Exposición Redes neuronales recurrentes y LMST

Carlos Mauricio Arteaga B..
Miguel Ángel Pablos P.

Series temporales Financieras

Instalación y Librerías

```
!pip install yfinance tensorflow pandas numpy matplotlib
!pip install --quiet tensorflow
```

- Se Instalan e importan los paquetes necesarios:
  - Tensorflow, Pandas, Numpy Matplotlib yfinanice

```
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
print('Librerías cargadas')
```

Yfinance: Librería que permite descargar información financiera desde Yahoo Finance

Series temporales Finaniceras

Descargar datos Financieros y separar los datos.

Se descarga la información financiera del S&P 500 ETF de los últimos 10 años 01/01/2015 al 30/04/2025, y se filtra la data solo en 2 columnas (Close, Volumen)

- **Close** = Precio de cierre
- **Volume** = Cantidad de acciones negociadas

Total de registros: 2597 registros diarios.

```
# Descargar datos diarios de SPY (S&P 500 ETF) desde 2015 hasta 2020
data = yf.download("SPY", start="2015-01-01", end="2025-05-01")
# Usaremos las columnas 'Close' (precio de cierre) y 'Volume'
data = data[['Close', 'Volume']]
print("Datos descargados:")
print(data.head(3))
print("...\n", data.tail(3))
print(f"\nTotal de registros: {len(data)}")
```

```
# Separar entrenamiento (2015-2023) y prueba (2024)
train_data = data['2015':'2023']
test_data  = data[data.index.year >= 2024]
```

Se dividen los datos de entrenamiento y test:

Entrenamiento: 2015 a 2023.
Test: 2024 al 2025

```
Datos de entrenamiento: 2015-01-02 a 2023-12-29 (2264 registros)
Datos de prueba: 2024-01-02 a 2025-04-30 (333 registros)
```

Series temporales Financieras

1 Predicción univariante con un Perceptrón Simple

```python
# Construir pares (X -> y) usando el cierre anterior para predecir el cierre actual
def crear_lags(data_series):
    X = data_series.shift(1).dropna() # desplazado 1 (Close anterior)
    y = data_series.loc[X.index]                # el valor de cierre correspondiente
    return X, y

X_train_uni, y_train_uni = crear_lags(train_data['Close'])
X_test_uni,  y_test_uni  = crear_lags(test_data['Close'])

print(f"Pares entrenamiento: X={X_train_uni.shape}, y={y_train_uni.shape}")
print(f"Pares prueba: X={X_test_uni.shape}, y={y_test_uni.shape}")
```

```
Pares entrenamiento: X=(2263, 1), y=(2263, 1)
Pares prueba: X=(332, 1), y=(332, 1)
```

En las variables de los datos de entrenamiento y test se les crean pares de datos donde cada registro contiene el precio de cierre del día anterior como entrada y el precio de cierre del día actual como salida, permitiendo que un modelo aprenda la relación entre ambos

Series temporales Finaniceras

Definir y entrenar el percepton simple.

```python
# Definir el perceptrón simple
model_perceptron = Sequential([
    Dense(1, activation='linear', input_shape=(1,))  # una neurona con entrada de dimensión 1
])
model_perceptron.compile(optimizer='adam', loss='mse')

# Entrenar el modelo en los datos de entrenamiento
history = model_perceptron.fit(X_train_uni, y_train_uni,
                               epochs=50, batch_size=16, verbose=0)

# Evaluar en los datos de prueba
mse_test = model_perceptron.evaluate(X_test_uni, y_test_uni, verbose=0)
print(f"Error cuadrático medio en prueba: {mse_test:.4f}")
```

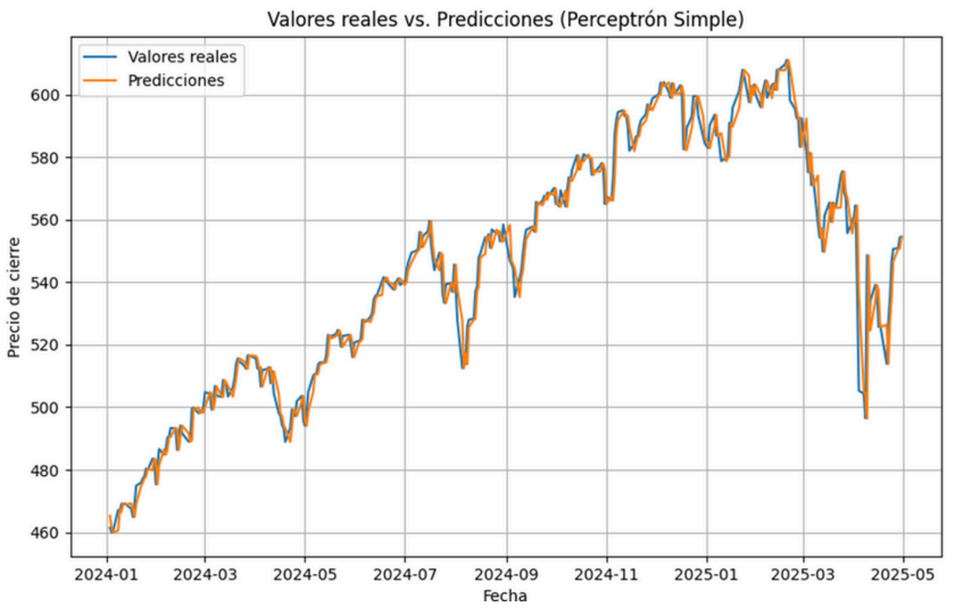
Se implementa un perceptrón simple usando TensorFlow/Keras para predecir el precio de cierre de SPY basado en el valor del día anterior.

Se realiza un modelo secuencial con una sola neurona. Se compila con activación Lineal y optimizador Adam.

Se entrena el modelo:

- Epochs = 50
- Bartch _Sice = 16, 32, 64 y 100

```
Error cuadrático medio en prueba con batch 16: 39.4874
Error cuadrático medio en prueba con batch 32: 38.0912
Error cuadrático medio en prueba con batch 64: 42.8631
Error cuadrático medio en prueba con batch 100: 268384.8438
```



Se aplica la raíz para dejar en las mismas unidades de precio indicando que las predicciones se alejan en 6.20 dólares de los valores reales

```python
rmse_test = np.sqrt(mse_test)
print(f"RMSE en prueba: {rmse_test:.2f} dólares")
```

```
RMSE en prueba: 6.20 dólares
```

Series temporales Financieras

2. Predicción univariante con un Perceptrón Multicapa (MLP)

Se realiza un modelo secuencial multicapa.
La capa oculta con 10 neuronas y se activa con función relu

La capa de salida con activación lineal

Se compilan con y optimizador Adam y la funcion de perdida es el MSE.

Se entrena el modelo:

- Epochs = 50
- Bartch _Sice = 16

Se aplica la raíz para dejar en las mismas unidades de precio indicando que las predicciones se alejan en 6.20 dólares de los valores reales

```python
model_mlp = Sequential([
    Dense(10, activation='relu', input_shape=(1,)),  # capa oculta de 10 neuronas
    Dense(1, activation='linear')                    # salida lineal
])
model_mlp.compile(optimizer='adam', loss='mse')

# Entrenar el modelo MLP
history_mlp = model_mlp.fit(X_train_uni, y_train_uni,
                            epochs=50, batch_size=16, verbose=0)

# Evaluar en prueba
mse_test_mlp = model_mlp.evaluate(X_test_uni, y_test_uni, verbose=0)
print(f"MSE en prueba (MLP): {mse_test_mlp:.4f}")
```

```
MSE en prueba (MLP): 40.8473
```

```
RMSE en prueba: 6.39 dólares
```

Series temporales Finaniceras

3. Predicción multivariante con MLP (múltiples variables de entrada)

El modelo ahora aprende a partir de 2 variables independientes: Close y Volume
Se Crea el desplazamiento temporal para que los datos iniciales sean los del día anterior. Adicional se define como la variable objetivo o precio de cierre como el valor del día actual.

Se aplica estos desplazamientos a los conjuntos de datos de entrenamiento y testeo

Se normaliza la variable Volume para dejarla en un rango similar al precio.

```python
def crear_lags_multivariado(data_frame):
    # data_frame tiene columnas ['Close', 'Volume']
    X = data_frame.shift(1).dropna()   # desplazar todo 1 día
    y = data_frame['Close'].loc[X.index]  # el cierre correspondiente
    return X, y

X_train_multi, y_train_multi = crear_lags_multivariado(train_data)
X_test_multi, y_test_multi = crear_lags_multivariado(test_data)
print("Columnas de X:", list(X_train_multi.columns))
print(f"Pares entrenamiento multivariado: X={X_train_multi.shape}, y={y_train_multi.shape}")
```

```
Columnas de X: [('Close', 'SPY'), ('Volume', 'SPY')]
Pares entrenamiento multivariado: X=(2263, 2), y=(2263, 1)
Pares entrenamiento multivariado: X=(332, 2), y=(332, 1)
```

```python
X_train_multi['Volume'] /= 1e8
X_test_multi['Volume']  /= 1e8
```

Series temporales Financieras

3. Definir y entrenar el MLP multivariante

Se implementa un modelo de MLP multivariado debido a que el modelo cuenta con 2 variables de entrada.

Se realiza un modelo secuencial con 10 neuronas en la capa oculta y función de activación RELU.
La capa de salida con funcion de activación linear.

Se compila con optimizador Adam y funcion de perdia MSE.

Se entrena el modelo:

- Epochs = 50
- Bartch _Sice = 16

Se aplica la raíz para dejar en las mismas unidades de precio indicando que las predicciones se alejan en 6.20 dólares de los valores reales

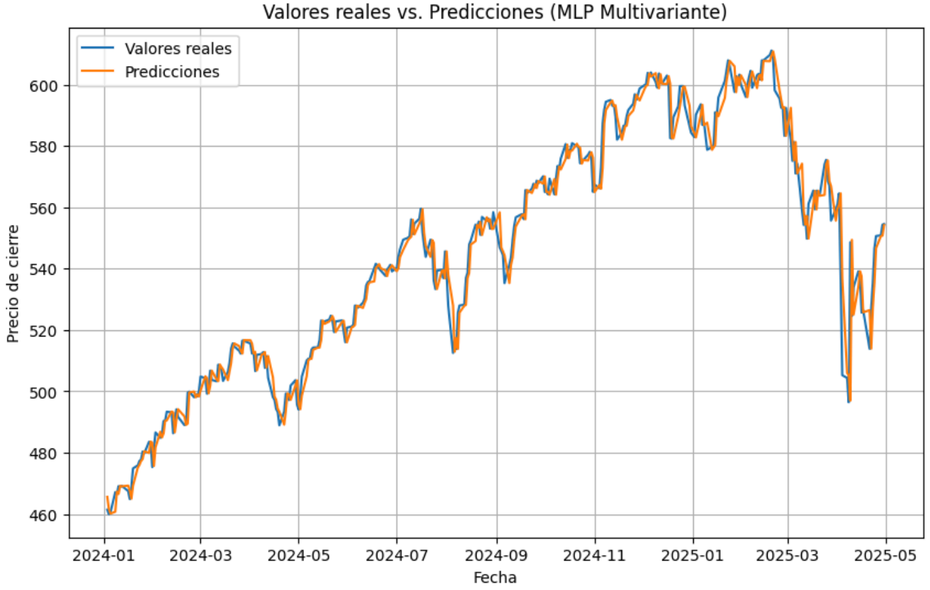
```python
model_mlp_multi = Sequential([
    Dense(10, activation='relu', input_shape=(2,)),  # 2 entradas ahora
    Dense(1, activation='linear')
])
model_mlp_multi.compile(optimizer='adam', loss='mse')

# Entrenar el modelo
model_mlp_multi.fit(X_train_multi, y_train_multi, epochs=50, batch_size=16, verbose=0)

# Evaluar en prueba
mse_test_multi = model_mlp_multi.evaluate(X_test_multi, y_test_multi, verbose=0)
print(f"MSE en prueba (multivariado): {mse_test_multi:.3f}")
```

```
MSE en prueba (multivariado): 38.123
```

```
RMSE en prueba: 6.17 dólares
```



Series temporales Finaniceras

4. Clasificación multiclase de tendencias semanales (Softmax)

Se convierte en un problema de clasificación supervisada, en el que los datos se etiquetaran de acuerdo a su comportamiento semanal:

Se calcula la variación semanal en porcentaje

- Sube: Cuando la variación es + 0.8%
- Baja: Cuando las variaciones son - 0.8%
- Estable: Cuando las varioanciones no superas + o - 0.8%

Se dividen los datos en entrenamiento y testeo

```python
# Convertir a serie semanal (resample al viernes con último valor)
weekly = data.resample('W-FRI').last()

# Calcular rendimiento porcentual semanal
weekly['Return'] = weekly['Close'].pct_change() * 100  # en %
weekly['Vol_prev'] = weekly['Volume'].shift(1)         # volumen semana previa
weekly = weekly.dropna()

# Definir la etiqueta de clase
threshold = 0.8  # 0.6% umbral
conditions = [
    weekly['Return'] > threshold,    # sube
    weekly['Return'] < -threshold   # baja
]
choices = [2, 0]  # 2 = Sube, 0 = Baja
weekly['Class'] = np.select(conditions, choices, default=1)  # 1 = Estable

# Features: retorno previo y volumen previo (de la semana anterior)
X_clas = weekly[['Return', 'Vol_prev']].shift(1).dropna()
y_clas = weekly['Class'].loc[X_clas.index]

# Separar entrenamiento (2015-2023) y prueba (2024) para clasificación
X_train_clas = X_clas[X_clas.index.year < 2023]
y_train_clas = y_clas[y_clas.index.year < 2023]
X_test_clas = X_clas[X_clas.index.year == 2024]
y_test_clas = y_clas[y_clas.index.year == 2024]

print(f"Semanas entrenamiento: {len(X_train_clas)}, Semanas prueba: {len(X_test_clas)}")
print("Distribución de clases en entrenamiento:", np.bincount(y_train_clas))
```

```
Semanas entrenamiento: 468, Semanas prueba: 70
Distribución de clases en entrenamiento: [122 160 186]
```

Series temporales Financeras

Definir y entrenar Softmax

Se implementa una red neuronal de clasificación para categorizar el comportamiento del mercado.

Se realiza un modelo secuencial con 8 neuronas en la capa oculta y función de activación RELU.
La capa de salida contiene 3 neuronas con función de activación Softmax.

Se compila con optimizador Adam y función de perdida Entropia cruzada.

Se entrena el modelo:

- Epochs = 50
- Bartch _Sice = 16

```python
# Definir MLP para clasificación (entrada: Return_prev, Vol_prev)
#from tensorflow.keras.layers import LeakyReLU
model_clas = Sequential([
    Dense(8, activation='relu', input_shape=(2,)),
    Dense(3, activation='softmax')
])
model_clas.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])


# Entrenar la red de clasificación
model_clas.fit(X_train_clas, y_train_clas, epochs=50, batch_size=16, verbose=0)

# Evaluar en semanas de prueba 2024 y 2025
loss, acc = model_clas.evaluate(X_test_clas, y_test_clas, verbose=0)
print(f"Precisión en semanas: {acc*100:.1f}%")
```

```
Precisión en semanas: 42.9%
```

```
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step
Semana       ClaseReal  ClasePred  (Probabilidades [Baja,Estable,Sube])
2024-01-05    0         1          ([0. 1. 0.])
2024-01-12    2         1          ([0. 1. 0.])
2024-01-19    2         1          ([0. 1. 0.])
2024-01-26    2         1          ([0. 1. 0.])
2024-02-02    2         1          ([0. 1. 0.])
2024-02-09    2         1          ([0. 1. 0.])
2024-02-16    1         1          ([0. 1. 0.])
2024-02-23    2         1          ([0. 1. 0.])
2024-03-01    2         1          ([0. 1. 0.])
2024-03-08    1         1          ([0. 1. 0.])
```

Taller_2

1. Instalación y Librerías

- Se Instalan paquetes necesarios para
  - Obtener datos económicos,
    - pandas_datareader, permite descargar datos financieros y macroeconómicos (como tasas de interés) desde fuentes como FRED, Yahoo Finance, etc.
    - fredapi: cliente oficial para descargar datos desde FRED (Federal Reserve Economic Data).
  - Manipular y visualizar datos
  - Construir modelos

```
# Instalación de librerías necesarias (ejecutar si no están instaladas)
!pip install pandas_datareader fredapi matplotlib scikit-learn torch torchvision torchaudio

```

Taller_2

2. Obtención de Datos

- Se utiliza pandas_datareader para descargar la serie DGS10 (tasa de interés del bono a 10 años) desde FRED.
- Período: 1 de enero de 2010 al 31 de diciembre de 2024.
- web.DataReader, es una función de pandas_datareader que permite conectar con fuentes externas.
- 'DGS10', Tasa de interés de bonos del Tesoro a 10 años (U.S. 10-Year Treasury Constant Maturity Rate)

```python
import pandas as pd
from pandas_datareader import data as pdr
import datetime
import numpy as np


# Definir el período de tiempo para la serie temporal (ejemplo: 10 años de datos)
start_date = datetime.datetime(2010, 1, 1)
end_date   = datetime.datetime(2024, 12, 31)

# Descargar la serie de tasa de interés a 10 años (DGS10) desde FRED
df = pdr.DataReader('DGS10', 'fred', start_date, end_date)

# Mostrar las primeras y últimas filas para verificar
print("Datos descargados:")
print(df.head(5))
print("...")
print(df.tail(5))

```

```
Datos descargados:
            DGS10
DATE
2010-01-01    NaN
2010-01-04   3.85
2010-01-05   3.77
2010-01-06   3.85
2010-01-07   3.85
...
            DGS10
DATE
2024-12-25    NaN
2024-12-26   4.58
2024-12-27   4.62
2024-12-30   4.55
2024-12-31   4.58
```

Taller_2

3. Limpieza y Exploración

- Se eliminan las filas con valores faltantes (NaN), comunes en días no laborables, festivos, etc.
- Se pueden utilizar diferentes tipos de imputación de datos como utilizar el ultimo dato conocido, pero para simplificar el entrenamiento se hace drop out a los valores faltantes.
- Se muestra el número de registros y las primeras/ últimas filas.
- Se grafica la serie temporal para inspección visual.

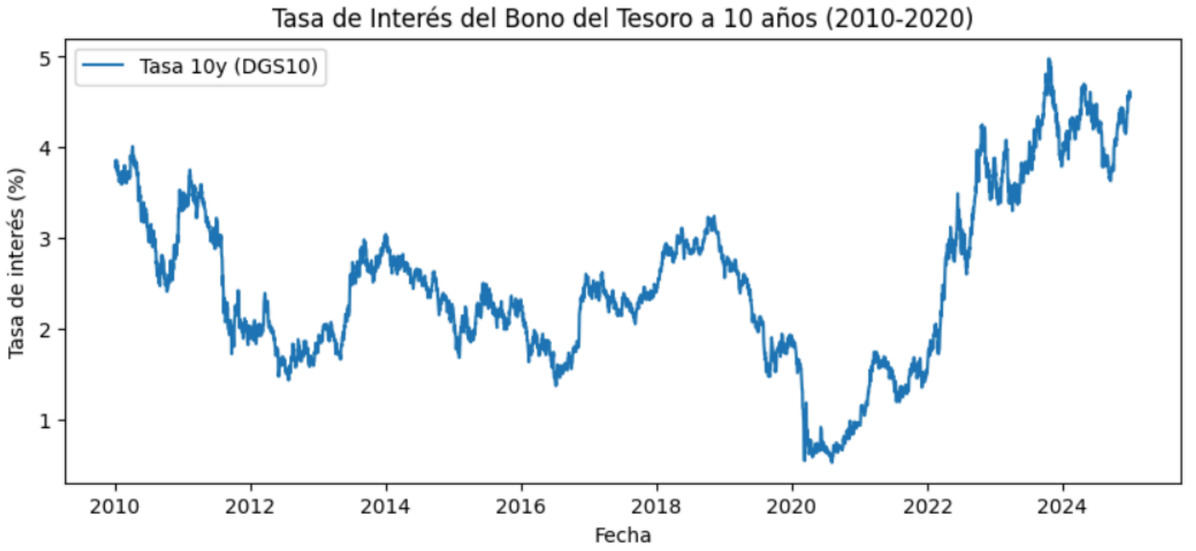
```python
# Eliminar filas con datos faltantes (NaN), típicos de fines de semana/feriados sin cotización
df = df.dropna()

print(f"Total de registros después de eliminar NaN: {len(df)}")
print(df.head(3))
print(df.tail(3))

# Graficar la serie temporal para visualizarla
import matplotlib.pyplot as plt

plt.figure(figsize=(10,4))
plt.plot(df.index, df['DGS10'], label='Tasa 10y (DGS10)')
plt.title('Tasa de Interés del Bono del Tesoro a 10 años (2010-2020)')
plt.xlabel('Fecha')
plt.ylabel('Tasa de interés (%)')
plt.legend()
plt.show()

```



Taller_2

4. Preparación de Datos

- shift(n) desplaza la serie n días hacia abajo.
- Se crean 3 nuevas columnas, lag1, lag2, lag3
- Estas variables son las entradas (features) para el modelo, porque reflejan la historia reciente de la serie.
- df['dia_semana'] = df.index.dayofweek
  - Extrae el día de la semana de cada fecha
  - 0 = lunes, 1 = martes, ..., 4 = viernes

```python
# Crear columnas de lags t-1, t-2, t-3
df['lag1'] = df['DGS10'].shift(1)
df['lag2'] = df['DGS10'].shift(2)
df['lag3'] = df['DGS10'].shift(3)

# Crear columna con el día de la semana de la fecha (0=Lunes, 6=Domingo)
df['dia_semana'] = df.index.weekday  # Monday=0, Sunday=6

# Eliminar filas iniciales con NaN generados por los lags
df = df.dropna()
print("Columnas del DataFrame después de añadir lags y día de semana:")
print(df.head(3))

```

```
Columnas del DataFrame después de añadir lags y día de semana:
            DGS10  lag1  lag2  lag3  dia_semana
DATE
2010-01-07   3.85  3.85  3.77  3.85           3
2010-01-08   3.83  3.85  3.85  3.77           4
2010-01-11   3.85  3.83  3.85  3.85           0
```

DATE	DGS10	lag1	lag2	lag3	dia_semana
2010-01-07	3.85	3.85	3.77	3.85	3
2010-01-08	3.83	3.85	3.85	3.77	4
2010-01-11	3.85	3.83	3.85	3.85	0
2010-01-12	3.74	3.85	3.83	3.85	1
2010-01-13	3.80	3.74	3.85	3.83	2

dia_semana	count
1	774
2	770
3	760
4	760
0	685

Taller_2

4. Preparación de Datos

- pd.get_dummies, transforma la columna dia_semana en 5 columnas binarias (diasem_0, ..., diasem_4).
- Se concatenan las variables one-hot al DataFrame.

```python
# Aplicar one-hot encoding a la columna dia_semana
dummies = pd.get_dummies(df['dia_semana'], prefix='diasem', drop_first=False)
df_model = pd.concat([df, dummies], axis=1)

# Verificar resultado de one-hot encoding, solo para los días presentes en los datos
print("Ejemplo de one-hot encoding de día de semana:")
print(df_model[['dia_semana','diasem_0','diasem_1','diasem_2','diasem_3','diasem_4']].head(3))
```

```
Ejemplo de one-hot encoding de día de semana:
            dia_semana  diasem_0  diasem_1  diasem_2  diasem_3  diasem_4
DATE
2010-01-07           3     False     False     False      True     False
2010-01-08           4     False     False     False     False      True
2010-01-11           0      True     False     False     False     False
```

Taller_2

5. División Entrenamiento/Prueba

- Se dividen los datos en 80% entrenamiento y 20% prueba, respetando el orden temporal.
- Se separa características (lag1, lag2, lag3, variables one-hot) y objetivo (DGS10).

```python
# Determinar índice de corte para ~80% entrenamiento, 20% prueba (división temporal)
train_frac = 0.8
train_size = int(len(df_model) * train_frac)
train_df = df_model.iloc[:train_size]
test_df  = df_model.iloc[train_size:]

print(f"Datos de entrenamiento: {len(train_df)} muestras")
print(f"Datos de prueba: {len(test_df)} muestras")

# Separar features X y target y para entrenamiento y prueba
# Check for existing columns in the dataframes
features = [col for col in ['lag1','lag2','lag3','diasem_0','diasem_1','diasem_2','diasem_3','diasem_4','diasem_5','diasem_6'] if col in train_df.columns] #
target = 'DGS10'

X_train = train_df[features].values
y_train = train_df[target].values
X_test  = test_df[features].values # Use the same filtered features for test data
y_test  = test_df[target].values

```

```
Datos de entrenamiento: 2999 muestras
Datos de prueba: 750 muestras
```

Taller_2

6. Escalado (Normalización)

- Se escala los lags (lag1, lag2, lag3) y el objetivo (DGS10) a [0, 1] usando MinMaxScaler.
- Las variables one-hot (dias de la semana) no se escalan (ya están en 0 y 1).
- Se ajustan los escaladores solo con datos de entrenamiento para evitar sesgos.
- values.reshape(-1, 1)), el MinMaxScaler espera una matriz (4,1)... es como transponer la matriz

```python
from sklearn.preprocessing import MinMaxScaler

# Inicializar escaladores
scaler_X = MinMaxScaler(feature_range=(0,1))
scaler_y = MinMaxScaler(feature_range=(0,1))

# Ajustar escalador de X solo con datos de entrenamiento (para evitar
# Importante: solo escalamos las columnas de lags, no las dummies (est
scaler_X.fit(train_df[['lag1','lag2','lag3']])

# Aplicar transformacion a lags de entrenamiento y prueba
X_train_scaled = X_train.copy()
X_test_scaled = X_test.copy()
X_train_scaled[:, :3] = scaler_X.transform(X_train[:, :3])
X_test_scaled[:, :3] = scaler_X.transform(X_test[:, :3])

# Ajustar y aplicar escalador de y (target)
scaler_y.fit(train_df[['DGS10']])
y_train_scaled = scaler_y.transform(y_train.reshape(-1,1)).flatten()
y_test_scaled = scaler_y.transform(y_test.reshape(-1,1)).flatten()
```

Taller_2

7. Modelos de Redes Neuronales

MLP (Perceptrón Multicapa)

Se define un MLP con:

- una capa oculta: nn.Linear(input_dim, hidden_dim) de la forma x@wT+b
- Función de activacion ReLu
- una capa de salida: nn.Linear(hidden_dim, 1) para predecir una tasa.
- input_dim, cantidad de variables de entrada 3 lags y 5 columnas one-hot del día.
- self.out, Toma el vector oculto de tamaño hidden_dim y lo reduce a una sola predicción escalar (la tasa DGS10 del día).
- def forward,
  - self.hidden(x): aplica la capa lineal oculta.
  - self.act, aplica la función ReLU.
  - self.out, aplica la capa de salida lineal.
- La primera capa transforma las 8 características en 20 dimensiones, y la segunda reduce a 1 dimensión (tasa predicha).

```python
import torch
import torch.nn as nn

# Definir el modelo MLP
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(MLP, self).__init__()
        # Capa oculta
        self.hidden = nn.Linear(input_dim, hidden_dim)
        # Capa de salida
        self.out = nn.Linear(hidden_dim, 1)
        # Función de activación
        self.act = nn.ReLU()

    def forward(self, x):
        # x tiene forma [batch_size, input_dim]
        h = self.act(self.hidden(x))
        y_pred = self.out(h)
        return y_pred

# Inicializar MLP con dimensiones
input_dim = X_train_scaled.shape[1]   # debería ser 10
hidden_dim = 20  # escogemos 16 neuronas en la capa oculta (
model_mlp = MLP(input_dim, hidden_dim)
print(model_mlp)
```

Taller_2

7. Modelos de Redes Neuronales

LSTM

- Se convierten los arreglos de Numpy contenidos en el dataframe a tensores de torch para poder realizar el entrenamiento.
- seq_len = 3, secuencia que le pasaremos a la LSTM será de longitud 3 - usamos los lag1, lag2 y lag3.
- X_train_seq = X_train_tensor[:, :seq_len],
  - Se toman solo las primeras 3 columnas (lag1, lag2, lag3).
  - X_train_seq.unsqueeze(2), le agrega una dimensión extra

```
# Convertir los datos escalados a tensores de PyTorch
X_train_tensor = torch.tensor(X_train_scaled.astype(np.float32), dtype=torch.float32)
X_test_tensor = torch.tensor(X_test_scaled.astype(np.float32), dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_scaled, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test_scaled, dtype=torch.float32)

# Reformatear X para LSTM: de [batch, features] a [batch, seq_len, input_size]
# En X_train_scaled, las primeras 3 columnas son los lags; las columnas 3-9 son las dumm
# Para LSTM, tomaremos sólo los lags como secuencia de longitud 3, y manejaremos las dum
seq_len = 3
# Extraer las columnas de lags como secuencia
X_train_seq = X_train_tensor[:, :seq_len]  # shape [batch, 3]
X_test_seq = X_test_tensor[:, :seq_len]
# Añadir dimensión de input_size=1 para cada valor
X_train_seq = X_train_seq.unsqueeze(2)  # shape [batch, 3, 1]
X_test_seq = X_test_seq.unsqueeze(2)    # shape [batch, 3, 1]
# Extraer las one-hot dummies aparte
X_train_dow = X_train_tensor[:, seq_len:]  # shape [batch, 7]
X_test_dow = X_test_tensor[:, seq_len:]    # shape [batch, 7]
```

Taller_2

7. Modelos de Redes Neuronales

LSTM

- LSTM captura patrones temporales usando los lags como secuencia.
- Se concatena con la codificación one-hot del día de la semana antes de la predicción.
- Estructura del modelo:
  - LSTM: procesa la secuencia [lag1, lag2, lag3]
  - fc: capa lineal final que combina salida de la LSTM + información del día.
- Forward pass:
  - Se toma la salida del último paso temporal de la LSTM (last_hidden).
  - Se concatena con las variables categóricas (dow_onehot).
  - Se genera una única predicción continua (y_pred).
- Hiperparámetros personalizables:
  - input_size = 1 → un valor por paso temporal.
  - hidden_size = 32 → tamaño del estado oculto.
  - num_layers = 1 → una capa LSTM.
  - dow_feature_size = 7 → columnas one-hot del día.

```python
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, dow_feature_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        # Capa LSTM
        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size, num_layers=num_layers, batch_first=True)
        # Capa final totalmente conectada: toma hidden_state + one-hot día_semana
        self.fc = nn.Linear(hidden_size + dow_feature_size, 1)

    def forward(self, seq, dow_onehot):
        # seq: [batch, seq_len, input_size]
        # dow_onehot: [batch, dow_feature_size]
        batch_size = seq.size(0)
        # LSTM forward
        lstm_out, (h_n, c_n) = self.lstm(seq)
        # lstm_out: [batch, seq_len, hidden_size] (porque batch_first=True)
        # h_n: [num_layers, batch, hidden_size] -> estado oculto final de cada capa
        # Tomamos el estado oculto de la última capa de LSTM para el último elemento de la secuencia:
        # Podemos obtenerlo de lstm_out directamente (último tiempo):
        last_hidden = lstm_out[:, -1, :]  # shape [batch, hidden_size]
        # Concatenar con vector one-hot
        combined = torch.cat((last_hidden, dow_onehot), dim=1)  # shape [batch, hidden_size + dow_feature_size]
        # Pasar por la capa totalmente conectada para obtener predicción
        y_pred = self.fc(combined)
        return y_pred
```

```python
# Inicializar el modelo LSTM
input_size = 1        # un valor de serie por tiempo
hidden_size = 32      # tamaño del estado oculto LSTM (hiperparámetro, aquí 32 neuronas)
num_layers = 1        # una capa LSTM (podríamos usar más)
dow_feature_size = X_train_dow.shape[1]  # tamaño de vector one-hot de día de semana (debería ser 7)
model_lstm = LSTMModel(input_size, hidden_size, num_layers, dow_feature_size)
print(model_lstm)
```

```
LSTMModel(
  (lstm): LSTM(1, 32, batch_first=True)
  (fc): Linear(in_features=37, out_features=1, bias=True)
)
```

Taller_2

7. Modelos de Redes Neuronales

CNN 1D

- Arquitectura de la red:
  - Conv1D: extrae patrones locales de la secuencia (lags) con una ventana deslizante (kernel_size).
  - ReLU: añade no linealidad después de la convolución.
  - Flatten: aplana la salida convolucional antes de la capa final.
  - Linear: combina los patrones aprendidos con las variables categóricas del día (dow_onehot) para predecir la tasa.
- Cálculo del tamaño de salida de la convolución:
  - L_out = seq_len - kernel_size + 1
  - El tamaño total de la capa lineal es: out_channels * L_out + dow_feature_size.
- Parámetros definidos:
  - in_channels = 1: un canal de entrada (secuencia univariada).
  - out_channels = 16: cantidad de filtros en la convolución.
  - kernel_size = 2: tamaño de la ventana de la convolución.
  - dow_feature_size = 7: tamaño del vector one-hot del día de la semana.

```
# Preparar entrada para CNN: [batch, channels, seq_len]
X_train_cnn = X_train_seq.permute(0, 2, 1)  # va de [batch, seq_len, 1] a [batch, 1, seq_len]
X_test_cnn = X_test_seq.permute(0, 2, 1)
# (X_train_dow y X_test_dow siguen igual para concatenarlos luego)

```

```
class CNNModel(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, dow_feature_size):
        super(CNNModel, self).__init__()
        # Capa convolucional
        self.conv1 = nn.Conv1d(in_channels=in_channels, out_channels=out_channels, kernel_size=kernel_size)
        # Función de activación
        self.act = nn.ReLU()
        # Capa de aplanamiento (podemos usar nn.Flatten o lo haremos manual en forward)
        # Capa final fully-connected
        # Nota: necesitamos calcular el tamaño de entrada a esta capa luego de la conv.
        # Si seq_len = 3 y kernel_size=2, la salida de conv tendrá longitud 2 (3-2+1=2).
        # out_channels = 16, entonces total features = 16*2 = 32. Luego se concatenan 7 dummies -> 39.
        conv_output_size = out_channels * (seq_len - kernel_size + 1)
        self.fc = nn.Linear(conv_output_size + dow_feature_size, 1)

    def forward(self, x, dow_onehot):
        # x: [batch, 1, seq_len]
        h = self.act(self.conv1(x))        # h: [batch, out_channels, L_out] con L_out = seq_len - kernel_size + 1
        # Aplanar salida convolucional
        h_flat = h.view(h.size(0), -1)     # shape [batch, out_channels * L_out]
        # Concatenar con one-hot de día semana
        combined = torch.cat((h_flat, dow_onehot), dim=1)  # [batch, conv_output_size + dow_feature_size]
        y_pred = self.fc(combined)
        return y_pred
```

```
# Inicializar modelo CNN
in_channels = 1
out_channels = 16
kernel_size = 2
dow_feature_size = X_train_dow.shape[1]  # 7
model_cnn = CNNModel(in_channels, out_channels, kernel_size, dow_feature_size)
print(model_cnn)

```

```
CNNModel(
  (conv1): Conv1d(1, 16, kernel_size=(2,), stride=(1,))
  (act): ReLU()
  (fc): Linear(in_features=37, out_features=1, bias=True)
)
```

Taller_2

8. Entrenamiento

- Modo entrenamiento:
  - model_mlp.train(): activa funciones como dropout (si existieran) y permite actualizar pesos.
  - optimizer_mlp.zero_grad(): reinicia los gradientes antes de cada paso.
- Forward pass:
  - y_pred = model_mlp(X_train_tensor): predice los valores continuos directamente a partir de todas las features.
- Cálculo de la pérdida:
  - loss_fn(...): se utiliza MSELoss para medir el error.
  - .flatten(): se aplica para que la predicción tenga la misma forma que y_train_tensor.
- Backpropagation y actualización de pesos:
  - loss.backward(): calcula gradientes.
  - optimizer_mlp.step(): actualiza los pesos del modelo.
- Evaluación opcional cada 10 épocas:
  - model_mlp.eval() y torch.no_grad() se usan para evaluar en el conjunto de prueba sin alterar los gradientes.
  - Se imprime el error en entrenamiento y prueba para monitorear el desempeño.

```python
# Entrenar el modelo MLP
for epoch in range(num_epochs):
    model_mlp.train()  # modo entrenamiento
    optimizer_mlp.zero_grad()
    # Forward pass: MLP toma directamente X_train_tensor (todas features)
    y_pred = model_mlp(X_train_tensor)
    # Calcular pérdida
    loss = loss_fn(y_pred.flatten(), y_train_tensor)  # flatten pred para comparar con vector
    # Backpropagation
    loss.backward()
    optimizer_mlp.step()

    # (Opcional) imprimir cada 10 épocas
    if (epoch+1) % 10 == 0:
        train_mse = loss.item()
        # Calcular MSE en prueba sin gradiente
        model_mlp.eval()
        with torch.no_grad():
            y_pred_test = model_mlp(X_test_tensor)
            test_loss = loss_fn(y_pred_test.flatten(), y_test_tensor)
        print(f"Epoch {epoch+1}/{num_epochs} - MSE entrenamiento: {train_mse:.6f} - MSE prueba: {test_loss.item():.6f}")
```

```
Epoch 10/100 - MSE entrenamiento: 0.038738 - MSE prueba: 0.009154
Epoch 20/100 - MSE entrenamiento: 0.013309 - MSE prueba: 0.072746
Epoch 30/100 - MSE entrenamiento: 0.006192 - MSE prueba: 0.005511
Epoch 40/100 - MSE entrenamiento: 0.003098 - MSE prueba: 0.017829
Epoch 50/100 - MSE entrenamiento: 0.001689 - MSE prueba: 0.002564
Epoch 60/100 - MSE entrenamiento: 0.000950 - MSE prueba: 0.003808
Epoch 70/100 - MSE entrenamiento: 0.000596 - MSE prueba: 0.000853
Epoch 80/100 - MSE entrenamiento: 0.000444 - MSE prueba: 0.000888
Epoch 90/100 - MSE entrenamiento: 0.000394 - MSE prueba: 0.000732
Epoch 100/100 - MSE entrenamiento: 0.000381 - MSE prueba: 0.000745
```

Taller_2

8. Entrenamiento

- Arquitectura híbrida:
  - LSTM captura patrones temporales usando los lags como secuencia.
  - Se concatena con la codificación one-hot del día de la semana antes de la predicción.
  - Estructura del modelo:
  - LSTM: procesa la secuencia [lag1, lag2, lag3] → salida de tamaño hidden_size.
  - fc: capa lineal final que combina salida de la LSTM + información del día.
- Forward pass:
  - Se toma la salida del último paso temporal de la LSTM (last_hidden).
  - Se concatena con las variables categóricas (dow_onehot).
  - Se genera una única predicción continua (y_pred).
  - Hiperparámetros personalizables:
  - input_size = 1 → un valor por paso temporal.
  - hidden_size = 32 → tamaño del estado oculto.
  - num_layers = 1 → una capa LSTM.
  - dow_feature_size = 7 → columnas one-hot del día.

```python
# Entrenar el modelo LSTM
for epoch in range(num_epochs):
    model_lstm.train()
    optimizer_lstm.zero_grad()
    # Forward: LSTM necesita secuencia y one-hot por separado
    y_pred = model_lstm(X_train_seq, X_train_dow)
    loss = loss_fn(y_pred.flatten(), y_train_tensor)
    loss.backward()
    optimizer_lstm.step()

    if (epoch+1) % 10 == 0:
        model_lstm.eval()
        with torch.no_grad():
            y_pred_test = model_lstm(X_test_seq, X_test_dow)
            test_loss = loss_fn(y_pred_test.flatten(), y_test_tensor)
        print(f"Epoch {epoch+1}/{num_epochs} - MSE entrenamiento: {loss.item():.6f} - MSE prueba: {test_loss.item():.6f}")

```

```
Epoch 10/100 - MSE entrenamiento: 0.079393 - MSE prueba: 0.069393
Epoch 20/100 - MSE entrenamiento: 0.043369 - MSE prueba: 0.243265
Epoch 30/100 - MSE entrenamiento: 0.027463 - MSE prueba: 0.073001
Epoch 40/100 - MSE entrenamiento: 0.014179 - MSE prueba: 0.067437
Epoch 50/100 - MSE entrenamiento: 0.002150 - MSE prueba: 0.001057
Epoch 60/100 - MSE entrenamiento: 0.001328 - MSE prueba: 0.007739
Epoch 70/100 - MSE entrenamiento: 0.000868 - MSE prueba: 0.000931
Epoch 80/100 - MSE entrenamiento: 0.000466 - MSE prueba: 0.000653
Epoch 90/100 - MSE entrenamiento: 0.000405 - MSE prueba: 0.000707
Epoch 100/100 - MSE entrenamiento: 0.000346 - MSE prueba: 0.001040
```

Taller_2

8. Entrenamiento

- Formato de entrada para CNN:
  - Los datos de entrada se reorganizan de [batch, seq_len, 1] a [batch, 1, seq_len] con permute, para cumplir con la forma que espera nn.Conv1d.
- Arquitectura de la red:
  - Conv1D: extrae patrones locales de la secuencia (lags) con una ventana deslizante (kernel_size).
  - ReLU: añade no linealidad después de la convolución.
  - Flatten: aplana la salida convolucional antes de la capa final.
  - Linear: combina los patrones aprendidos con las variables categóricas del día (dow_onehot) para predecir la tasa.
- Cálculo del tamaño de salida de la convolución:
  - L_out = seq_len - kernel_size + 1
  - El tamaño total de la capa lineal es: out_channels * L_out + dow_feature_size.
- Parámetros definidos:
  - in_channels = 1: un canal de entrada (secuencia univariada).
  - out_channels = 16: cantidad de filtros en la convolución.
  - kernel_size = 2: tamaño de la ventana de la convolución.
  - dow_feature_size = 7: tamaño del vector one-hot del día de la semana.

```python
# Entrenar el modelo CNN
for epoch in range(num_epochs):
    model_cnn.train()
    optimizer_cnn.zero_grad()
    # Forward: CNN también toma secuencia (como [batch,1,seq_len]) y one-hot
    y_pred = model_cnn(X_train_cnn, X_train_dow)
    loss = loss_fn(y_pred.flatten(), y_train_tensor)
    loss.backward()
    optimizer_cnn.step()

    if (epoch+1) % 10 == 0:
        model_cnn.eval()
        with torch.no_grad():
            y_pred_test = model_cnn(X_test_cnn, X_test_dow)
            test_loss = loss_fn(y_pred_test.flatten(), y_test_tensor)
        print(f"Epoch {epoch+1}/{num_epochs} - MSE entrenamiento: {loss.item():.6f} - MSE prueba: {test_loss.item():.6f}")

```

```
Epoch 10/100 - MSE entrenamiento: 0.057109 - MSE prueba: 0.027830
Epoch 20/100 - MSE entrenamiento: 0.021567 - MSE prueba: 0.140737
Epoch 30/100 - MSE entrenamiento: 0.013998 - MSE prueba: 0.077645
Epoch 40/100 - MSE entrenamiento: 0.008841 - MSE prueba: 0.030452
Epoch 50/100 - MSE entrenamiento: 0.004014 - MSE prueba: 0.028798
Epoch 60/100 - MSE entrenamiento: 0.001268 - MSE prueba: 0.005018
Epoch 70/100 - MSE entrenamiento: 0.000383 - MSE prueba: 0.002137
Epoch 80/100 - MSE entrenamiento: 0.000316 - MSE prueba: 0.000940
Epoch 90/100 - MSE entrenamiento: 0.000343 - MSE prueba: 0.000751
Epoch 100/100 - MSE entrenamiento: 0.000309 - MSE prueba: 0.000925
```

Taller_2

9. Evaluación de Desempeño

- Evaluación sin gradientes:
  - with torch.no_grad() asegura que no se calculen gradientes durante la inferencia (más eficiente y seguro).
- Inversión del escalado:
  - Se aplicó scaler_y.inverse_transform(...) para convertir las predicciones y valores reales a la escala original antes de calcular los errores.
- Métricas utilizadas:
  - MAE: mide el promedio de las diferencias absolutas entre predicción y realidad.
  - RMSE: penaliza más fuertemente los errores grandes, útil para evaluar la precisión.
- Comparación de resultados:
  - Los tres modelos muestran errores similares.
  - MLP tiene el MAE más bajo, mientras que el CNN obtiene el RMSE más bajo.

```python
# Calcular MAE y RMSE para cada modelo
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Obtener predicciones en el conjunto de prueba para cada modelo
model_mlp.eval()
with torch.no_grad():
    pred_test_mlp = model_mlp(X_test_tensor)
    pred_test_mlp_orig = scaler_y.inverse_transform(pred_test_mlp.reshape(-1, 1)).flatten()  # Invertir escalado

model_lstm.eval()
with torch.no_grad():
    pred_test_lstm = model_lstm(X_test_seq, X_test_dow)
    pred_test_lstm_orig = scaler_y.inverse_transform(pred_test_lstm.reshape(-1, 1)).flatten()  # Invertir escalado

model_cnn.eval()
with torch.no_grad():
    pred_test_cnn = model_cnn(X_test_cnn, X_test_dow)
    pred_test_cnn_orig = scaler_y.inverse_transform(pred_test_cnn.reshape(-1, 1)).flatten()  # Invertir escalado

# Obtener los valores reales en la escala original (invertido el escalado)
y_test_orig = scaler_y.inverse_transform(y_test.reshape(-1, 1)).flatten()
```

```python
mae_mlp  = mean_absolute_error(y_test_orig, pred_test_mlp_orig)
mae_lstm = mean_absolute_error(y_test_orig, pred_test_lstm_orig)
mae_cnn  = mean_absolute_error(y_test_orig, pred_test_cnn_orig)
# Calculate RMSE without 'squared' argument and take the square root manually
rmse_mlp  = np.sqrt(mean_squared_error(y_test_orig, pred_test_mlp_orig))
rmse_lstm = np.sqrt(mean_squared_error(y_test_orig, pred_test_lstm_orig))
rmse_cnn  = np.sqrt(mean_squared_error(y_test_orig, pred_test_cnn_orig))


print("Desempeño en el conjunto de prueba:")
print(f"MLP -> MAE: {mae_mlp:.4f} , RMSE: {rmse_mlp:.4f}")
print(f"LSTM -> MAE: {mae_lstm:.4f} , RMSE: {rmse_lstm:.4f}")
print(f"CNN  -> MAE: {mae_cnn:.4f} , RMSE: {rmse_cnn:.4f}")
```

```
Desempeño en el conjunto de prueba:
MLP -> MAE: 9.7377 , RMSE: 9.9108
LSTM -> MAE: 9.8067 , RMSE: 9.9867
CNN  -> MAE: 9.7893 , RMSE: 9.9692
```

Taller_2

10. Visualización

- Eje X: Fecha
  - Representa el tiempo en el que se hacen las predicciones, utilizando el índice de test_df.
- Eje Y: Tasa de interés (%)
  - Muestra el valor real y los valores predichos por cada modelo en porcentaje.
- Línea negra (Real)
  - Representa la serie temporal original de la tasa de interés real.
- Líneas de colores (MLP, LSTM, CNN)
  - Representan las predicciones hechas por cada modelo.
  - Se observa que las predicciones siguen el patrón general pero subestiman consistentemente los valores reales, probablemente por una combinación entre escalado, arquitectura y regularización del modelo.
- Objetivo del gráfico
  - Evaluar visualmente qué tan bien se ajustan los modelos al comportamiento real de la serie de tiempo.

```
 7 test_df = test_df[:len(y_test_orig)]
 8
 9 # Graficar las predicciones vs reales para un subconjunto del periodo de prueba
10 plt.figure(figsize=(10, 6))
11 plt.plot(test_df.index, y_test_orig, label='Real', color='black')
12 plt.plot(test_df.index, pred_test_mlp_orig, label='Predicción MLP', alpha=0.7)
13 plt.plot(test_df.index, pred_test_lstm_orig, label='Predicción LSTM', alpha=0.7)
14 plt.plot(test_df.index, pred_test_cnn_orig, label='Predicción CNN', alpha=0.7)
15 plt.legend()
16 plt.title('Comparación de Predicciones vs Real - Datos de Prueba')
17 plt.xlabel('Fecha')
18 plt.ylabel('Tasa de interés (%)')
19 plt.show()
```