
Y_train_pred = ada.predict(X_train)
Y_test_pred = ada.predict(X_test)

# Measure the accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='AdaBoost')
```

[15] ✓ 14.8s

	MAE	RMSE	Bias
AdaBoost			
Train	9.9	21.0	-0.4
Test	18.0	47.7	2.7

```
# n_estimators = [100]
learning_rate = [0.005,0.01,0.05,0.1,0.15,0.2,0.25,0.3,0.35]
loss = ['square', 'exponential', 'linear']
param_dist = {#'n_estimators': n_estimators,
              'learning_rate': learning_rate,
              'loss': loss}

def model_mae(model, X, Y):
    Y_pred = model.predict(X)
    mae = np.mean(np.abs(Y - Y_pred))/np.mean(Y)
    return mae
# Let's now go into our optimization loop.
# for each max_depth tried by using ada_cv.best_score and ada_cv.best_params_

from sklearn.model_selection import RandomizedSearchCV
results = [] # To record the best score and parameters obtained

for max_depth in range(2,18,2):
    ada = AdaBoostRegressor(DecisionTreeRegressor(max_depth=max_depth))
    ada_cv = RandomizedSearchCV(ada, param_dist, n_jobs=-1, cv=6, n_iter=20, scoring='neg_mean_absolute_error')
    ada_cv.fit(X_train, Y_train)
    print('Tuned AdaBoost Parameters:', ada_cv.best_params_)
    print('Result:', ada_cv.best_score_)
    results.append([ada_cv.best_score_, ada_cv.best_params_, max_depth])

# We can then transform results into a DataFrame
# The method idxmax() on our DataFrame to print the parameter set that got the lowest error
# RandomizedSearchCV is returning the negative mean absolute error.

results = pd.DataFrame(data=results, columns=['Score', 'Best Params', 'Max_Depth'])
optimal = results['Score'].idxmax()
print(results.iloc[optimal])
```

[17] ✓ 51m 4.4s

```
Best Params    {'loss': 'exponential', 'learning_rate': 0.01}
Max_Depth                                                   12
```

MACHINE LEARNING

5 OPTIMAL ADAPTATIVE BOOSTING


```
# Test the optimal parameter found against our training set
ada = AdaBoostRegressor(DecisionTreeRegressor(max_depth=12),
                        n_estimators=100,
                        learning_rate=0.01,
                        loss='exponential')

# Fit the AdaBoost to the traiing data
ada.fit(X_train, Y_train)

# Create predictions for the training and test sets
Y_train_pred = ada.predict(X_train)
Y_test_pred = ada.predict(X_test)

# Measure the accuracy
kpi_ML(Y_train, Y_train_pred, Y_test, Y_test_pred, name='AdaBoost optimized')
```

[18] ✓ 17.5s

	MAE	RMSE	Bias
AdaBoost optimized			
Train	3.4	9.9	-0.0
Test	17.8	48.2	4.2

```
# AdaBoost Optimized Forecast for the next month
df = import_data()
X_train, Y_train, X_test, Y_test = datasets(df, x_len=12, y_len=1, test_loops=0)

ada = AdaBoostRegressor(DecisionTreeRegressor(max_depth=12),
                        n_estimators=100,
                        learning_rate=0.01,
                        loss='exponential')

ada = ada.fit(X_train, Y_train)
forecast  = pd.DataFrame(data=ada.predict(X_test),index=df.index)
print('AdaBoost Optimized Forecast for the next period')
display(forecast)
```

[19] ✓ 18.5s

AdaBoost Optimized Forecast for the next period

	0
Product	
Product 1	1547.000000
Product 10	814.233333
Product 11	1287.000000
Product 12	168.816901
Product 13	305.625000
...	...
Product 65	0.060267
Product 66	9.000000
Product 7	288.857143
Product 8	658.487603
Product 9	118.000000

66 rows × 1 columns

WRAP-UP

BEST OPTIMAL MACHINE LEARNING MODEL FOR OUR PRODUCTS

DATA SCIENCE FOR SUPPLY CHAIN FORECASTING

FOR MAE (TEST SET)

Forestx200	MAE	RMSE	Bias
Train	11.9	30.4	-0.0
Test	17.5	45.4	2.3

Forest Optimized x 200 Forecast for the next period

Product	0
Product 1	1525.167175
Product 10	804.875529
Product 11	1078.683406
Product 12	169.169076
Product 13	305.843355
...	...
Product 65	0.086483
Product 66	9.101335
Product 7	295.069647
Product 8	692.907534
Product 9	121.680588

66 rows × 1 columns

FOR RMSE (TEST SET)

ETRx200	MAE	RMSE	Bias
Train	14.1	35.6	-0.0
Test	17.6	44.9	2.4

ETR Optimized x 200 Forecast for the next period

Product	0
Product 1	1508.069742
Product 10	809.593151
Product 11	1085.162544
Product 12	170.495017
Product 13	288.021175
...	...
Product 65	0.233671
Product 66	8.463769
Product 7	284.940312
Product 8	663.120976
Product 9	141.181486

66 rows × 1 columns

DATA SCIENCE FOR SUPPLY CHAIN FORECASTING

THANKS.

Diego Beteta
linkedin.com/in/diego-beteta/