

# Predicción con Series de Tiempo

**Un problema de predicción involucra el uso de observaciones del pasado para predecir una o más posibles observaciones futuras.**

# Ejemplos

- Mercadeo/ventas: ¿Cómo serán nuestras ventas en el último trimestre?
- Salud: ¿Necesitaremos más camas en el hospital, el año entrante?
- Deportes: ¿Este año, cuándo bajará de 20 C la temperatura de la piscina?
- Energía: ¿Cuál será el consumo de energía de esta casa, mañana?

# Retos

## Las Series de Tiempo (ST)

- Tienen una variable de entrada:
  - Son endógenas - pueden depender de otras y la salida depende de ésta
  - Pueden tener variables exógenas - que no influyen en las otras
- Pueden tener un patrón obvio: tendencia, ciclos estacionales
- Predicen el siguiente paso en tiempo, pero muchas veces queremos tener más. Esto hace más difícil la predicción
- Pueden ser representadas por modelos que son:
  - Estáticos...funcionan bien con el tiempo y no necesitan actualizaciones
  - Dinámicos...por ejemplo, hay que volver a entrenarlo cada semana

Hay un sinnúmero de formas de predecir las ST, algunas pueden reflejar un número de retos diferentes y otras no. Cuál se selecciona depende del problema que tenemos a mano.

# Principio de la Navaja de Occam (William de Ockham)



Debemos escoger la solución con el  
menor número de suposiciones



# Nuestra progresión en ST

1. Establecer una base de referencia simple e.g. promediar los datos (línea base)
2. Probar modelos SARIMA (Seasonal Autoregressive Integrated Moving Average)
3. Probar “alisamiento” exponencial (funciones exponenciales vrs. lineales)
4. Probar una red neuronal simple
5. Probar aprendizaje profundo (CNN, LSTM, etc)

\*\* Normalmente con los pasos 1 al 3 es suficiente

# ¿Cómo entrenar nuestros modelos?

En ML normalmente partimos los datos en conjuntos de entrenamiento y conjuntos de prueba...mejor si son balanceados,

En ST no se puede, porque los datos tienen un orden. Por esto se hace una función adhoc:

```
def division_entreno_prueba(datos, n_prueba):  
    return datos[: -n_prueba], datos[-n_prueba:]
```



# ¿Cómo probar nuestros modelos?

Luego de ajustar el modelo con nuestros datos, queremos hacer una predicción de una observación nueva, y luego compararla con el valor real que iba a venir en seguida. Para esto usamos el RMSE que es estándar en ML.

```
def medir_rmse(actual, predicho):  
    return sqrt(mean_squared_error(actual, predicho))
```

# ¿Cómo probar nuestros modelos?

Se debe probar el modelo con todas las observaciones del conjunto de prueba.

Esto implica partir el modelo múltiples veces, cada vez agregando un punto más al conjunto de datos de entrenamiento, y ver qué predice.

A este proceso de dividir los datos y ver hacia adelante se le denomina **validación hacia adelante (walk forward validation)**

# ¿Cómo probar nuestros modelos?

```
def validacion_al_frente(datos, n_prueba):  
    predicciones = []  
  
    entreno, prueba = division_entreno_prueba(datos, n_prueba)  
  
    modelo = model_fit(entreno)  
  
    historia = [x for x in entreno]  
  
    # Caminar hacia adelante  
    for i in range(len(test)):  
  
        # Ajustar el modelo y predecir  
        yhat = model_predict(modelo, historia)  
        predicciones.append(yhat)  
        historia.append(prueba[i])  
  
    # Estimar el error  
    error = medir_rmse(prueba, predicciones)  
  
    print(f"> {error:.3f}")
```

# ¿Cómo afinar los modelos?

- Los diferentes métodos utilizan una variedad de hiper-parámetros
- No es posible saber qué hiper-parámetros y valores darán los mejores resultados
- **Búsqueda en Malla (Grid Search)** - probar todas las combinaciones y ver cuáles dan mejor resultado.

Ej: Para el modelo de base de referencia, promedios, probar promediar los últimos 1, 2, 3, ... , N valores y ver cuál da el mejor resultado. Este sería tan solo uno de los hiper-parámetros.

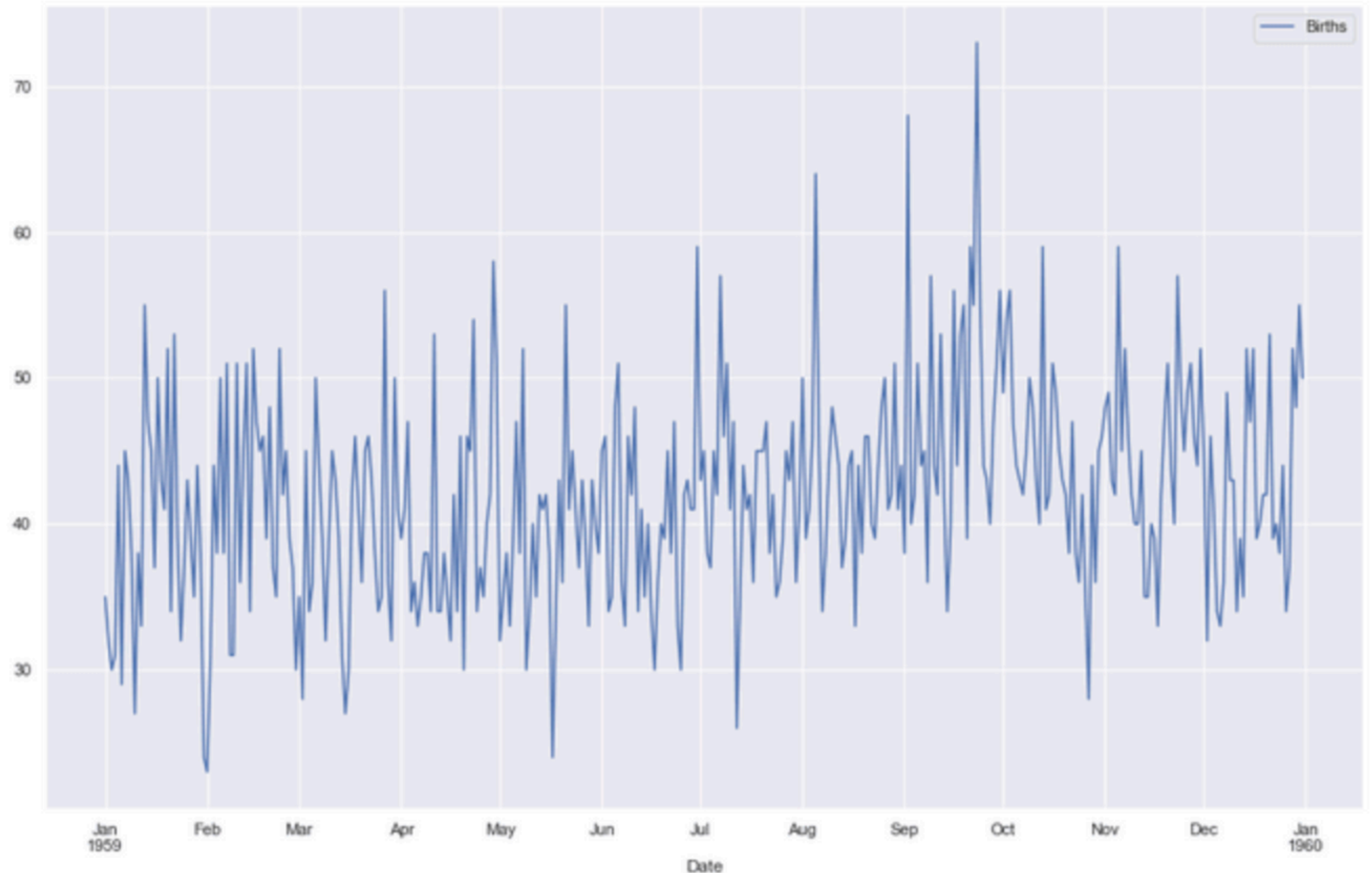
**Manos a la obra  
hacer predicciones**

**Para probar los modelos, se utilizarán 4 conjuntos de datos**



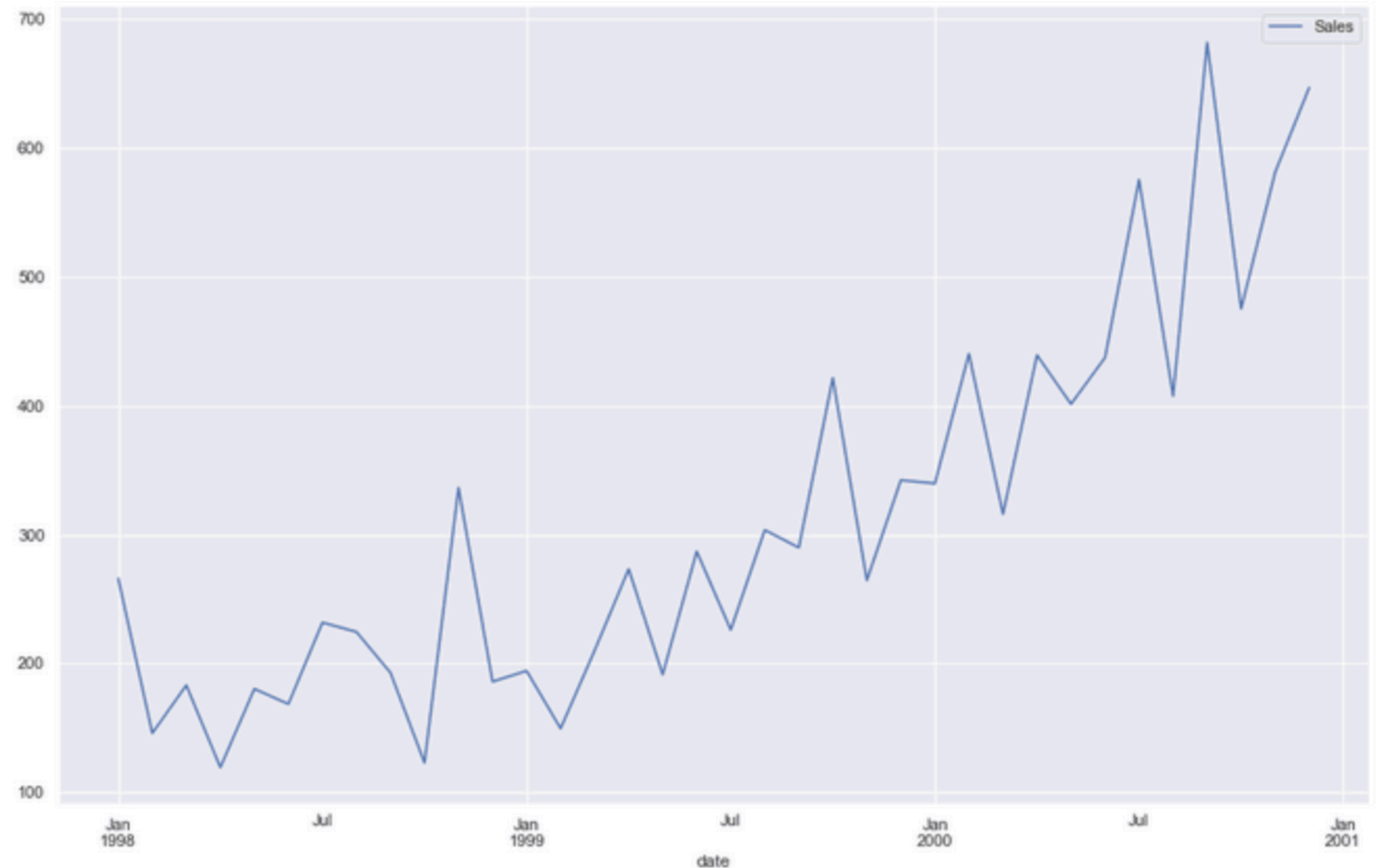
# Nacimientos de mujeres en California en 1959

- Este conjunto casi no tiene tendencia
- Problema: conocer los datos del año siguiente para saber si se necesitan más camas, personal, etc.
- En este conjunto de datos, un paso es un día



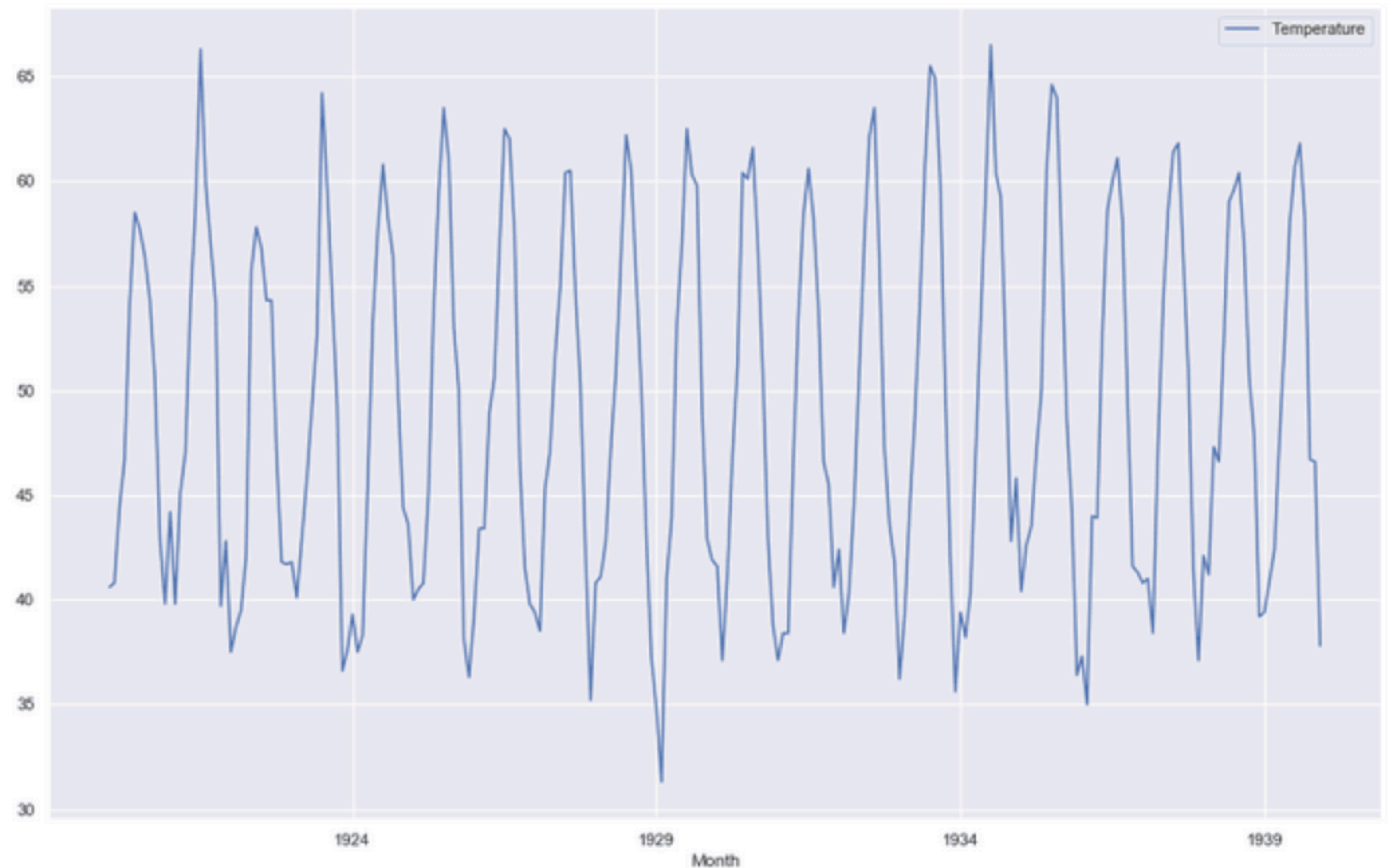
# Ventas de shampoo en un período de 3 años

- Este conjunto tiene tendencia
- Problema: conocer los datos del siguiente año para planificar la producción con tiempo
- En este conjunto de datos, un paso es un mes



# Temperatura promedio mensual en un período de 3 años

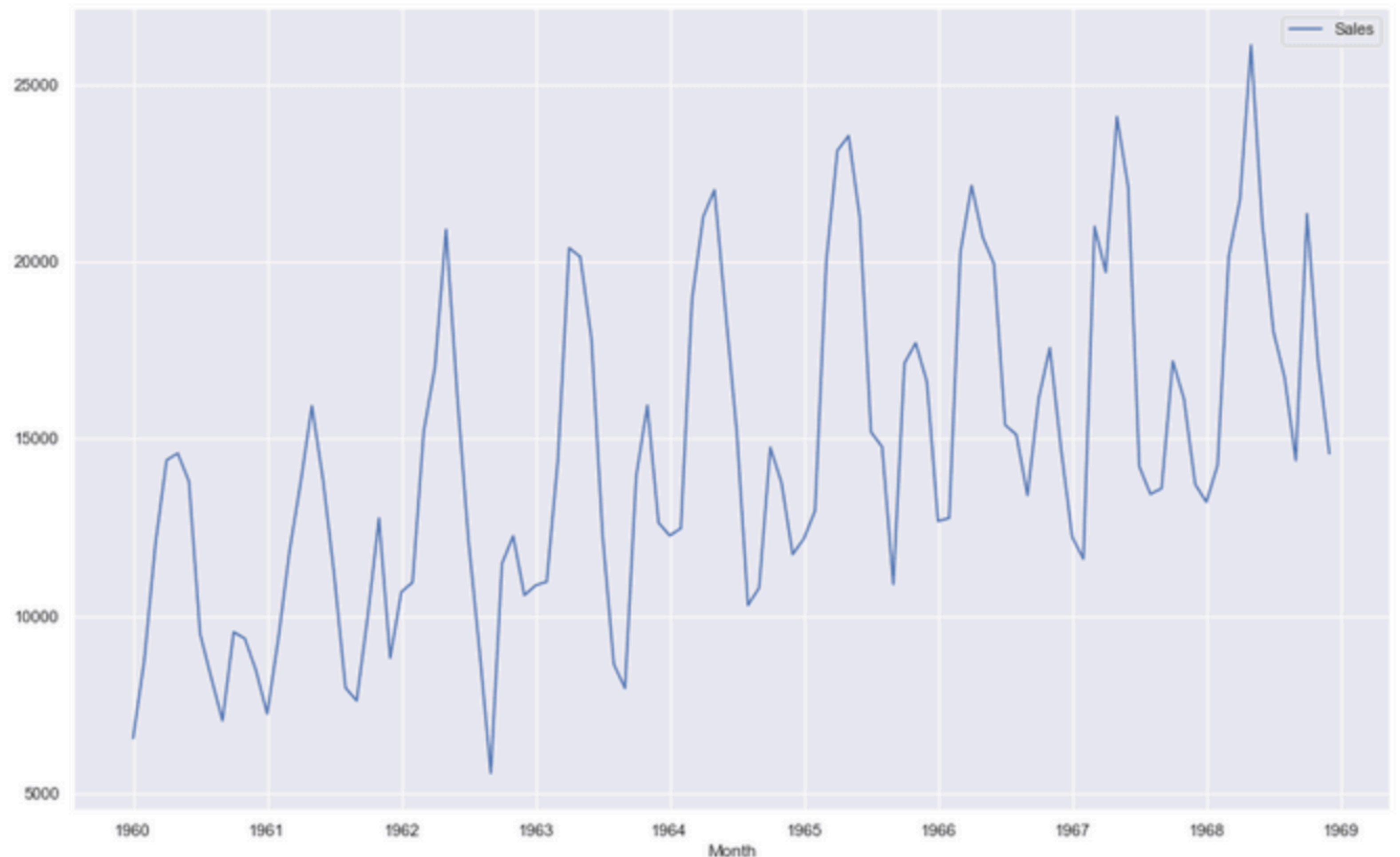
- Este conjunto no tiene tendencia pero tiene algo que se conoce como estacionalidad (seasonality)
- Problema: Compañía de helados necesita saber cuándo incrementar la producción-
- En este conjunto de datos, un paso es un mes





# Venta de Autos en Quebec en los años 60's

- Este conjunto tiene tendencia y tiene estacionalidad: parece que a la gente le gusta comprar en la primavera y el otoño
- Problema: Agencia necesita saber como manejar el inventario
- En este conjunto de datos, un paso es un mes



# Métodos para trabajar ST

# Modelo de base de referencia: Promedio

- Tomar los últimos N valores y obtener la media/mediana de los mismos

```
def prediccion_promedio(historia, config):
```

```
    n, tipo_prom = config
```

```
    if tipo_prom is "prom":
```

```
        return mean(historia[-n:])
```

```
    return median(historia[-n:])
```

- Aunque el modelo parezca un poco “tonto”, puede funcionar bien cuando hay algo de ruido y cuando queremos enfatizar fuertemente los últimos valores



# Modelos SARIMA

- Disponible a través de la librería **statmodels**
- Requiere básicamente 3 parámetros
  - **Order:** una tupla de parámetros **p**, **d** y **q** para modelar la tendencia. Controlan el orden de:
    - La autoregresión
    - La diferencia
    - El promedio en movimiento
  - **Seasonal order:** una tupla de parámetros **p**, **d**, **q** y **m** para modelar la estacionalidad. Controlan el orden de:
    - La autoregresión estacional
    - La diferencia estacional
    - El promedio en movimiento estacional
    - El número de pasos que contribuyen a un período estacional
  - **Trend:** un parámetro para controlar el modelo de la tendencia determinística. Puede ser **n** (ninguna tendencia), **c** (constante), **t** (lineal) y **ct** (constante con tendencia lineal)

# Modelos SARIMA

- Si sabemos lo suficiente del problema, podemos especificar estos parámetros correctamente. Si no, utilizamos el proceso de **Grid Search**

# Modelos SARIMA

```
def prediccion_sarima(historia, orden, orden_estacional, tendencia):  
    modelo = SARIMAX(historia, order = orden, seasona_order = orden_estacional,  
                     trend = tendencia, enforce_stationarity = False,  
                     enforce_invertibility = False)  
  
    modelo_ajustado = modelo.fit(dispatch = False)  
  
    yhat = modelo_ajustado.predict(len(historia), len(historia))  
  
    return yhat[0]
```

# Modelos de alisamiento exponencial o Winter-Holt

- Es un método de predicción de ST para datos univariados
- Con SARIMA la predicción es una simple suma lineal (con pesos) de observaciones del pasado
- Con estos modelos se usa un peso exponencialmente decreciente para observaciones del pasado
- Hay tres tipos de modelos de alisamiento exponencial:
  - Un método simple que no asume una estructura sistemática alguna
  - Una variante que explícitamente maneja tendencias
  - Una variante más sofisticada que adicionalmente maneja estacionalidad (este es el que utilizaremos )

# Modelos de alisamiento exponencial o Winter-Holt

- La implementación de la librería **statsmodels** trae un optimizador que automáticamente afina los siguientes hiper-parámetros:
  - alpha - el coeficiente de alisamiento para el nivel
  - beta - el coeficiente de alisamiento para la tendencia
  - gamma - el coeficiente de alisamiento para la estacionalidad
  - phi - el coeficiente de alisamiento para la tendencia amortiguada

# Modelos de alisamiento exponencial o Winter-Holt

- Los hiper-parámetros que se deben configurar manualmente (o con **Grid Search**) son:
  - **trend (t)** - tipo de componente de la tendencia: **add** (aditivo), **mul** (multiplicativo) o **None**
  - **damped (d)** - si el componente de tendencia debe ser amortiguado o no: **True** o **False**
  - **seasonality (s)** - tipo de componente de la estacionalidad: **add** (aditivo), **mul** (multiplicativo) o **None**
  - **Seasonal periods (p)** - el número de pasos de tiempo en un período estacional (ej: 12 para 12 meses en una estructura estacional anual)
  - **Boxcox (b)** - Si se debe realizar una transformación de potencia (power transform) de la serie
  - **Remove bias (r)** - Si el sesgo/tendencia debe eliminarse de los datos




# Modelos de alisamiento exponencial o Winter-Holt

```
def AlisamientoExponencial(historia, t, d, s, p, b, r):  
    historia = array(historia)  
  
    modelo = ExponentialSmoothing(historia, trend = t, damped = d,  
                                   seasonal = s, seasonal_periods = p)  
  
    modelo_ajustado = modelo.fit(optimized = True, use_boxcox = b,  
                                 remove_bias = r)  
  
    yhat = modelo_ajustado.predict(len(historia), len(historia))  
  
    return yhat[0]
```

# Redes Neuronales

- Requiere que los datos originales se enmarquen en un formato diferente:

tiempo	medición		X	Y
1	100		?	100
2	110		100	110
3	108		110	108
4	115		108	115
5	120		115	120
			120	?



- Esto se llama **aproximación de ventana o método de rezago (window approach o lag method)**. El número de estados anteriores es el tamaño de la ventana o del rezago...en este ejemplo es 1
- El beneficio es que, con hacer esto, podemos trabajar con cualquier método estándar de ML, sea lineal o no.

# Redes Neuronales

## Transformación de series

```
def series_a_supervisado(datos, n_entrada, n_salida = 1):  
    df = DataFrame(datos)  
  
    columnas = [ ]  
  
    for i in range(n_entrada, 0, -1):  
        columnas.append(df.shift(-1))  
  
    for i in range(0, n_salida):  
        columnas.append(df.shift(-1))  
  
    agregado = concat(columnas, axis = 1)  
    agregado.dropna(inplace = True)  
  
    return agregado.values
```

# Redes Neuronales

## Predicción con Tensorflow y Keras

```
def prediccion_red_neuronal(historia, n_entrada, n_nodos, n_epocas, tamaño_tandas):  
    datos = series_a_supervisado(historia, n_entrada)  
    entreno_X, entreno_y = datos[:, :-1], datos[:, -1]  
    modelo = Sequential()  
    modelo.add(Dense(n_nodos, activation='relu', input_dim=n_entrada))  
    modelo.add(layers.Dense(1))  
    modelo.compile(loss = "mse", optimizer = "adam")  
    modelo_ajustado = modelo.fit(entreno_X, entreno_y, epochs = n_epocas,  
                                batch_size = tamaño_tandas, verbose = 0)  
    x_entrada = array(historia[-n_entrada:]).reshape(1, n_entrada)  
    yhat = modelo.predict(x_entrada, verbose = 0)  
    return yhat[0]
```

# Redes Neuronales

- Este es un modelo simple que solo tiene capa de entrada y de salida
- Podemos experimentar con:
  - el número de datos que se ven a la vez (eg. 12/24)
  - El número de capas escondidas (eg 50/100/500)
  - El número de épocas (eg 100), y
  - El tamaño de las tandas (eg 100)

# Bonificación



# Prophet

## Predicción de ST - llave en mano

- Ingenieros de Facebook se cansaron de reinventar la rueda
- Desarrollaron su propia herramienta de fuente abierta
- Basado en un modelo aditivo donde tendencias no-lineales se pueden ajustar con estacionalidades anuales, semanales, y diarias
- Tolera bien los datos faltantes, cambios en tendencias y datos atípicos

# Prophet

**Requiere de una pequeña transformación de los datos a usar**

```
series = pd.read_csv("monthly-car-sales.csv", header = 0, index_col = None)
```

```
series["ds"] = pd.to_datetime(series["month"])
```

```
series["y"] = series[["Sales"]].astype(float)
```

```
series = series[["ds", "y"]]
```

```
series.head( )
```

# Prophet

## La ejecución requiere

- Los parámetros más importantes:
  - El número de períodos a predecir
  - La frecuencia de los datos (eg meses)
  - El modo de estacionalidad
  - Pueden agregarse diferentes estacionalidades parar semanas, meses, etc.

# Prophet

## Ejecución

```
modelo = Prophet(mcmc_samples = 500, seasonality_mode = “multiplicative”)
```

```
modelo_ajustado = modelo.fit(series)
```

```
futuro = modelo_ajustado.make_future_dataframe(periods = 48, freq = “M”)
```

```
prediccion = modelo_ajustado.predict(futuro)
```

```
prediccion[["ds", "yhat", "yhat_lower", "yhat_upper"]].tail( )
```