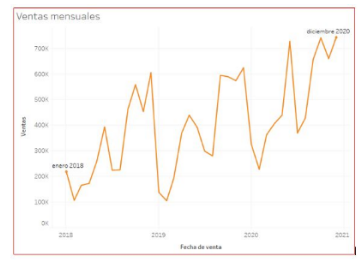
**Series de Tiempo I**

**Serie de Tiempo**:Conjuntode observaciones tomadas en *intervalos regulares* (**lags**), **ordenadas por el tiempo** en que se produjeron.

**Análisis de Series de Tiempo** (Time Series Analysis): Son **métodos para proyectar la evolución**, **obtener estadísticas y otras características** de una **variable** a lo **largo del tiempo**.

**Pronóstico de Series de Tiempo** (Time Series Forecasting): Usar modelos para predecir valores a futuro.

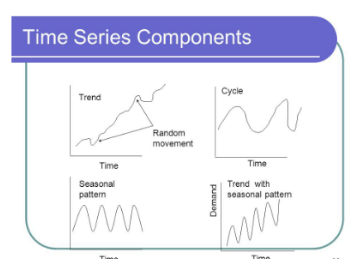
La serie de tiempo se proyecta como una variable (y) en función del tiempo (x):



Datos que podemos expresar como series de tiempo: Variables macroeconómicas (PBI, inflación, reservas de BCRA); transacciones comerciales (ventas, compras, pago a proveedores); datos de producción (unidades producidas, material consumido); consumo energético; activos financieros; variables sociales (mortalidad infantil, pobreza).

Una serie de tiempo puede tener **4 componentes:**

1. **Tendencia** (Trend): componente permanente; persiste en el tiempo.
2. **Estacionalidad** (Seasonality): Patrón estacional que se repite con regularidad.
3. **Componente Aleatoria** (Residual - Remainder): Shocks que no presentan un efecto duradero; Ruido; Movimientos Random.
4. **Ciclos** (Cycle): Otro tipo de dinámica que no la captura ni la tendencia ni la estacionalidad:



En Python contamos con el módulo **statsmodel.api**, que a su vez cuenta con el método **tsa.seasonal\_decompose**, que nos sirve para generar gráficos con la serie de tiempo, su tendencia, estacionalidad y componente aleatorio.

**En Python:**

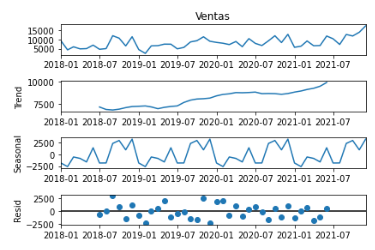
import statsmodel.api as sm

plt.figure(figsize=(16,10))

decomposition = sm.tsa.seasonal\_decompose(y, model = ‘additive’)

decomposition.plot()

plt.show()



Tenemos 2 maneras de descomponer una serie de tiempo en sus componentes:

1. **Descomposición Aditiva:**

****

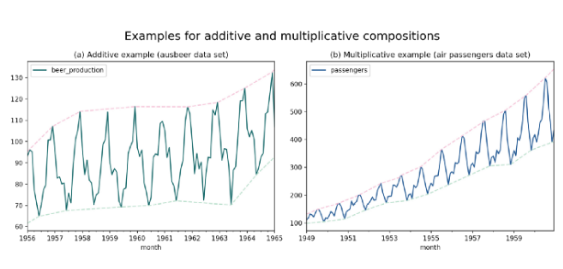
Conviene más en aquellos casos donde la variación estacional se mantiene relativamente constante (lineal). Es decir, que la altura de sus picos es constante.

1. **Descomposición Multiplicativa:**

****

Conviene más en aquellos casos donde la tendencia crece o decrece y la amplitud de la variación estacional aumenta o disminuye (exponencial).

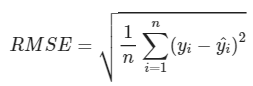
T es la Tendencia; S es la Estacionalidad; S es el ciclo; e es el error.



**Modelos Básicos para Pronósticos**:

Vamos a estudiar modelos aplicados a los componentes de las series de tiempo para generar predicciones. Para cada modelo vamos a definirlo; ajustarlo a los datos de entrenamiento, evaluarlo usando RMSE (error cuadrático medio raiz) sobre los datos de testeo y compararlo con otros modelos.

RMSE: Resulta de comparar los datos predichos con los valores observados:



**En Python:**

def RMSE(predicted, actual):

mse = (predicted – actual) \*\* 2

rmse = np.sqrt(mse.sum() / mse.count())

return rmse

**Dataset**: Vamos a trabajar con la información de ventas de una empresa multinacional en varios países, entre los años 2018 y 2021; considerando las ventas de la categoría Muebles.

df = pd.read\_excel(‘../Data/Supertienda.xls’)

mobiliario = df.loc[df[‘Categoría’] == ‘Mobiliario’]

mobiliario.head(2)



Vamos a mensualizar las ventas por fecha de pedido. Arrancamos agrupando las ventas por el atributo ‘Fecha\_pedido’ y creando un índice sobre dicho atributo para indicar la fecha en que se usa la serie de tiempo:

mobiliariogb = mobiliario.groupby(‘Fecha\_\_pedido’)[‘Ventas’].sum().reset\_index()

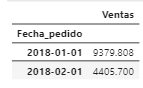
mobiliariogb = mobiliariogb.set\_index(‘Fecha\_pedido’)

Con la función **resample** de Pandas vamos a llevar la serie a un espacio de tiempo más amplio (**downsampling**)**.** En este caso lo haremos pasando de días a meses; **MS** indica **comienzo del mes**.

Con **mean()** vamos a agrupar todas las ventas del mes y calcular su promedio:

df\_ventas = mobiliariogb[‘Ventas’].resample(‘MS’).mean().to\_frame()

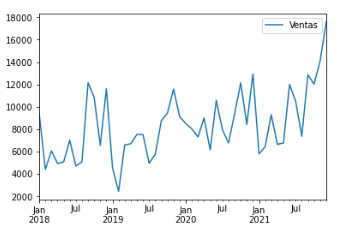
df\_ventas.head(2)



plt.figure(figsize=(4,3))

df\_ventas.plot()

plt.show()



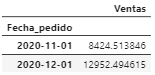
Ahora, vamos a dividir el dataset en train y test. Pondremos shuffle = False por tratarse de una serie de tiempo.

from sklearn.model\_selection import traain\_test\_split

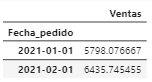
df\_train, df\_test = train\_test\_split(df\_ventas, test\_size = 12, random\_state = 42, shuffle = False)

Para datos de entrenamiento va a considerar los primeros tres años, mientras que para test va a tomar el último año.

df\_train.tail(2)



df\_test.head(2)



**Random Walk:** Se dice que un proceso Yt es **random walk** (sigue una trayectoria al azar) si:



Siendo ε **ruido blanco** (white noise). Es la parte de la serie que no se puede predecir a partir de la evidencia.

Si se trata de un modelo random walk con tendencia, representamos la misma con una d llamada **drift** (deriva). Entonces tendremos un **Modelo Random Walk with Drift**:



Con respecto a la serie de white noise ε, tiene las siguientes características:

* Media µ = 0 y varianza constante: 
* Completamente random. Por eso se lo llama ruido.
* Observaciones Independientes Idénticamente Distribuidas (iid).
* No tienen correlación entre sí.

En caso de que estas observaciones tengan una distribución normal, se lo llama **Gaussian White Noise** (Ruido Blanco Gaussiano).

Se puede aplicar un **pronóstico ingenuo** (naive), que consiste en suponer que el valor de la serie Yt será igual a su valor previo Yt-1. Ejemplo: Estimación en el valor de las acciones.

**En Python:**

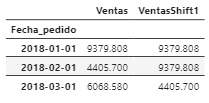
Vamos a generar el desplazamiento de nuestro target en el set de entrenamiento (lag = 1):

df\_train[‘VentasShift1’] = df\_train.Ventas.shift()

# la primer observación quedará en Nan. Vamos a reemplazarla por el valor siguiente:

df\_train[‘VentasShift1’].fillna(method = ‘bfill’, inplace = True)

df\_train.head(3)



Ahora vamos a generar el desplazamiento en el set de testeo (lag = 1):

df\_test[‘VentasShift1’] = df\_test.Ventas.shift()

df\_test[‘VentasShift1’].fillna(method=’bfill’, inplace = True)

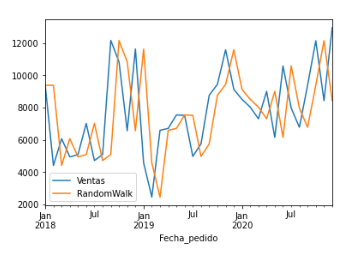
df\_test.head(3)



Vamos a definir como Random Walk al desplazamiento en un valor (shift = 1).

df\_train[‘RandomWalk’] = df\_train.VentasShift1

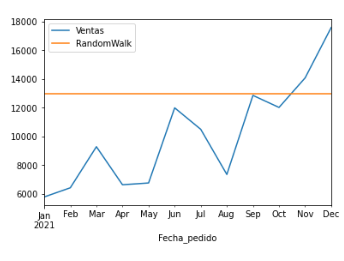
df\_train.plot(kind=’line’, y = [‘Ventas’, ‘RandomWalk’])



La predicción sobre test va a ser la última observada en train:

df\_test[‘RandomWalk’] = pd.Series(df\_train[‘Ventas’][-1], index = df\_test.index)

df\_test.plot(kind=’line’, y = [‘Ventas’, ‘RandomWalk’])



Ahora vamos a pasar a calcular el RMSE y almacenar los resultados:

df\_Results = pd.DataFrame(columns = [‘Model’, ‘RMSE’])

df\_Results.loc[0, ‘Model’] = ‘Random Walk’

df\_Results.loc[0, ‘RMSE’] = RMSE(df\_test.RandomWalk, df\_test.Ventas)

df\_Results



**Media Constante:** Modelo más básico e ingenuo de todos: Es tomar la media del dataset de entrenamiento y usarlo para predecir a futuro.

**En Python:**

model\_mean\_pred = df\_train[‘Ventas’].mean()

model\_mean\_pred



Se aplica la media tanto a entrenamiento como a testeo:

df\_train\_mc = df\_train.copy()

df\_train\_mc[‘Mean’] = model\_mean\_pred

df\_test\_mc = df\_test.copy()

df\_test\_mc[‘Mean’] = model\_mean\_pred

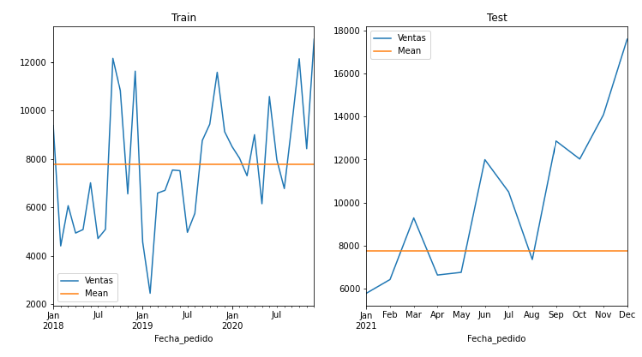
A simple vista, podemos ver que el método no predice correctamente:

fig, axes = plt.subplots(1, 2, figsize = (12, 6))

df\_train\_mc.plot(ax = axes[0], y = [‘Ventas’, ‘Mean’], title = ‘Train’)

df\_test\_mc.plot(ax = axes[1], y = [‘Ventas’, ‘Mean’], title = ‘Train’)

plt.show()



Calculamos la performance con RMSE y la guardamos para seguir comparando con otros modelos:

model\_RMSE = RMSE(df\_test\_mc.Mean, df\_test\_mc.Ventas)

df\_Results.loc[1, ‘Model’] = ‘Mean’

df\_Results.loc[1, ‘RMSE’] = model\_RMSE

df\_Results.head()



**Tendencia Lineal:**

El **componente** de la serie **tendencia** corresponde a una **evolución de largo plazo**. Es posible modelarla de diferentes maneras, según su dinámica. La **tendencia lineal** representa una recta para predecir:

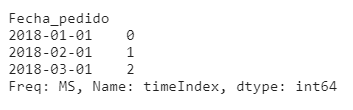


Siendo **TIME** la llamada “Dummy de Tiempo”. Se trata de una secuencia que representa el tiempo.

**En Python:**

df\_ventas[‘timeIndex’] = pd.Series(np.arange(len(df\_ventas[‘Ventas’])), index = df\_ventas.index)

df\_ventas.timeIndex.hea(3)



Ahora vamos a hacer los sets de entrenamiento y testeo con esta nueva variable:

from sklearn.model\_selection import train\_test\_split

df\_train, df\_test = train\_test\_split(df\_ventas, test\_size = 12, random\_state = 42, shuffle = False)

Vamos a usar un modelo de regresión lineal entre ‘Ventas’ y ‘TimeIndex’ para generar la recta. Vamos a usar **statsmodel** para ajustar modelos con us API; usando fórmulas estilo R. Con la sentencia **smf.ols(formulas = ‘Ventas ~ timeIndex’, data = df\_train).fit()** podemos hacer una regresión lineal (ols) con dos variables (formula).

**En Python:**

import statsmodel.api as smf

model\_linear = smf.ols(formula = ‘Ventas ~ timeIndex’, data = df\_train).fit()

Vamos a generar las predicciones sobre train y test:

df\_train[‘LinearTrend’] = model\_linear.predict(df\_train.timeIndex)

df\_test[‘LinearTrend’] = model\_linear.predict(df\_test.timeIndex)

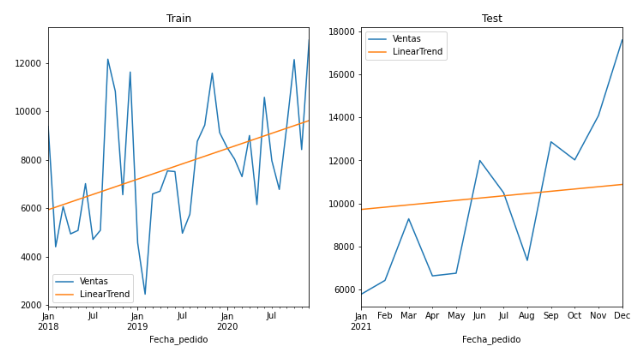
Visualmente, vamos a ver que la recta se ajusta mejor que la media constante:

fig, axes = plt.subplots(1, 2, figsize = (12, 6))

df\_train.plot(kind = ‘line’, y = [‘Ventas’, ‘LinearTrend’], ax = axes[0], title =’Train’)

df\_test.plot(kind = ‘line’, y = [‘Ventas’, ‘LinearTrend’], ax = axes[1], title =Test’)

plt.show()



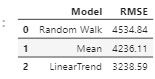
Vamos a calcular la performance con RMSE. De esta forma, vamos a confirmar analíticamente que este método es efectivamente mejor que Mean.

model\_RMSE = RMSE(df\_test.LinearTrend, df\_test.Ventas)

df\_Results.loc[2, ‘Model’] = ‘LinearTrend’

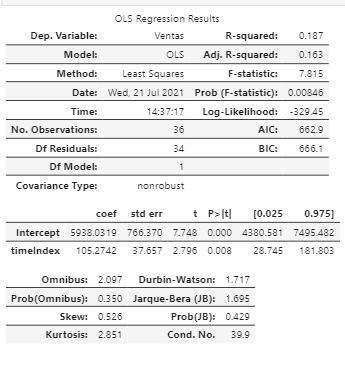
df\_Results.loc[2, ‘RMSE’] = model\_RMSE

df\_Results.head()



Con statsmodels, podemos ver los resultados de la regresión lineal:

model\_linear.summary()



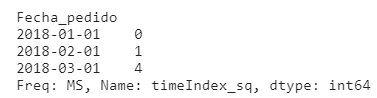
**Tendencia Cuadrática**: En caso de que la tendencia presente una dinámica no lineal, podemos incluirle un término cuadrático al modelo. Logramos esto elevando la dummy del tiempo al cuadrado:



**En Python:**

df\_ventas[‘timeIndex\_sq’] = df\_ventas[‘timeIndex’] \*\* 2

df\_ventas.tieIndex\_sq.head(3)



Generamos los datasets de entrenamiento y testeo con la nueva variable:

from sklearn.model\_selection import train\_test\_split

df\_train, df\_test = train\_test\_split(df\_ventas, test\_size = 12, random\_state = 42, shuffle = False)

Vamos a incorporar el valor cuadrático de la fórmula al modelo:

import statsmodel.formula.api as smf

model\_quadratic = smf.ols(‘Ventas ~ timeIndex + timeIndex\_sq’, data = df\_train).fit()

Ahora vamos a generar las predicciones del modelo sobre el set de entrenamiento y de testeo:

df\_train[‘QuadraticTrend’] = model\_quatratic.predict(df\_train[[‘timeIndex’, ‘timeIndex\_sq’]])

df\_test[‘QuadraticTrend’] = model\_quatratic.predict(df\_test[[‘timeIndex’, ‘timeIndex\_sq’]])

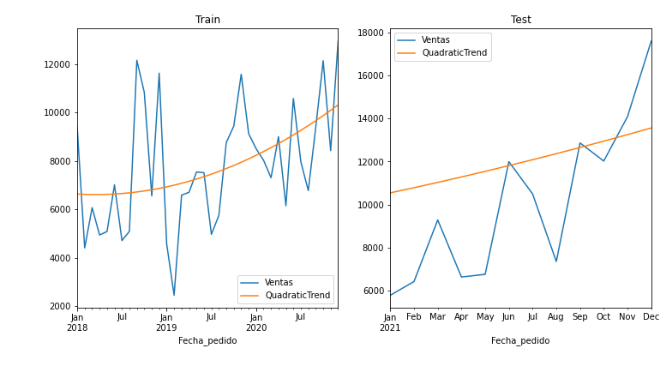
Vamos a ver visualmente las predicciones:

Fig, axes = plt.subplots(1, 2, figsize = (12, 6))

df\_train.plot(kind = ‘line’, y = [‘Ventas’, ‘QuadraticTrend’], ax = axes[0], title = ‘train’)

df\_test.plot(kind = ‘line’, y = [‘Ventas’, ‘QuadraticTrend’], ax = axes[1], title = ‘test’)

plt.show()



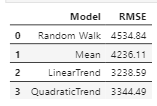
Vamos a calcular la performance con RMSE. Vemos que conviene ajustar con tendencia líneal que con tendencia cuadrática:

Model\_RMSE = RMSE(df\_test.QuadraticTrend, df\_test.Ventas)

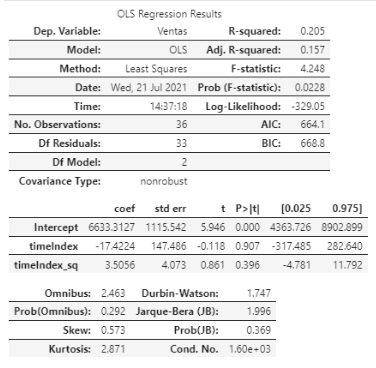
df\_Results.loc[3, ‘Model’] = ‘QuadraticTrend’

df\_Results.loc[3, ‘RMSE’] = model\_RMSE

df\_Results.head()



model\_quadratic.summary()



**Tendencia Con Transformación Logarítmica**: A veces, la varianza de la serie va aumentando con el paso del tiempo. Si hacemos una transformación logarítmica de la serie de tiempo, podemos ayudar a estabilizar la varianza. Entonces el modelo a generar será sobre el logaritmo de las ventas y sus predicciones en lugar de sobre las ventas directamente. Las predicciones luego se transformará en un valor de ventas con una función exponencial y terminaremos evaluando el modelo con este último valor.

**En Python:**

df\_train[‘log\_Ventas’] = np.log(df\_train[’Ventas’])

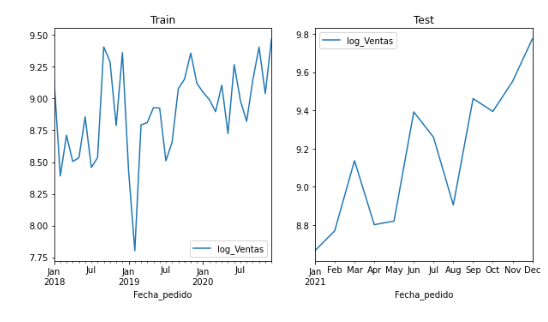
df\_test[‘log\_Ventas’] = np.log(df\_test[’Ventas’])

fig, axes = plt.subplots(1, 2, figsize = (10, 5))

df\_train.plot(kind = ‘line’, y = [‘log\_Ventas’], ax = axes[0], title = ‘Train’)

df\_test.plot(kind = ‘line’, y = [‘log\_Ventas’], ax = axes[1], title = ‘Test’)

plt.show()



Vamos a generar el modelo de regresión lineal usando el logaritmo de las ventas:

import statsmodel as smf

model\_log = smf.ols(‘log\_Ventas ~ timeIndex’, data = df\_train).fit()

Predecimos sobre los sets de entrenamiento y testeo:

df\_train[‘LogTrend’] = model\_log.predict(df\_train[[‘timeIndex]])

df\_test[‘LogTrend’] = model\_log.predict(df\_test[[‘timeIndex]])

El modelo está prediciendo logaritmos de Ventas, pero lo que nos interesa conocer realmente, son los valores de Ventas que corresponden a estos logaritmos. Entonces:

df\_train[‘back\_Logtrend’] = np.exp(df\_train[‘LogTrend’])

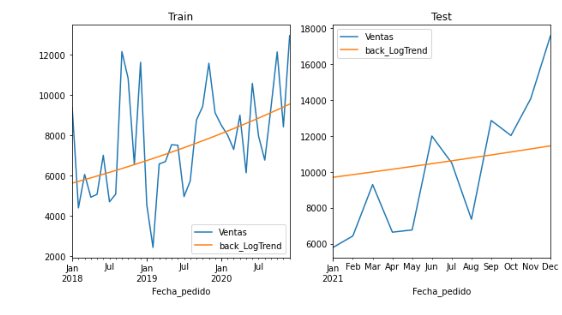
df\_test[‘back\_Logtrend’] = np.exp(df\_test[‘LogTrend’])

Vamos a estudiar visualmente las diferencias entre las ventas predichas y las ventas reales:

fig, axes = plt.subplots(1, 2, figsize = (10, 5))

df\_train.plot(kind = ‘line’, y = [‘Ventas’, ‘back\_LogTrend’], ax = axes[0], title = ‘Train’)

df\_test.plot(kind = ‘line’, y = [‘Ventas’, ‘back\_LogTrend’], ax = axes[1], title = ‘Test’)



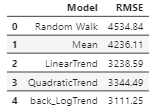
Vamos a calculary la performance de este modelo con RMSE:

model\_RMSE = RMSE(df\_test.back\_LogTrend, df\_test.Ventas)

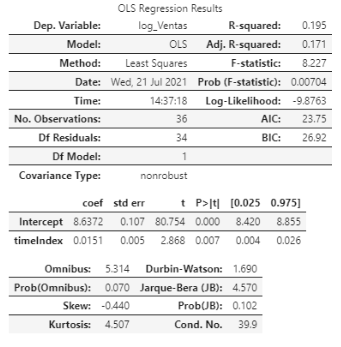
df\_Results.loc[4, ‘Model’] = ‘back\_LogTrend’

df\_Results.loc[4, ‘RMSE’] = model\_RMSE

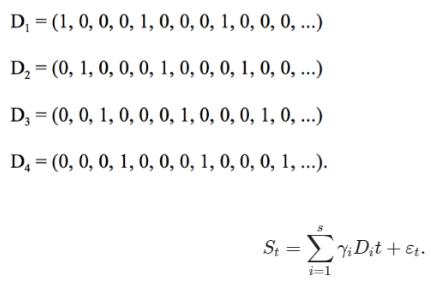
df\_Results.head()



model\_log.summary()



**Transformación Logarítmica:** La **estacionalidad** es un **patrón estacional** que **se repite con regularidad**. Es posible modelar la estacionalidad usando variables dummy. En este ejemplo modelamos una estacionalidad trimestral:



Es posible crear otros tipos de dummies, tales como vacaciones, feriados, entre otros.

**En Python:**

Vamos a establecer variables dummy para los meses de la fecha de pedido de ventas. Empezamos creando la columna con el mes:

df\_train[‘month’] = [d.strftime(‘%b’) for d in df\_train.index]

df\_test[‘month’] = [d.strftime(‘%b’) for d in df\_test.index]

Ahroa creamos las variables dummy del mes:

dummies\_mes\_train = pd.get\_dummies(df\_train[‘month’])

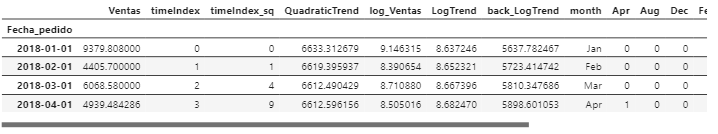
dummies\_mes\_test = pd.get\_dummies(df\_test[‘month’])

Ahora hacemos el join entre el DataFrame y la serie de tiempo:

df\_train = df\_train.join(dummies\_mes\_train)

df\_test = df\_test.join(dummies\_mes\_train)

df\_train.head(4)



Ahora vamos a generar el modelo de regresión lineal, agregándole al logaritmo de las ventas la estacionalidad (las variables dummies del mes):

model\_log\_est = smf.ols(‘log\_Ventas ~ timeIndex + Apr + Aug + Dec + Feb + Jan + Jul + Jun + Mar + May + Nov + Oct + Sep’, data = df\_train).fit()

Predecimos sobre train y test:

df\_train[’model\_log\_est’] = model\_log\_est.predict(df\_train[[‘timeIndex’, ‘Apr’, ‘Aug’, ‘Dec’, ‘Feb’, ‘Jan’, ‘Jul’, ‘Jun’, ‘Mar’, ‘May’, ‘Nov’, ‘Oct’, ‘Sep’]])

df\_test[’model\_log\_est’] = model\_log\_est.predict(df\_test[[‘timeIndex’, ‘Apr’, ‘Aug’, ‘Dec’, ‘Feb’, ‘Jan’, ‘Jul’, ‘Jun’, ‘Mar’, ‘May’, ‘Nov’, ‘Oct’, ‘Sep’]])

Aplicando la función exponencial calculamos el valor predicho de la venta a partir del valor predicho del algoritmo de la venta:

df\_train[‘back\_LogEstTrend’] = np.exp(df\_train[‘model\_log\_est’])

df\_test[‘back\_LogEstTrend’] = np.exp(df\_test[‘model\_log\_est’])

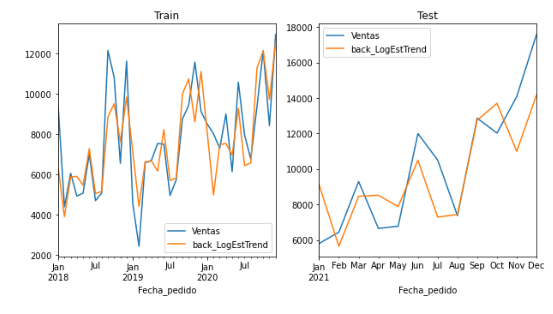
Vamos a graficar las ventas predichas vs el valor real:

fig, axes = plt.subplots(1, 2, figsize = (10,5))

df\_train.plot(kind = ‘line’, y = [‘Ventas’, ‘back\_logEstTrend’], ax = axes[0], title = ‘Train’)

df\_test.plot(kind = ‘line’, y = [‘Ventas’, ‘back\_logEstTrend’], ax = axes[1], title = ‘Test’)

plt.show()



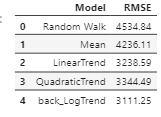
Calculamos la performance con RMSE. En este caso, el modelo ajusta mejor que todo lo que probamos hasta ahora.

model\_RMSE = RMSE(df\_test.back\_LogEstTrend, df\_test.Ventas)

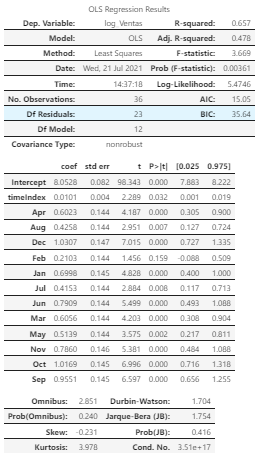
df\_Results.loc[5, ‘Model’] = ‘back\_LogEstTrend’

df\_Results.loc[5, ‘RMSE’] = model\_RMSE

df\_Results.head()

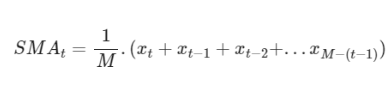


model\_log\_est.summary()



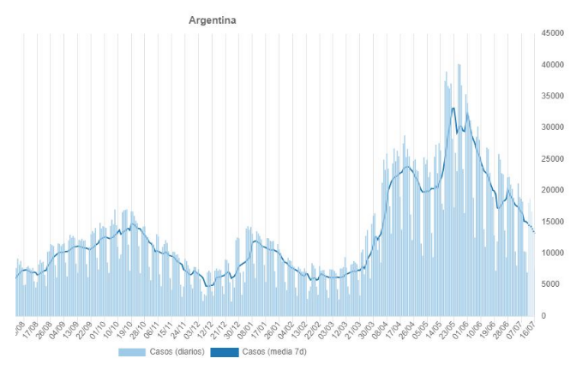
**Media Movil**: (Moving Average o Rolling Mean) promedio de los valores de una serie temporal para un **período de tiempo determinado (Ventana de Estimación)**. El **tamaño de la ventana M** es la cantidad de observaciones usadas para estimar.

**Media Móvil Simple:**

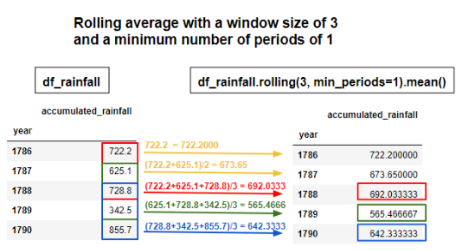


El valor para el tiempo t es un promedio de los M valores reales previos. Es un indicador de **tendencia** porque produce un suavizado sobre la serie temporal. No la vamos a usar para predecir, pero nos puede servir como método de preprocesamiento de la serie.

En este ejemplo podemos ver la diferencia entre casos diarios y la media móvil semanal de casos de Covid-19:



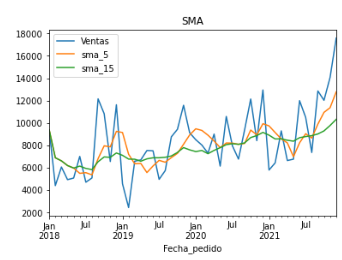
Con el método de Pandas **Series.rolling** podemos calcular la SMA de una forma fácil. Si bien las primeras filas van a contener valores nulos, ya que puede calcular recién a partir de las M observaciones, podemos sortear esto con **min\_periods = 1**.



A mayor ventana, más suave será la serie calculada:

df\_ventas[‘sma\_5’] = df\_ventas.Ventas.rolling(5, min\_periods = 1).mean()

df\_ventas[‘sma\_15’] = df\_ventas.Ventas.rolling(15, min\_periods = 1).mean()

df\_ventas.plot(kind = ‘line’, y = [‘Ventas’, ‘sma\_5’, ‘sma\_15’], title = ‘SMA’)

**Single Exponential Smoothing:** En el **Suavizamiento Exponencial Simple**, los pesos siguen una caída exponencial, donde se le da más peso a las observaciones más recientes y menos a las más antiguas.

Matemáticamente:



t es la predicción para el período t; la cual es un promedio ponderado entre el valor predicho de y en el período t-1 (t-1)y el valor real de y en el período t-1 (yt-1).

Α es conocido como el **smoothing parameter**, y puede variar entre 0 y 1. Cuando sea 1, las predicciones serán iguales al último valor observado (**método naive**).

Este método sirve para pronosticar datos sin tendencia ni patrón estacional claro.

Con la media constante, todas las predicciones serían iguales al promedio de los valores observados.

La **expresión generalizada** del **Single Exponential Smoothing** es la siguiente:



De esta forma, el peso de las observaciones decrece de manera exponencial a medida que nos alejamos en el tiempo. α de ratio de caída: Cuanto más cercano a 1, más rápido decaen los pesos.

**En Python:**

Podemos generar 3 modelos variando el **smoothing factor α** (0.3, 0.5 & 0.8):

from statsmodel.tsa.holtwinters import SimpleExpSmoothing

model\_exp\_smoothing\_3 = SimpleExpSmoothing(df\_train.Ventas).fit(smoothing\_level = 0.3, optimized = False)

model\_exp\_smoothing\_5 = SimpleExpSmoothing(df\_train.Ventas).fit(smoothing\_level = 0.5, optimized = False)

model\_exp\_smoothing\_8 = SimpleExpSmoothing(df\_train.Ventas).fit(smoothing\_level = 0.8, optimized = False)

Vamos a ver los resultados sobre los datos de train:

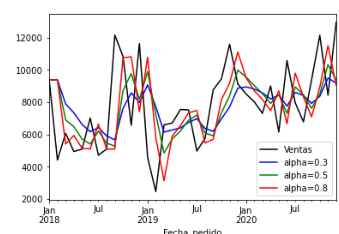
df\_train.plot(kind = ‘line’, y = ‘Ventas’, color = ‘black’)

model\_exp\_smoothing\_3.fittedvalues.plot(label=’alpha=0.3’, color = ‘b’)

model\_exp\_smoothing\_5.fittedvalues.plot(label=’alpha=0.5’, color = ‘g’)

model\_exp\_smoothing\_8.fittedvalues.plot(label=’alpha=0.8’, color = ‘r’)

plt.legend()



Vamos a predecir sobre test:

df\_test[‘Simple\_smoothing\_3’] = model\_exp\_smoothing\_3.forecast(len(df\_test))

df\_test[‘Simple\_smoothing\_5’] = model\_exp\_smoothing\_5.forecast(len(df\_test))

df\_test[‘Simple\_smoothing\_8’] = model\_exp\_smoothing\_8.forecast(len(df\_test))

Entre estos 3 modelos, vemos que el que mejor predice es el de α = 0.3

model\_RMSE\_3 = RMSE(df\_test[‘Simple\_smoothin\_3’], df\_test.Ventas)

model\_RMSE\_5 = RMSE(df\_test[‘Simple\_smoothin\_5’], df\_test.Ventas)

model\_RMSE\_8 = RMSE(df\_test[‘Simple\_smoothin\_8’], df\_test.Ventas)

df\_Results.loc[6, ‘Model’] = ‘Simple Smoothing a=0.3’

df\_Results.loc[6, ‘RMSE’] = model\_RMSE\_3

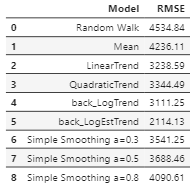
df\_Results.loc[7, ‘Model’] = ‘Simple Smoothing a=0.5’

df\_Results.loc[7, ‘RMSE’] = model\_RMSE\_5

df\_Results.loc[8, ‘Model’] = ‘Simple Smoothing a=0.8’

df\_Results.loc[8, ‘RMSE’] = model\_RMSE\_8

df\_Results



**Conclusiones:**

* Una **serie de tiempo** cuenta con los **componentes:** **Tendencia** (componente ‘permanente’; efecto que persiste en el tiempo a largo plazo)**, Estacionalidad** (movimientos periódicos de la serie)**, Ciclo** (dinámica no capturada por la tendencia o la estacionalidad)**, Componente Aleatoria** (shocks que no presentan efecto duradero)**.**
* La **tendencia** y la **estacionalidad** **pueden modelarse** con **dummies de tiempo y estacionales.**
* Existen **varios modelos** para **predecir** los **nuevos valores** de la **serie** de tiempo: **Tendencia Lineal, Tendencia Cuadrática, Transformación Logarítmica con y sin estacionalidad menusal, Single Exponential Smoothing**, entre otros.
* Con la **media móvil** es posible **suavizar la serie**.