Anomalías IA

Detección de Anomalías con Machine Learning

Juan Miguel Sarria Orozco

1. Construcción de modelo LSTM y detección de anomalías

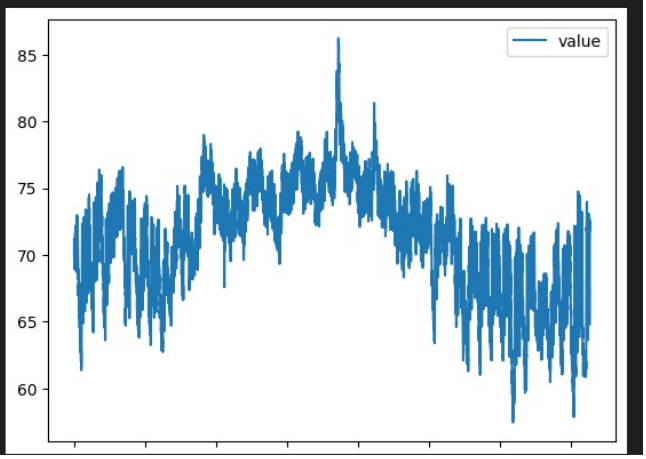
a. Construcción de modelo básico LSTM:

Explicación: Para la construcción del modelo LSTM, los pasos iniciales incluyen la lectura del archivo CSV y la verificación de los datos. A continuación, se muestra el código utilizado para esta etapa y su salida, que incluye una gráfica que representa los datos iniciales:

```
from numpy import array
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from keras.models import Sequential
from keras.layers import LSTM, Dense
from keras.layers import Input
import joblib


# Cargar los datos (7267,1) (según el fichero cambiará el número de filas)
df = pd.read_csv("datos.csv",parse_dates=True)

print(df)
print(df.shape)

df.plot()
plt.show()
```

Análisis: La salida obtenida nos indica que el archivo contiene 7267 filas y 2 columnas (fechas y temperatura). Además, la gráfica nos proporciona una visualización inicial de los datos para analizar tendencias y patrones en los datos originales, así como para identificar posibles anomalías.

```
                timestamp      value
0     2013-07-04 00:00:00  69.880835
1     2013-07-04 01:00:00  71.220227
2     2013-07-04 02:00:00  70.877805
3     2013-07-04 03:00:00  68.959400
4     2013-07-04 04:00:00  69.283551
...                   ...        ...
7262  2014-05-28 11:00:00  72.370206
7263  2014-05-28 12:00:00  72.172956
7264  2014-05-28 13:00:00  72.046565
7265  2014-05-28 14:00:00  71.825226
7266  2014-05-28 15:00:00  72.584089

[7267 rows x 2 columns]
(7267, 2)
```



Explicación: Después, creamos dos DataFrames en Pandas: uno para almacenar los valores y otro para las fechas. Esto nos permite gestionar por separado la información temporal y los valores de temperatura, facilitando la manipulación de los datos.

Además, se define el tamaño de ventana, que en este caso es de 3, con el objetivo de aplicar un enfoque de ventanas deslizantes sobre los valores de temperatura. Este método crea un conjunto de subarrays (ventanas) que contienen cada tres valores consecutivos. Esta estructura es esencial para entrenar el modelo LSTM, ya que permite que el modelo capture patrones secuenciales en los datos.

```python
# Creo mis 2 dataFrames con ambas columnas y se crean las ventanas temporales
columnaDatos = df['value']
columnaFechas= df['timestamp']
nDatosVentana = 3
ventana = np.lib.stride_tricks.sliding_window_view(columnaDatos, window_shape=nDatosVentana)
```

Análisis: Estas ventanas temporales nos permiten estructurar los datos para el entrenamiento de la LSTM, asegurando que cada paso en la secuencia contenga información histórica suficiente para identificar patrones y predecir valores futuros.

```
array([[69.88083514, 71.22022706, 70.87780496],
       [71.22022706, 70.87780496, 68.95939994],
       [70.87780496, 68.95939994, 69.28355102],
       ...,
       [72.37020644, 72.17295622, 72.04656545],
       [72.17295622, 72.04656545, 71.82522648],
       [72.04656545, 71.82522648, 72.58408858]])
```

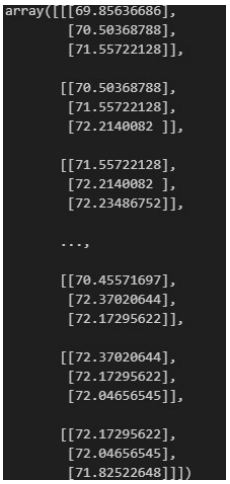
Explicación: En esta sección, se crean dos arrays, X e y, para representar las secuencias de datos de entrenamiento y las etiquetas de predicción, respectivamente. X contiene todas las ventanas de datos menos la última, mientras que y alberga los valores que queremos predecir, a partir del cuarto dato en adelante. Esto permite al modelo LSTM tener datos previos de referencia para intentar predecir el valor siguiente en la secuencia.

Para validar la precisión del modelo, se ha optado por dividir los datos en un conjunto de entrenamiento y otro de prueba, asignando el 70% para entrenamiento y el 30% para prueba. Aunque en ejercicios de series temporales a menudo se usa todo el conjunto para ajustar el modelo, esta división ayuda a evaluar la capacidad de generalización del modelo en datos no vistos.

El siguiente paso implica transformar los datos para ajustarse al formato esperado por la LSTM, que requiere que cada entrada tenga tres dimensiones: número de muestras, pasos de tiempo y número de características.

```
# Dividir los datos en entrenamiento y prueba es lo habitual
X = array(ventana[:-1])  # Todas las ventanas menos la última. Hay que convertir a array numpy
y = array(columnaDatos[nDatosVentana:])  # Lo que predices (el siguiente valor después de cada
ventana)

porcentaje_entrenamiento = 0.7
nEntrenamiento = int(len(X) * porcentaje_entrenamiento)

X_train, X_test = X[:nEntrenamiento], X[nEntrenamiento:]
y_train, y_test = y[:nEntrenamiento], y[nEntrenamiento:]

# Redimensionar los datos para la RNN
# LSTM espera 3 dimensiones: número muestras, pasos temporales, número features
n_features = 1
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], n_features))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], n_features))
```

Análisis: Este enfoque permite que el modelo aprenda patrones de secuencias previas en el conjunto de entrenamiento y se pruebe en datos no observados, evaluando así su capacidad para detectar patrones y anticipar valores en secuencias nuevas. Aunque no queda del todo claro si para este tipo de ejercicio mejora el modelo o no, cosa que veremos en el apartado c con las pruebas para mejorar la red,



Explicación: En esta fase, se construye la red neuronal utilizando una arquitectura LSTM básica, compuesta por una capa LSTM seguida de una capa densa de salida. Esta configuración permite que el modelo aprenda patrones en las secuencias temporales de los datos.

A continuación, se entrena el modelo con el conjunto de entrenamiento (X_train y y_train), utilizando un número de 30 épocas y un tamaño de lote de 32. También se proporciona un conjunto de validación (X_test y y_test) para monitorear el rendimiento del modelo durante el entrenamiento y evitar el sobreajuste.

Una vez finalizado el entrenamiento, se generan predicciones utilizando todo el conjunto de datos de entrada X, aplanando los resultados para facilitar su manipulación posterior.

```
# Crear la RNN
model = Sequential()
model.add(Input(shape=(nDatosVentana, n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

# Entrenar la RNN
model.fit(X_train, y_train, epochs=30, batch_size=32, validation_data=(X_test, y_test))

# Hacer predicciones sobre todo el conjunto
y_pred = model.predict(X)

#Aplano los datos de las predicciones y los transformo en un array de numpy para trabajar mejor
con ellos
y_pred_trimmed = y_pred.flatten()
y_predArray = array(y_pred_trimmed)
```

Análisis: La elección de una arquitectura sencilla, con una sola capa LSTM de 50 unidades, es un enfoque práctico para comenzar el modelado de series temporales. Al entrenar el modelo, el uso del optimizador Adam y la función de pérdida de error cuadrático medio (MSE) son opciones comunes que ayudan a minimizar el error en la predicción. Posteriormente, el aplanado de las predicciones facilita su manipulación en análisis posteriores, como la evaluación de errores y la detección de anomalías,

proporcionando una base sólida para los pasos de evaluación y visualización posteriores en el proceso de detección de anomalías.

```
array([70.70611 , 70.30272 , 69.49637 , ..., 71.731125, 72.09253 ,
       71.92158 ], dtype=float32)
```

b. Detección de anomalías:

Explicación: Para la detección de anomalías, se calcula el error absoluto entre los valores reales y las predicciones del modelo. Este error proporciona una medida cuantitativa de cuán lejos se encuentra cada predicción de su valor real. A partir de esta serie de errores, se calculan la media (MAE) y la desviación estándar, que sirven para establecer un umbral. Este umbral, definido como la media más dos veces la desviación estándar, permite identificar valores que se desvían significativamente del comportamiento normal del modelo.

```
# Calcular el error absoluto para cada predicción
errors = abs(y.flatten() - y_predArray.flatten())

# Calcular la media y la desviación estándar de los errores
mae = np.mean(errors)
std_error = np.std(errors)

# Establecer un umbral para las anomalías
threshold = mae + 2 * std_error  # Por ejemplo, 2 desviaciones estándar

#Imprimo por pantalla los valores
print(f"Error Absoluto Medio (MAE): {mae}")
print(f"Desviacion estandar: {std_error}")
print(f"Umbral: {threshold}")
```

Análisis: Con un error absoluto medio (MAE) de 0.76 y una desviación estándar de 0.61, el umbral para detectar anomalías se establece en aproximadamente 2.00. Este enfoque estadístico permite identificar fácilmente aquellos puntos que se desvían considerablemente de las expectativas del modelo, sugiriendo la posibilidad de anomalías en los datos. A medida que avanzamos en el análisis, en el punto C exploraremos la posibilidad de mejorar estos resultados mediante el ajuste de hiperparámetros, la experimentación con diferentes tamaños de ventana o la implementación de otros enfoques de modelado.

```
Error Absoluto Medio (MAE): 0.7669293437557151
Desviacion estandar: 0.6161344885186921
Umbral: 1.9991983207930994
```

Explicación: A continuación, se crea un array que contiene las fechas y se inicializa un array booleano para identificar las anomalías. En el bucle, se evalúan los errores y se comparan con el umbral previamente establecido. Cuando un error supera este umbral, se considera una anomalía y se registran los detalles correspondientes, incluyendo la fecha, el valor esperado, el valor recibido y la diferencia entre ellos. Esto permite tener una visión clara de las discrepancias en los datos.

```
# Paso a array de numpy el dataTable de fechas para manipularlo mejor y creo un array de
anomalias de booleanos
columnaFechas = array(columnaFechas[nDatosVentana:])
anomalies = []

#Creo mi array con las anomalias e imprimo por pantalla las anomalias y sus diferencias
for index in range(len(y_predArray)):
    if errors[index] > threshold:  # Verificamos si es una anomalia
        expected = y_predArray[index]  # Valor esperado
        received = y[index]     # Valor recibido
        difference = abs(received - expected)  # Diferencia
        print(f"Fecha: {columnaFechas[index]}, Esperado: {expected}, Recibido: {received},
Diferencia: {difference}")
        plt.text(columnaFechas[index], y_predArray[index], f"{errors[index]:.2f}", color='red',
fontsize=8, ha='right')
        anomalies.append(True)
    else:
       anomalies.append(False)

anomalies = np.array(anomalies)

#Resumen de las anomalias
print("El número de anomalías es " + str(np.sum(anomalies)) + " sobre " + str(df.shape[0]))
```

Análisis: Tras la evaluación del modelo, hemos identificado un total de 269 anomalías de un total de 7267 datos. Esto implica que aproximadamente el 3.7% de los datos son considerados anómalos bajo el umbral establecido de aproximadamente 2. Es relevante notar que algunas anomalías presentan diferencias de más de 3 puntos con respecto a los valores esperados, lo que indica variaciones significativas. En el punto C, nos enfocaremos en reducir el número de anomalías detectadas. Sin embargo, es importante equilibrar este objetivo con la precisión del modelo; si bien reducir el número de anomalías es deseable, un aumento en el error podría deteriorar la efectividad de la red. Por lo tanto, primero buscaremos optimizar el modelo para minimizar el error antes de ajustar el umbral de detección de anomalías.

```
Fecha: 2013-07-16 06:00:00, Esperado: 70.11036682128906, Recibido: 67.99007603, Diferencia: 2.120290791289065
Fecha: 2013-07-17 16:00:00, Esperado: 72.24505615234375, Recibido: 74.73026071, Diferencia: 2.485204557656246
Fecha: 2013-07-18 11:00:00, Esperado: 70.08805084228516, Recibido: 72.41314268, Diferencia: 2.325091837714851
Fecha: 2013-07-18 14:00:00, Esperado: 71.70443725585938, Recibido: 73.89859399, Diferencia: 2.19415673414062
Fecha: 2013-07-18 18:00:00, Esperado: 74.10547637939453, Recibido: 76.39001911, Diferencia: 2.284542730605466
...
Fecha: 2014-05-28 09:00:00, Esperado: 66.02896118164062, Recibido: 68.03307954, Diferencia: 2.004118358359378
Fecha: 2014-05-28 10:00:00, Esperado: 67.185791015625, Recibido: 70.45571697, Diferencia: 3.2699259543749974
Fecha: 2014-05-28 11:00:00, Esperado: 68.6278076171875, Recibido: 72.37020644, Diferencia: 3.742398822812504
El número de anomalías es 269 sobre 7267
```

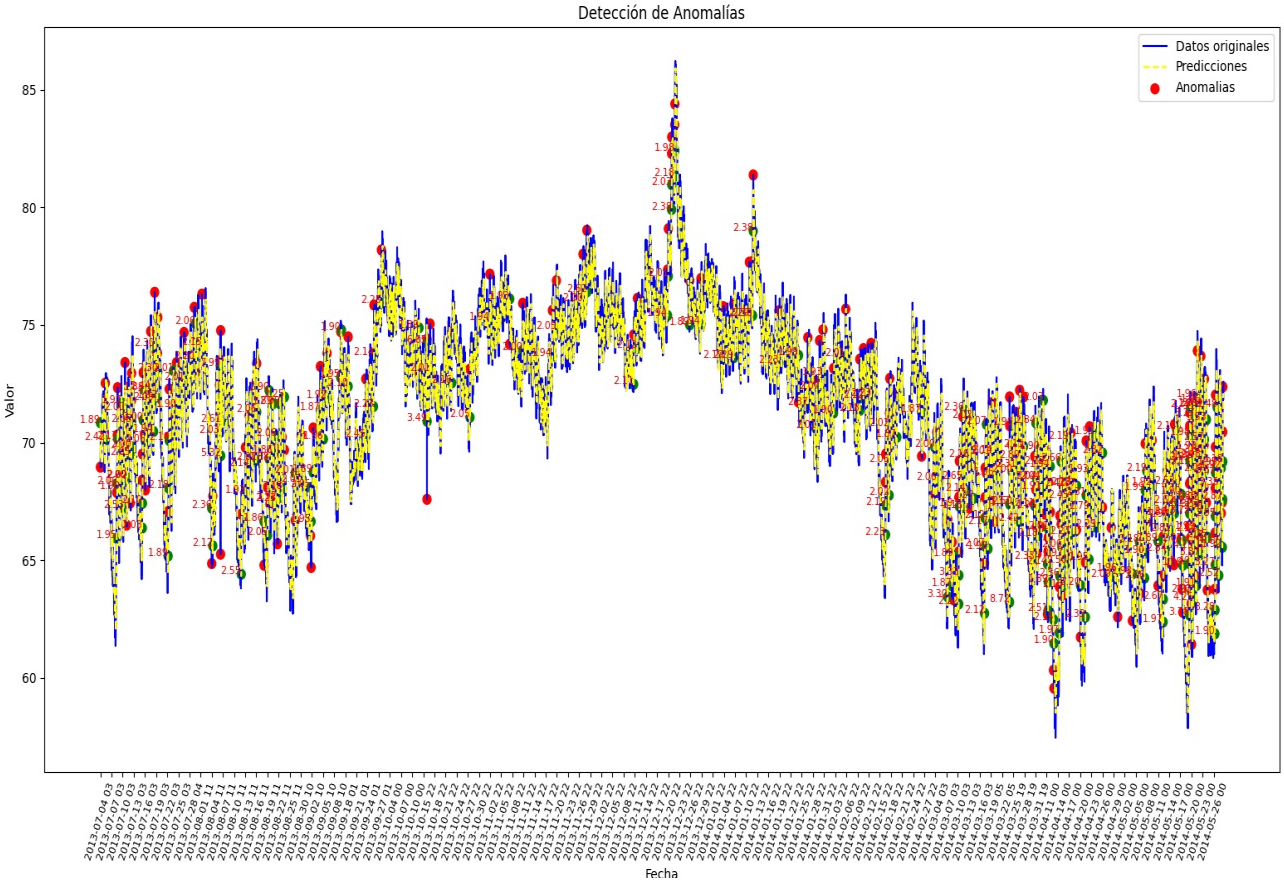
Explicación: Finalmente, empleamos las funciones de Matplotlib para visualizar los resultados del modelo. En la gráfica, utilizamos una línea azul para representar los datos reales, una línea amarilla discontinua para las predicciones, y puntos rojos para señalar las anomalías detectadas en los datos reales. Además, agregamos puntos verdes para las anomalías en las predicciones, y se incluyen etiquetas en rojo que indican la diferencia entre los valores esperados y los reales en cada anomalía. Para mejorar la legibilidad de la gráfica, dividimos las fechas en intervalos de tres días, evitando así la congestión visual.

```
#Todas las funciones de matplotlab para mostrar la grafica de la manera que quiero hacera
#Muestro una linea azul con los valores reales, otras linea amarilla con las predicciones, y
puntos rojos con las anomalias en los valores reales
#Y puntos verdes con las anomalias en las predicciones y el valor de la diferencia en los puntos
rojos, ademas
#Divido las fechas cada 3 dias para no colapsar la grafica
plt.plot(columnaFechas, y, color='blue',label='Datos originales')
plt.plot(columnaFechas, y_predArray, color='yellow', linestyle='--', label='Predicciones')   #
Añadir y_pred en color amarillo
plt.scatter(x=columnaFechas, y=y, c='red', alpha=anomalies.astype(int),s=50, label='Anomalias')
plt.scatter(x=columnaFechas, y=y_predArray, c='green', alpha=anomalies.astype(int),s=50)
plt.xlabel("Fecha")
plt.ylabel("Valor")
plt.title("Detección de Anomalías")

columnaFechas = pd.to_datetime(columnaFechas)
tick_positions = np.arange(0, len(columnaFechas), 72)  # Crear intervalos cada 72 puntos de
datos
tick_labels = [date.strftime("%Y-%m-%d %H") for date in columnaFechas[tick_positions]]  #
Formatear las fechas

plt.xticks(tick_positions, tick_labels, rotation=70, fontsize=8)

plt.legend()
plt.show()
```

Análisis: La visualización gráfica proporciona una representación clara de la relación entre los datos reales y las predicciones del modelo. La línea azul representa de manera efectiva la serie temporal de los datos originales, mientras que la línea amarilla discontinua muestra cómo las predicciones del modelo se alinean con estos datos. Las anomalías son identificables con puntos rojos que destacan en el gráfico, y su superposición con los puntos verdes en las predicciones resalta las discrepancias importantes en los valores estimados por el modelo.

La inclusión de etiquetas que indican la diferencia entre los valores esperados y reales en las anomalías añade un valor significativo a la visualización, permitiendo una rápida identificación de las discrepancias más pronunciadas. Esta representación visual es crucial para evaluar el rendimiento del modelo, ya que permite observar tanto la precisión de las predicciones como la efectividad del proceso de detección de anomalías.

En resumen, este análisis y la visualización resultante no solo validan la efectividad del modelo LSTM en la detección de anomalías, sino que también proporcionan una base sólida para futuras mejoras, ya que podemos identificar claramente áreas donde el modelo podría ser ajustado o entrenado de manera diferente para mejorar la precisión y reducir el número de anomalías detectadas.


Detección de Anomalías
Datos originales
Predicciones
Anomalias
Valor
Fecha

c. Mejora del modelo:

Epochs = 10 y 100

Explicación: En este experimento, se modifica el número de epochs durante el entrenamiento del modelo LSTM. Inicialmente, se establecieron en 30, y ahora se evalúan dos configuraciones diferentes: 10 y 100 epochs. Esta variación nos permitirá observar cómo afecta el número de iteraciones al rendimiento del modelo.

```
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))
Error Absoluto Medio (MAE): 0.798668544336115
Desviacion estandar: 0.6507286049780729
Umbral: 2.100125754292261

model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test))
Error Absoluto Medio (MAE): 0.7163083988601631
Desviacion estandar: 0.5575118384906699
Umbral: 1.8313320758415028
```

Análisis: Los resultados obtenidos muestran que el rendimiento del modelo tiende a mejorar a medida que se incrementa el número de epochs. Esto es esperado, ya que un mayor número de epochs permite que el modelo ajuste sus parámetros de manera más precisa a los datos de entrenamiento. Sin embargo, hay que tener en cuenta el riesgo de sobreajuste (overfitting), que ocurre cuando el modelo se adapta demasiado a los datos de entrenamiento y pierde capacidad de generalización a nuevos datos.

Dado que el entrenamiento con 100 epochs parece proporcionar un buen equilibrio entre ajuste y generalización, se decide mantener esta configuración por el momento.

Bach_Size = 16 y 256

Explicación: En esta fase del experimento, se modifica el tamaño del batch, probando configuraciones de 16 y 256. El tamaño del batch determina cuántas muestras se utilizan para actualizar los pesos del modelo en cada iteración. Un tamaño de batch pequeño puede hacer que el modelo se actualice con mayor frecuencia, mientras que un tamaño de batch grande puede proporcionar una estimación más estable del gradiente.

```
model.fit(X_train, y_train, epochs=100, batch_size=16, validation_data=(X_test, y_test))
Error Absoluto Medio (MAE): 0.7150867835432817
Desviacion estandar: 0.5520750612164396
Umbral: 1.8192369059761608

model.fit(X_train, y_train, epochs=100, batch_size=256, validation_data=(X_test, y_test))
Error Absoluto Medio (MAE): 0.7366003204916476
Desviacion estandar: 0.5805658246832226
Umbral: 1.8977319698580928
```

Análisis: Tras realizar estas pruebas, se observa que no ha habido cambios

significativos en el rendimiento del modelo. Esto sugiere que el tamaño del batch no tiene un impacto considerable en la capacidad de aprendizaje del modelo en este caso particular.

Sin datos de entrenamiento

Explicación: En esta fase, se decide entrenar la red neuronal utilizando el 100% del conjunto de datos disponible, eliminando la división entre los conjuntos de entrenamiento y prueba. Esto implica que todos los datos se utilizan para ajustar los parámetros del modelo, lo que puede ser beneficioso en situaciones donde el conjunto de datos es limitado o donde se espera que el modelo se generalice bien.

```
porcentaje_entrenamiento = 1
nEntrenamiento = int(len(X) * porcentaje_entrenamiento)

X_train, X_test = X[:nEntrenamiento], X[nEntrenamiento:]
                  y_train, y_test = y[:nEntrenamiento], y[nEntrenamiento:]

                  Error Absoluto Medio (MAE): 0.7287521417336164
                  Desviacion estandar: 0.5673979645387099
                  Umbral: 1.8635480708110364
```

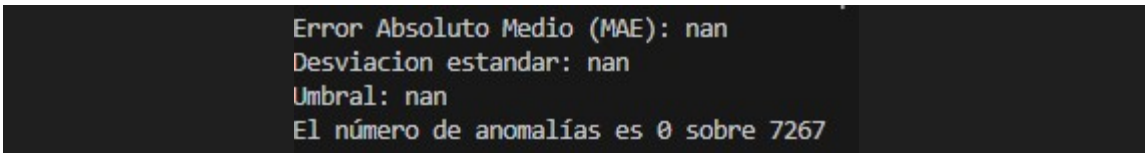
Análisis: Al realizar esta modificación, se observa que no ha habido variación en el rendimiento del modelo en comparación con los intentos anteriores que incluían la separación de los datos. Esto sugiere que, en este caso específico, entrenar el modelo con todo el conjunto de datos no afecta negativamente su capacidad de generalización.

Dado que la red no se está evaluando en un conjunto de datos separado, eliminamos la necesidad de tener un conjunto de prueba, esta estrategia puede ser viable cuando se cuenta con un conjunto de datos limitado y se busca maximizar el uso de la información disponible.

Tamaño de ventana 2 vs 30

Explicación: En esta etapa, se experimenta con diferentes tamaños de ventana para observar cómo afectan al rendimiento del modelo de detección de anomalías. Se prueban dos configuraciones: una ventana pequeña de tamaño 2 y una ventana grande de tamaño 30..

```
nDatosVentana = 2

                  Error Absoluto Medio (MAE): 0.7067645714695927
                  Desviacion estandar: 0.5425505900775593
                  Umbral: 1.7918657516247114
nDatosVentana = 30
```



Análisis: La elección del tamaño de ventana es crucial en la modelación de series temporales y detección de anomalías. Un tamaño de ventana demasiado pequeño puede llevar a predicciones poco informadas, mientras que uno demasiado grande puede causar sobreajuste. Por lo tanto, un tamaño de ventana de 5 se establece como una buena opción, ya que proporciona suficiente contexto temporal para que el modelo aprenda de manera efectiva sin perder su capacidad de generalización.

Añadir mas neuronas a mi capa 100 vs 300

Explicación: Aunque añadir mas neuronas relentizara el entrenamiento, al disponer de un buen PC con una buena GPU vamos a hacer una prueba con 100 y 300 neuronas para ver la diferencia.

```
model.add(LSTM(100, activation='relu'))
Error Absoluto Medio (MAE): 0.6960193689990007
Desviacion estandar: 0.536160063108954
Umbral: 1.768339495216909
model.add(LSTM(300, activation='relu'))
Error Absoluto Medio (MAE): 0.7365740241567473
Desviacion estandar: 0.5591575672575749
Umbral: 1.8548891586718972
```

Análisis: Después de realizar estas pruebas, se concluye que no es beneficioso aumentar el número de neuronas en la capa LSTM a 300, dado que no proporciona mejoras en el rendimiento y ralentiza el entrenamiento. Por lo tanto, se decide mantener la configuración con 100 neuronas, que parece ser más eficiente tanto en términos de velocidad de entrenamiento como de capacidad de generalización del modelo. Esto destaca la importancia de encontrar un equilibrio entre la complejidad del modelo y la eficiencia en el entrenamiento.

Activacion tanh y sigmoid

Explicación: En esta fase, se experimenta con diferentes funciones de activación en la capa LSTM para observar su efecto en el rendimiento del modelo. Se prueban las activaciones tanh y sigmoid

```
model.add(LSTM(100, activation='tanh'))
Error Absoluto Medio (MAE): 0.7015987747987397
Desviacion estandar: 0.5401199375372444
Umbral: 1.7818386498732286
model.add(LSTM(100, activation='sigmoid'))
Error Absoluto Medio (MAE): 0.7206316915926319
Desviacion estandar: 0.5572022502937846
Umbral: 1.835036192180201
```

Análisis: Tras experimentar con las funciones de activación tanh y sigmoid, se concluye que el cambio en las activaciones no tiene un impacto significativo en el rendimiento del modelo. Esto resalta la robustez del modelo y su capacidad para aprender de los datos, independientemente de la función de activación. Por lo tanto, se puede continuar con la configuración actual sin realizar modificaciones adicionales en este aspecto

Añadir mas capas y Dropout tanh y sigmoid

Explicación: En esta fase, se busca aumentar la complejidad del modelo añadiendo más capas LSTM y utilizando técnicas de regularización como Dropout y Batch Normalization para mejorar el rendimiento general y evitar el sobreajuste. Se hace la siguiente configuración:

```
model = Sequential()
model.add(Input(shape=(nDatosVentana, n_features)))
model.add(LSTM(50, activation='tanh', return_sequences=True))  # Cambiar a tanh
model.add(Dropout(0.2))  # Añadir Dropout
model.add(BatchNormalization())  # Añadir Batch Normalization
model.add(LSTM(50, activation='tanh'))  # Otra capa LSTM
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

```
Error Absoluto Medio (MAE): 0.7219134732603565
Desviacion estandar: 0.5449297651701054
Umbral: 1.8117730036005673
```

Analisis: A pesar de los esfuerzos por mejorar el modelo mediante la adición de más capas y técnicas de regularización, los resultados no justifican el aumento en la complejidad y el tiempo de entrenamiento. Se observa que estas modificaciones, en lugar de mejorar el rendimiento, pueden llevar a un sobreajuste y ralentizar el proceso de entrenamiento. Por lo tanto, es recomendable mantener un enfoque más simple, posiblemente volviendo a una arquitectura más básica que ofrezca un rendimiento sólido sin la necesidad de una mayor complejidad

Cambio en el Umbral para la Detección de Anomalías

Explicación: Para ajustar la sensibilidad del modelo en la detección de anomalías, se modifica la forma en que se calcula el umbral. El nuevo umbral se define de la siguiente manera

```
threshold = mae + 2*std_error
threshold = 2*mae + std_error
```

Analisis: Al modificar el umbral de detección de anomalías, se puede mejorar la precisión en la identificación de datos atípicos, especialmente en conjuntos de datos donde las fluctuaciones menores no son significativas o relevantes. Este cambio puede ayudar a afinar el modelo para que se ajuste mejor a los objetivos específicos del análisis, aunque es fundamental validar el rendimiento del modelo después de realizar estos cambios para asegurarse de que no se pase por alto información importante.

Escalar los datos con MinMaxScaler

Explicación: Se utiliza MinMaxScaler para transformar los datos a un rango específico, generalmente entre 0 y 1. Este escalado permite que los datos estén en una escala común, lo que puede ser crucial para el rendimiento de muchos algoritmos de aprendizaje automático, especialmente aquellos que son sensibles a la escala de los datos, como las redes neuronales.

```
# Configurar MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
columnaDatos = df['value'].values.reshape(-1, 1)
columnaDatos = scaler.fit_transform(columnaDatos)  # Escalar valores de entrada

--

y_pred_inverted = scaler.inverse_transform(y_pred).flatten()
y_inverted = scaler.inverse_transform(y).flatten()
```

```
Error Absoluto Medio (MAE): 0.6872492418899335
Desviacion estandar: 0.5305140093945948
Umbral: 1.9050124931744619
```

Análisis: Tras implementar todos estos cambios, incluyendo el escalado de datos, se puede observar cómo se ha disminuido la tasa de errores por tanto como ha mejorado la detección de anomalías

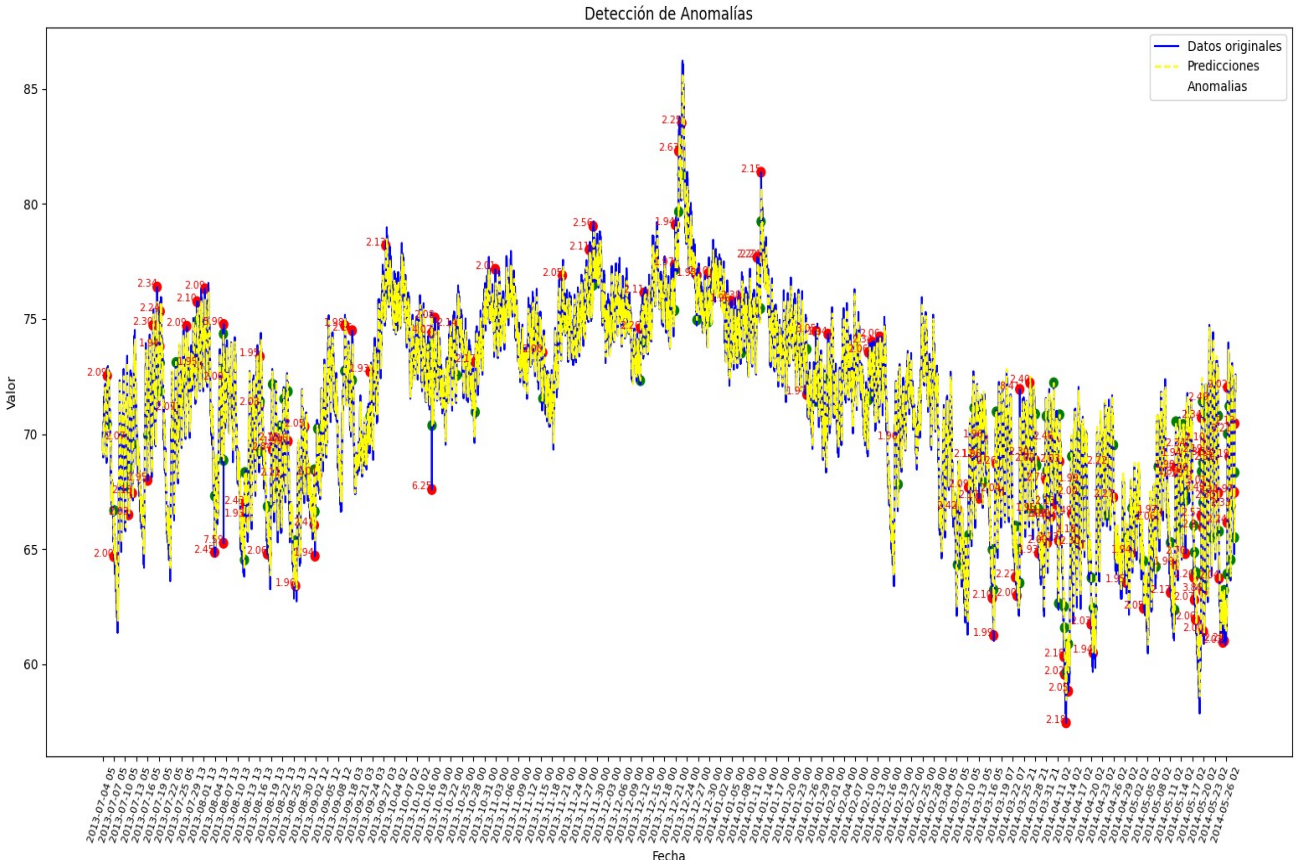
Conclusion:

Despues de todos estos cambios los errores y anomalias han quedado de la siguiente forma:

```
Error Absoluto Medio (MAE): 0.6872492418899335
Desviacion estandar: 0.5305140093945948
Umbral: 1.9050124931744619
```

```
El número de anomalías es 144 sobre 7267
```

Aunque si se ha podido mejorar algo la red y afinarla un poco mas, las mejoras han sido minimas, incluso empeorando los tiempos de entrenamiento, los datos son muy parecidos al entrenamiento estandar del comienzo y aunque se han realizado numerosas modificaciones, el modelo parece haber alcanzado un nivel de rendimiento que no se puede mejorar significativamente.



2. Autoencoder

Explicación: Los pasos iniciales pata el autoencoder son similares a los del entrenamiento utilizando la red neuronal LSTM, volvemos a crear la ventana para permitir que el autoencoder aproveche mejor la información temporal y haga predicciones más precisas y contextualizadas, mejorando así la detección de patrones inusuales o anómalos y guardamos un array redimensionandolo a 3 dimensiones para que encaje en nuestra red LSTM, ya que vamos a simular un Autoencoder mediante una red LSTM

```
from numpy import array
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import LSTM, Dense, RepeatVector, TimeDistributed, Input

# Cargar los datos
df = pd.read_csv("datos.csv", parse_dates=True)

# Crear los dataFrames con ambas columnas y ventanas temporales
columnaDatos = df['value']
columnaFechas = df['timestamp']
nDatosVentana = 5
ventana = np.lib.stride_tricks.sliding_window_view(columnaDatos, window_shape=nDatosVentana)

# Convertimos en array numpy
X = array(ventana)

# Redimensionar los datos para el autoencoder
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Análisis: La creación de ventanas temporales permite al modelo capturar patrones temporales y contextuales. Al redimensionar los datos a tres dimensiones, se asegura que la red pueda procesar adecuadamente la secuencia. La estructura del autoencoder LSTM es adecuada para detectar anomalías, ya que intenta aprender y reconstruir las características normales de los datos, permitiendo identificar desviaciones en la reconstrucción como anomalías

```
       [[70.87780496],
        [68.95939994],
        [69.28355102],
        [70.06096581],
        [69.27976479]],

       ...,

       [[68.03307954],
        [70.45571697],
        [72.37020644],
        [72.17295622],
        [72.04656545]],
...
       [[72.37020644],
        [72.17295622],
        [72.04656545],
        [71.82522648],
        [72.58408858]]])
```

Explicación: Después, creamos el autoencoder emulandolo utilizando las redes neuronales LSTM. Entrenamos el modelo y generamos predicciones utilizando todo el conjunto X, El autoencoder se entrena con el conjunto de datos X para que aprenda a reconstruir sus entradas.

```
# Crear el autoencoder LSTM
model = Sequential()
model.add(Input(shape=(nDatosVentana, n_features)))
model.add(LSTM(100, activation='relu'))
model.add(RepeatVector(nDatosVentana))
model.add(LSTM(100, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mse')
model.summary()

# Entrenar el autoencoder
model.fit(X, X, epochs=100, batch_size=32)

# Hacer predicciones sobre el conjunto de datos
y_pred = model.predict(X)
```

Análisis: La estructura del autoencoder permite que el modelo aprenda patrones complejos y temporales en los datos, lo que es esencial para la detección de anomalías. Al entrenar el modelo para reconstruir sus propias entradas, se espera que las anomalías se reflejen en una reconstrucción deficiente, lo que facilitará su identificación. Un número mayor de épocas (100 en este caso) busca mejorar la convergencia del modelo, pero hay que vigilar el sobreajuste. La utilización de LSTM es adecuada debido a su capacidad para manejar secuencias temporales, lo que incrementa la efectividad del autoencoder en contextos de series temporales.

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 100)	40,800
repeat_vector (RepeatVector)	(None, 5, 100)	0
lstm_1 (LSTM)	(None, 5, 100)	80,400
time_distributed (TimeDistributed)	(None, 5, 1)	101

Explicación: Para la detección de anomalías, volvemos a calcular el error absoluto y la media de la desviación estándar para establecer un umbral, similar al ejercicio 1.

```
# Calcular el error absoluto en cada predicción para identificar anomalías
errors = np.mean(abs(X - y_pred), axis=1)  # Error promedio por ventana

# Calcular el umbral basado en el error promedio y desviación estándar
mae = np.mean(errors)
std_error = np.std(errors)
threshold = 2 * mae + std_error

print(f"Error Absoluto Medio (MAE): {mae}")
print(f"Desviacion estandar: {std_error}")
print(f"Umbral: {threshold}")
```

Análisis: Con un error absoluto medio (MAE) de 0.05 y una desviación estándar de 0.03, se puede inferir que el autoencoder se adapta mejor a los datos que una red neuronal tradicional. Esto sugiere que el autoencoder es eficaz para capturar las características de la serie temporal y reconstruirlas con alta precisión. Como resultado, la detección de anomalías puede ser más confiable, ya que el modelo es capaz de identificar patrones normales en los datos y resaltar desviaciones significativas. Esto valida la elección del autoencoder como herramienta para la detección de anomalías en este contexto.

```
Error Absoluto Medio (MAE): 0.054036676284412694
Desviacion estandar: 0.02998575669993063
Umbral: 0.13805910926875603
```

Explicación: En este paso, se crea un array de fechas a partir de la columna de timestamps y se inicializa una lista para registrar las anomalías detectadas. Se itera sobre las predicciones del autoencoder y se evalúa si el error supera el umbral establecido; si es así, se registra como una anomalía. Para cada anomalía, se imprimen la fecha, el dato esperado, el dato recibido y la diferencia entre ambos. Como el array de predicciones tiene tres dimensiones, se accede al último valor de la predicción y del dato original en la ventana de forma adecuada. Al final, se convierte la lista de anomalías en un array de NumPy y se imprime el total de anomalías detectadas en relación al total de datos analizados.

```
# Convertir fechas a array de numpy
columnaFechas = array(columnaFechas[nDatosVentana - 1:])
anomalies = []

# Detectar e imprimir anomalías
for index in range(len(y_pred)):
    error = errors[index]
    if error > threshold:
        expected = y_pred[index, -1, 0]  # Último valor de la predicción para la ventana
        received = X[index, -1, 0]       # Último valor de los datos originales en la ventana
        difference = abs(received - expected)
        print(f"Fecha: {columnaFechas[index]}, Esperado: {expected}, Recibido: {received},
Diferencia: {difference}")
        anomalies.append(True)
    else:
        anomalies.append(False)

anomalies = np.array(anomalies)

# Resumen de las anomalías
print("El número de anomalías es " + str(np.sum(anomalies)) + " sobre " + str(df.shape[0]))
```

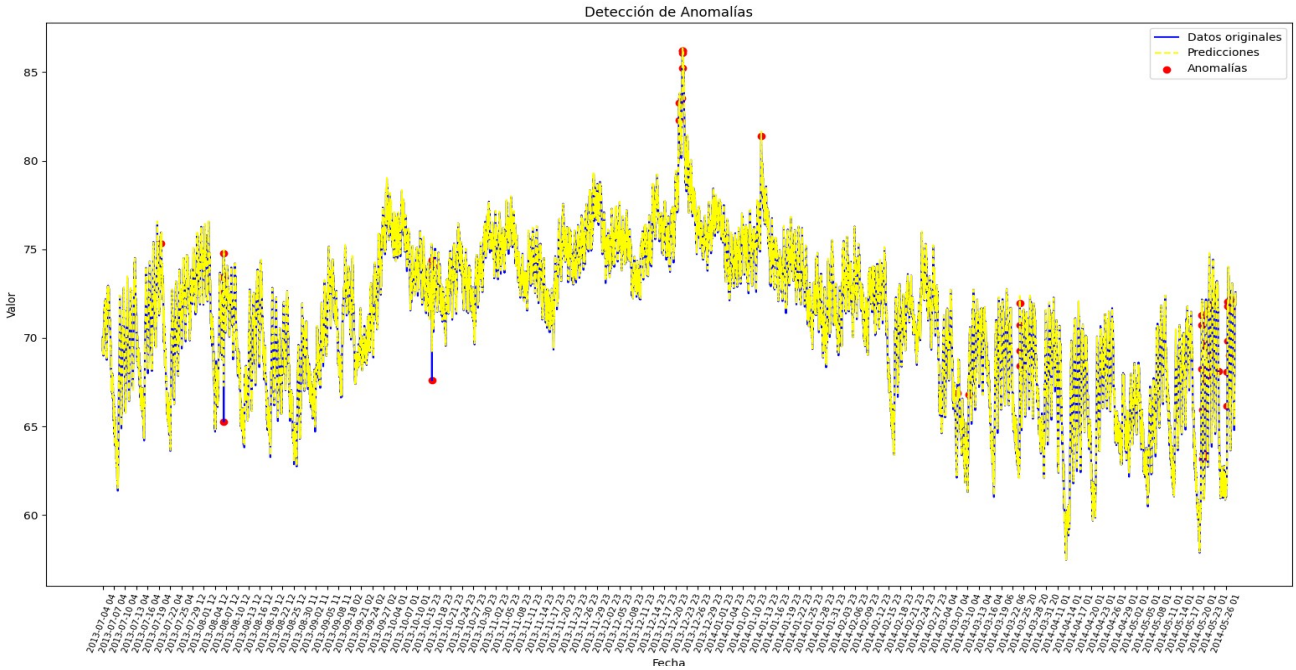
Análisis: Después del entrenamiento, se identifican 63 anomalías entre 7267 datos, lo que sugiere que el autoencoder ha mejorado la precisión en la detección de anomalías en comparación con las redes LSTM anteriores. El umbral calculado de aproximadamente 0.13 respalda esta afirmación, ya que permite una identificación más efectiva de los patrones inusuales en los datos.

```
Fecha: 2014-03-28 10:00:00, Esperado: 68.24600219726562, Recibido: 68.54764706, Diferencia: 0.30164486273437774
Fecha: 2014-03-31 10:00:00, Esperado: 65.43736267089844, Recibido: 65.76317725, Diferencia: 0.32581457910156075
Fecha: 2014-04-11 10:00:00, Esperado: 65.67864990234375, Recibido: 65.96957647, Diferencia: 0.2909265676562569
Fecha: 2014-04-11 11:00:00, Esperado: 66.31224822998047, Recibido: 66.59359265, Diferencia: 0.2813444200195363
Fecha: 2014-04-11 12:00:00, Esperado: 68.10161590576172, Recibido: 68.33141592, Diferencia: 0.229800014238279
...
Fecha: 2014-05-27 12:00:00, Esperado: 71.94558715820312, Recibido: 72.17782106, Diferencia: 0.23223390179687442
Fecha: 2014-05-27 13:00:00, Esperado: 72.53894805908203, Recibido: 72.68078037, Diferencia: 0.14183231091796245
Fecha: 2014-05-28 13:00:00, Esperado: 71.87395477294922, Recibido: 72.04656545, Diferencia: 0.17261067705078403
El número de anomalías es 63 sobre 7267
```

Explicación: En este último paso, se utiliza Matplotlib para visualizar los resultados del análisis de anomalías. Se grafican los datos originales en azul y las predicciones generadas por el autoencoder en amarillo, utilizando una línea discontinua. Las anomalías detectadas en los datos reales se marcan con puntos rojos, mientras que las anomalías en las predicciones se indican con puntos verdes. Además, se incluye un valor flotante en rojo que muestra la diferencia entre el dato real y la predicción en las anomalías, facilitando así la interpretación de los resultados.

```
# Visualización de resultados
plt.plot(columnaFechas, columnaDatos[nDatosVentana - 1:], color='blue', label='Datos
originales')
plt.plot(columnaFechas, y_pred[:, -1, 0], color='yellow', linestyle='--',
label='Predicciones')  # Predicción en amarillo
plt.scatter(columnaFechas[anomalies], columnaDatos[nDatosVentana - 1:][anomalies], color='red',
label='Anomalías')
plt.xlabel("Fecha")
plt.ylabel("Valor")
plt.title("Detección de Anomalías")

# Configurar ticks en el eje X cada 3 días para claridad
columnaFechas = pd.to_datetime(columnaFechas)
tick_positions = np.arange(0, len(columnaFechas), 72)
tick_labels = [date.strftime("%Y-%m-%d %H") for date in columnaFechas[tick_positions]]

plt.xticks(tick_positions, tick_labels, rotation=70, fontsize=8)
plt.legend()
plt.show()
```

Análisis: La visualización ofrece una representación clara de la efectividad del autoencoder en la identificación de anomalías en el conjunto de datos. El modelo ha detectado un número significativamente menor de anomalías, lo que sugiere una mayor precisión en su capacidad de detección. Este enfoque resulta especialmente valioso, ya que las anomalías detectadas son las más relevantes y significativas, dado que el umbral establecido para su identificación es relativamente bajo (0,13).


Detección de Anomalías
Datos originales
Predicciones
Anomalías
Valor
Fecha

3. Isolation forest

Explicación: Los pasos iniciales para utilizar Isolation Forest son similares a los utilizados en el entrenamiento con redes neuronales y autoencoders; sin embargo, una diferencia clave es que no es necesario crear ventanas temporales. Isolation Forest funciona de manera diferente al clasificar directamente cada punto de datos como normal (1) o anómalo (-1), basado en su posición en relación con la distribución de los datos. En este enfoque, la entrada es un array bidimensional, que permite al modelo evaluar cada punto individualmente en lugar de predecir un valor futuro como en otros métodos.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest

# Cargar los datos
df = pd.read_csv("datos.csv", parse_dates=True)
columnaDatos = df['value']
columnaFechas = df['timestamp']

# Convertir a array y reshape para Isolation Forest
X = columnaDatos.values.reshape(-1, 1)
```

Análisis: Los Isolation Forest tienen la capacidad para detectar anomalías sin la necesidad de crear estructuras temporales complejas, lo que simplifica el proceso de preparación de datos. Este modelo es eficiente en identificar puntos que se desvían significativamente de la norma al basarse en la idea de que las anomalías son más fáciles de "aislar" en comparación con los puntos normales. Al trabajar con un array bidimensional, se facilita el manejo de los datos, y el modelo puede ser más rápido y directo en su clasificación, lo que resulta en una detección de anomalías efectiva y sin complicaciones innecesarias.

```
array([[69.88083514],
       [71.22022706],
       [70.87780496],
       ...,
       [72.04656545],
       [71.82522648],
       [72.58408858]])
```

Explicación: En esta etapa, configuramos y entrenamos el modelo Isolation Forest estableciendo una tasa de contaminación del 0,01. Esto indica que anticipamos que aproximadamente el 1% de los datos serán considerados anomalías. Además, para asegurar la reproducibilidad de los resultados en cada ejecución, se define una semilla para la generación de números aleatorios utilizando random_state=0. Con estos parámetros establecidos, procedemos a entrenar el modelo utilizando el conjunto de datos y luego generamos predicciones sobre el mismo, donde cada punto se clasifica como normal (1) o anómalo (-1).

```
# Configurar y entrenar Isolation Forest
clf = IsolationForest(contamination=0.01, random_state=0)
clf.fit(X)

# Predecir anomalías
y_pred = clf.predict(X)
```

Análisis: El establecimiento de una tasa de contaminación del 0,01 es una estrategia importante, ya que permite al modelo identificar anomalías basándose en expectativas realistas sobre la naturaleza de los datos. Este enfoque es especialmente útil en conjuntos de datos donde las anomalías son raras y se desea evitar la identificación errónea de valores normales. Al utilizar random_state, se garantiza que los resultados sean consistentes y replicables, lo que es crucial para validar el rendimiento del modelo en diferentes ejecuciones. En general, la configuración inicial del modelo es fundamental para su eficacia, y esta aproximación permite una detección de anomalías más precisa y confiable.

```
array([1, 1, 1, ..., 1, 1, 1])
```

Explicación: En esta etapa, convertimos las predicciones del modelo en un array de booleanos que identifica las anomalías. Asignamos el valor True a los puntos que han sido clasificados como anómalos (predicción de -1) y False a los normales (predicción de 1). Luego, recorremos el array de anomalías y, para cada anomalía detectada, imprimimos la fecha y el valor correspondiente. Al final, proporcionamos un resumen que indica el número total de anomalías detectadas en relación con el total de datos en el conjunto.

```
# Convertir a booleanos para anomalías (1: normal, -1: anómalo)
anomalies = y_pred == -1

# Imprimir anomalías y diferencias de valores
for index, is_anomaly in enumerate(anomalies):
    if is_anomaly:
        fecha = columnaFechas[index]
        valor = columnaDatos[index]
        print(f"Fecha: {fecha}, Valor: {valor} - Anomalía detectada")

# Resumen de las anomalías
print(f"El número de anomalías es {np.sum(anomalies)} sobre {df.shape[0]}")
```

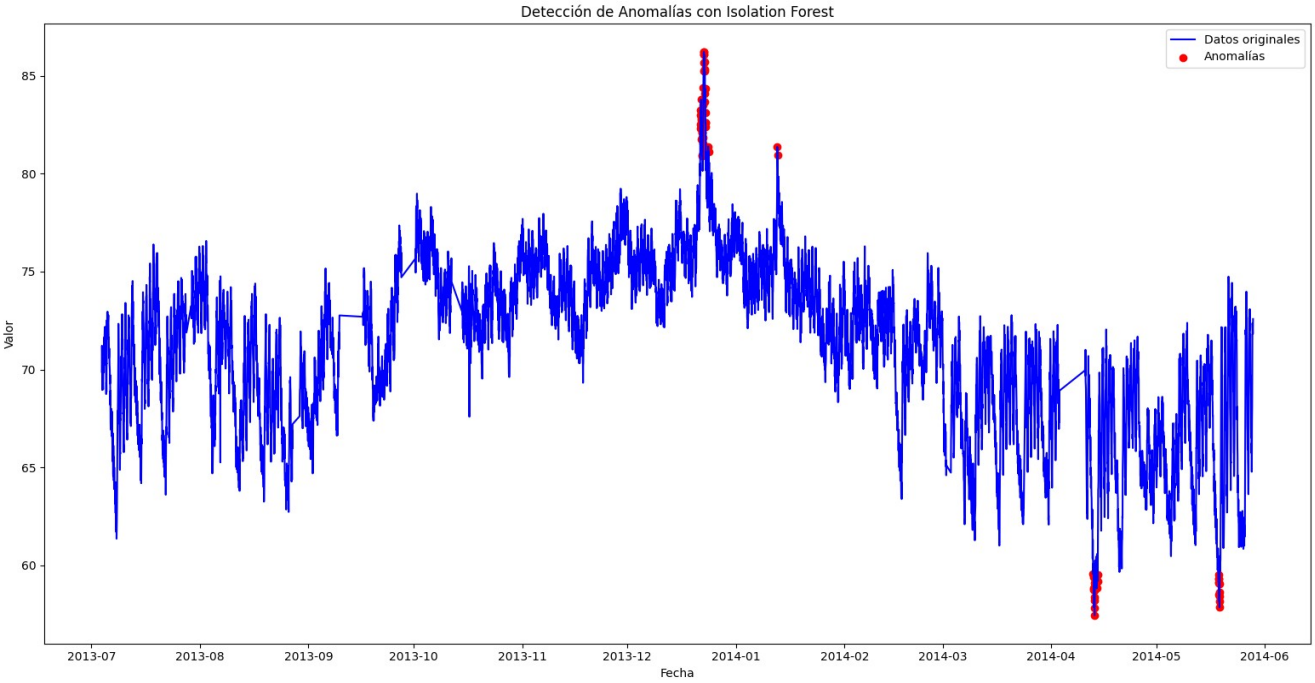
Análisis: Al aplicar el modelo Isolation Forest, observamos que el número de anomalías detectadas es notablemente inferior en comparación con otros métodos, lo que puede ser una indicación de que el modelo está funcionando correctamente al filtrar valores atípicos. La elección de una tasa de contaminación del 0,01 es clave, ya que establece expectativas claras sobre la proporción de datos que se considerarán anómalos. Este porcentaje coincide aproximadamente con el número de anomalías detectadas, lo que sugiere que el modelo está alineado con la naturaleza del conjunto de datos y su distribución. Sin embargo, es importante considerar que este enfoque puede ser menos sensible a anomalías más sutiles, tan solo mostrandonos los extremos.

```
Fecha: 2013-12-22 17:00:00, Valor: 84.39093203 - Anomalía detectada
Fecha: 2013-12-22 18:00:00, Valor: 85.22768546 - Anomalía detectada
Fecha: 2013-12-22 19:00:00, Valor: 86.09488844 - Anomalía detectada
Fecha: 2013-12-22 20:00:00, Valor: 86.20418922 - Anomalía detectada
Fecha: 2013-12-22 21:00:00, Valor: 86.22321261 - Anomalía detectada
Fecha: 2013-12-22 22:00:00, Valor: 85.64943737 - Anomalía detectada
...
Fecha: 2014-05-19 01:00:00, Valor: 57.8619057 - Anomalía detectada
Fecha: 2014-05-19 02:00:00, Valor: 58.63929497 - Anomalía detectada
Fecha: 2014-05-19 03:00:00, Valor: 59.07469099 - Anomalía detectada
El número de anomalías es 73 sobre 7267
```

Explicación: Por ultimo visualizamos el modelo graficamente con las librerias de matplotlab y efectivamente nos da como anomalias los picos mas altos y bajos de la grafica.

```
# Visualización
columnaFechas = pd.to_datetime(columnaFechas)
plt.plot(columnaFechas, columnaDatos, color='blue', label='Datos originales')
plt.scatter(columnaFechas[anomalies], columnaDatos[anomalies], color='red', label='Anomalías')
plt.xlabel("Fecha")
plt.ylabel("Valor")
plt.title("Detección de Anomalías con Isolation Forest")
plt.legend()
plt.show()
```

Análisis: La visualización del modelo revela que el Isolation Forest ha identificado principalmente los picos más altos y bajos en la serie de datos como anomalías. Este comportamiento se debe a la naturaleza del algoritmo, que se centra en detectar puntos que se desvían significativamente de la distribución general de los datos. Dado que estos picos representan valores extremos en comparación con el resto del conjunto de datos, el modelo los clasifica correctamente como anómalos. Sin embargo, es importante señalar que este enfoque puede pasar por alto anomalías más sutiles que no alcanzan los extremos de la distribución.



Conclusiones finales:

1. LSTM:
    - **Error Absoluto Medio (MAE):** 0.71
    - **Desviación Estándar:** 0.54
    - **Umbral:** 1.96
    - **Número de Anomalías:** 144
    - **Observaciones:** Este modelo parece identificar una gran cantidad de anomalías (144), lo cual podría indicar que está siendo muy sensible. La media y la desviación estándar son relativamente altas, lo que sugiere que hay una variabilidad significativa en los errores de predicción. Puede detectar anomalías en patrones de tiempo más complejos pero a su vez podría dar falsos positivos.

2. Autoencoder:
    - **Error Absoluto Medio (MAE):** 0.05
    - **Desviación Estándar:** 0.02
    - **Umbral:** 0.13
    - **Número de Anomalías:** 63
    - **Observaciones:** Este modelo presenta un MAE bajo y una desviación estándar muy baja, indicando que el modelo es bastante preciso en la reconstrucción de los datos. Sin embargo, el número de anomalías detectadas es menor (63), lo que sugiere que el modelo puede ser menos sensible, posiblemente perdiendo algunas anomalías que el LSTM pudo identificar. Alta precisión en la detección de anomalías con un MAE bajo pero puede no capturar patrones secuenciales tan bien como LSTM.

3. Isolation Forest:
    - **Número de Anomalías:** 73
    - **Observaciones:** Isolation Forest es un enfoque de detección de anomalías que se basa en la separación de instancias de datos. Dado que este método se centra en las instancias más inusuales (picos altos y bajos), es efectivo para detectar puntos extremos, pero puede perder algunos patrones subyacentes en los datos.