Introduzione

La complessità dei mercati finanziari e nel nostro caso valutari, rendono essenziale l'impiego di modelli di Machine Learning e Deep Learning avanzati per riuscire a catturare le dinamiche interne dei mercati stessi, spesso difficili da interpretare con il solo ragionamento umano. Perciò è necessario affidarsi a questi strumenti per migliorare l'accuratezza ed ottimizzare l'esito delle previsioni. In questo progetto, ci siamo dunque concentrati sulla modellazione delle serie storiche dei tassi di cambio EUR/USD e GBP/USD, con l'obiettivo di individuare il modello più efficace e allo stesso tempo efficiente per la previsione dei prezzi futuri. Data l'ampia disponibilità di strumenti offerti dal corso, siamo stati in grado di scegliere con precisione i modelli che garantivano le migliori prestazioni in termini di accuratezza e generalizzazione, adattandoli alle caratteristiche specifiche dei dataset analizzati.

Scelta dei Modelli

Abbiamo selezionato due modelli principali per affrontare il problema delle previsioni:

1. LSTM (Long Short-Term Memory): Questo primo modello è notoriamente uno dei modelli più diffusi per la modellazione delle serie temporali ed occuperà gran parte del nostro progetto date le numerose modifiche che abbiamo voluto apportare al fine di raggiungere i risultati migliori possibili.
2. XGBoost (Extreme Gradient Boosting): Un algoritmo basato su alberi decisionali, in grado di gestire dati strutturati e fornire previsioni robuste. Il modello XGBoost è stato incluso per confrontare le prestazioni delle reti neurali ricorrenti con quelle di un modello di boosting in modo tale da avere due diverse prospettive sull'accuratezza della previsione dei tassi di cambio. Da un lato, le reti neurali ricorrenti come LSTM, sono progettate per catturare dipendenze a lungo termine nelle serie temporali, mentre dall'altro, XGBoost sfrutta un approccio di boosting adattivo che può essere molto efficace nel catturare pattern non lineari nei dati, senza richiedere una memoria esplicita delle dipendenze temporali.

L'analisi si è concentrata sulla valutazione delle prestazioni dei modelli in termini di:

- Errore Quadratico Medio (MSE): per quantificare la deviazione media delle previsioni dai valori reali (espresso anche nella forma a radice attraverso la metrica RMSE)
- Errore Assoluto Medio (MAE): per valutare l'errore medio in termini di valore assoluto.
- Coefficiente di Determinazione (R^2): per misurare la capacità del modello di spiegare la variabilità dei dati.

Il primo passaggio in assoluto è stato ovviamente importare i due dataset in formato CSV che abbiamo scaricato direttamente dal database di serie storiche economico-finanziarie Yahoo-Finance filtrando i dati per cadenza giornaliera e ritagliando un periodo totale di 5 anni dal 13 dicembre 2019 fino al 13 dicembre 2024. Qui sotto abbiamo riportato uno dei due dataset:

	Data	Ultimo	Apertura	Massimo	Minimo	Vol.	Var. %
0	13.12.2024	1,0472	1,0474	1,0481	1,0453	NaN	0,04%
1	12.12.2024	1,0467	1,0496	1,0531	1,0463	NaN	-0,26%
2	11.12.2024	1,0494	1,0528	1,0540	1,0480	NaN	-0,30%
3	10.12.2024	1,0526	1,0555	1,0568	1,0499	NaN	-0,25%
4	09.12.2024	1,0552	1,0564	1,0595	1,0532	NaN	-0,15%
...	...	...	...	...	...	...	...
1301	19.12.2019	1,1120	1,1113	1,1145	1,1107	NaN	0,08%
1302	18.12.2019	1,1111	1,1150	1,1156	1,1110	NaN	-0,34%
1303	17.12.2019	1,1149	1,1143	1,1176	1,1128	NaN	0,06%
1304	16.12.2019	1,1142	1,1122	1,1159	1,1122	NaN	0,21%
1305	13.12.2019	1,1119	1,1129	1,1200	1,1111	NaN	-0,08%

1306 rows × 7 columns

Come possiamo notare dalla tabella di fianco, le variabili estratte riportano le date e i valori del massimo, minimo, apertura, chiusura della giornata.
Inoltre sono presenti due ulteriori variabili: Volume che presenta esclusivamente valori nulli e la variazione percentuale giornaliera.

In base ai nostri obiettivi abbiamo immediatamente modificato i dataset eliminando le colonne superflue e mantenendo solamente le due variabili necessarie per addestrare il nostro modello ovvero “Data” e “Ultimo” le quali necessitano comunque di piccole modifiche di forma, infatti abbiamo convertito la colonna “Data” in formato datetime e convertito i dati di chiusura in valori numerici di tipo float sostituendo la virgola con il punto per essere riconoscibili al software.
Infine abbiamo impostato la colonna “Data” come indice e generato un plot degli andamenti dei due dataset grazie alla libreria *Matplotlib* già utilizzata nel corso:

Data una prima introduzione generale ed una breve spiegazione del processo di data cleaning, possiamo passare alla spiegazione dei singoli modelli e delle scelte che abbiamo compiuto durante l'implementazione degli stessi per garantire una maggiore efficienza delle previsioni.

Innanzitutto è stato necessario scaricare attraverso il comando *pip install* le principali librerie che ci hanno permesso di sfruttare i comandi necessari per la generazione delle reti neurali elencate. Per quanto riguarda il primo modello LSTM abbiamo richiamato le seguenti librerie:

```
from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, LSTM
```

Come possiamo notare nella prima riga di codice abbiamo importato uno strumento fondamentale della libreria *sklearn "MinMaxScaler"* che ci servirà per la normalizzazione dei dati. Questo ci permetterà di trasformare i valori in un intervallo predefinito [0,1], mantenendo le proporzioni originali e migliorando l'efficienza degli algoritmi di apprendimento. Questo procedimento potrebbe risultare un problema qualora il dataset scelto abbia una distribuzione diversa rispetto alla normale, tuttavia per necessità tecniche consideriamo i due dataset approssimabili ad una normale nonostante i valori outliers non abbiano un ruolo marginale nella serie storica considerata. Bisogna inoltre evidenziare un problema fondamentale che potrebbe essere generato dallo strumento di normalizzazione utilizzato, infatti quando utilizziamo un *MInMaxScaler* su tutto il dataset prima di separare i dati di addestramento e di test, introduciamo un potenziale problema di *data leakage* ovvero "perdita di dati" dal test verso il train. In altre parole, stiamo consentendo al modello di visualizzare le informazioni contenute nel test set ancor prima dell'addestramento, generando di conseguenza una distorsione.
Quindi ai fini della nostra analisi sarà necessario compiere un passaggio aggiuntivo di separazione dei dati nel training set e test set prima di effettuare lo Scaler e successivamente fare il fit dello scaler solo sui dati di training e non sui dati di test e per quest'ultimi applicare semplicemente la formula di scalatura basata sui parametri iniziali delle feature impostate nel range [0,1]:

```
scaled_train = scaler.fit_transform(train_df[['Ultimo']])
Scaled_test = scaler.transform(test_df[['Ultimo']])
```

Solo al termine del processo predittivo li convertiremo nella scala originaria per renderli visualizzabili graficamente.

Dopo aver normalizzato i dati abbiamo dovuto generare due array che contenessero:

1. Le sequenza di input (*x_train, x_test)* dove ogni sequenza è composta da 60 valori consecutivi, che rappresentano i prezzi di chiusura normalizzati nei 60 giorni precedenti. Abbiamo scelto questo numero effettuando un tuning dei parametri principalmente

perché nel contesto delle serie temporali finanziarie 60 giorni corrispondono all'incirca ad un trimestre completo considerando i valori di chiusura solo nei giorni lavorativi come accade nel mercato reale, ciò permette quindi di cogliere trend e pattern stagionali di breve/medio termine.

2. I valori target (*y_train, y_test)* dove per ogni sequenza di 60 valori, il target è il prezzo successivo ovvero il giorno successivo nella serie storica quindi il 61esimo.

Attraverso un banale ciclo *for* possiamo quindi generare i nostri array di sequenze di training e di target di training necessari al nostro modello LSTM dal momento che questo modello lavora con dati sequenziali. Per questo motivo, invece di usare singoli valori come input, il modello deve ricevere sequenze di 60 giorni di dati passati per fare una previsione sul giorno successivo ed iterare questo algoritmo per tutta la serie storica analizzata.

Passiamo adesso alla creazione del modello LSTM resa possibile grazie all'importazione direttamente dalla libreria *keras* di tre strumenti fondamentali: *sequential* utile per l'addestramento delle reti neurali strato dopo strato dove ogni livello presenta un singolo input e un singolo output, "*Dense"* e *"LSTM"* che ci serviranno per definire, compilare e addestrare il nostro modello.

```
model = Sequential([
        LSTM(128, return_sequences=True, input_shape=(x_train.shape[1], 1)),
        LSTM(64, return_sequences=False),
        Dense(25),
        Dense(1)
        ])
```

Abbiamo riportato e implementato una conformazione classica del modello di Deep Learning LSTM che prevede una struttura sequenziale iniziale di due layer Encoder-Decoder. Il primo, composto da 128 neuroni, cattura le dipendenza di lungo termine della serie e restituisce una sequenza di output a ogni step temporale, infatti *input_shape=(x_train.shape[1], 1))* è il numero di timesteps (60 giorni) di cui il valore 1 indica una sola feature (il prezzo di chiusura). Mentre il secondo, composto a sua volta da 64 neuroni riceve l'output del primo LSTM e restituisce un unico valore che verrà poi processato dagli ultimi due layer densi *fully connected,* di cui il primo composto da 25 neuroni elabora le informazioni ricevute e le invia direttamente al layer finale che restituisce il valore predetto del giorno successivo. Questo layer cercherà di bilanciare i dati all'interno del modello al fine di ottenere un dato conclusivo che sia il più attendibile possibile e il meno soggetto a bias (anche se, come anticipato all'inizio, dipende fortemente dalla distribuzione del titolo e nel nostro caso della valuta osservata). Bisogna inoltre ricordare che la libreria keras imposta funzioni di attivazione predefinite, come in questo caso, utilizzando la funzione *tanh,* utile soprattutto per mantenere i valori tra -1 e 1, stabilizzando l'apprendimento. Il modello inoltre è stato compilato attraverso l'ottimizzatore *Adam* che andrà ad aggiornare i pesi durante il modello, e inserendo come loss function il mean square error, ovvero la funzione più comune per le regressioni, che è stata ampiamente approfondita nel corso svolto.

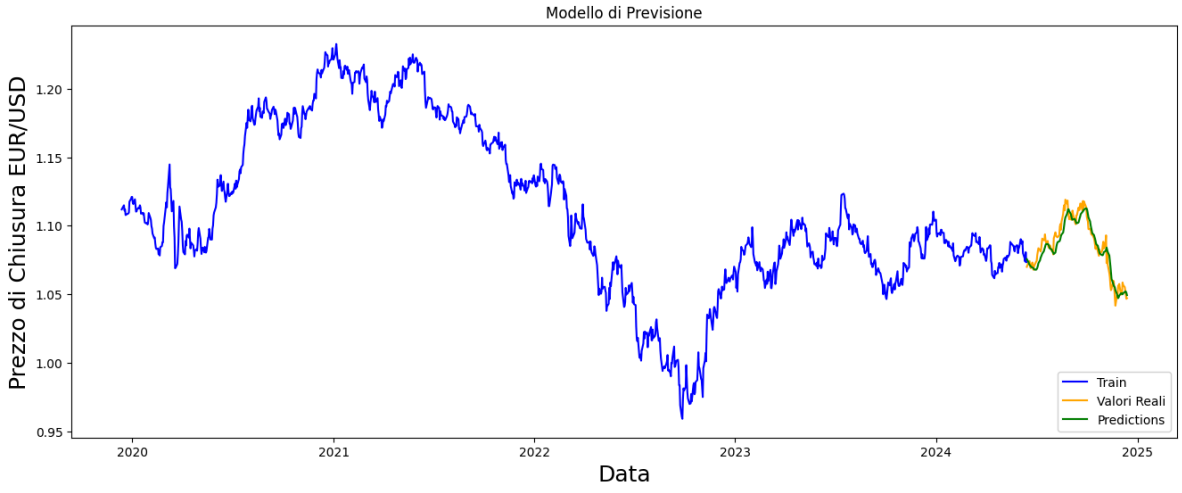
Il primo tentativo di addestramento effettuato è stato impostato nel seguente modo:

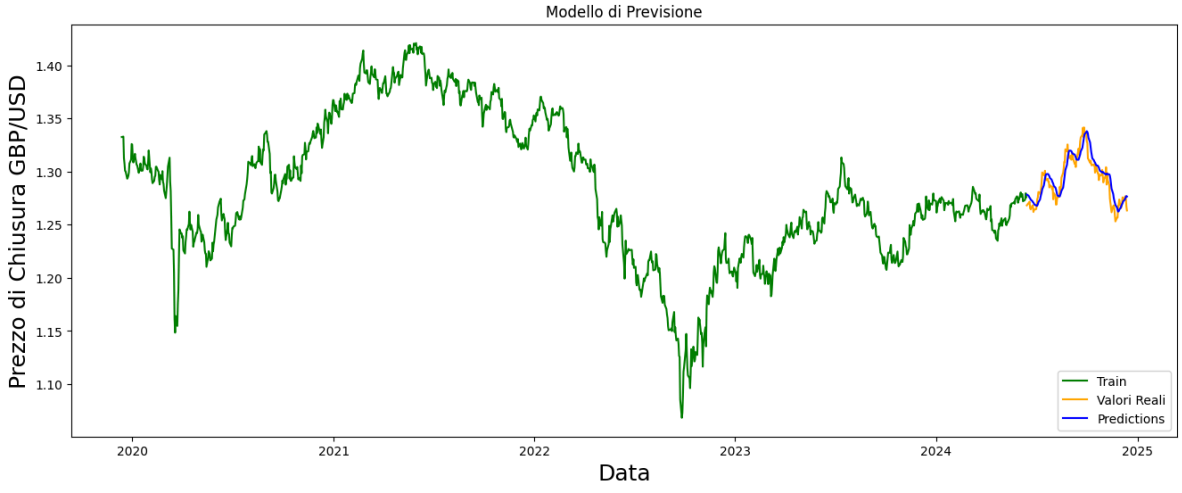
```
model.fit(x_train, y_train, batch_size=1, epochs=1)
```

Il numero di epoche indica quante volte l'intero set di training, suddiviso in batch, viene passato in addestramento al modello. I batch rappresentano porzioni del set di addestramento che vengono passate, un'interazione alla volta, all'interno del modello. Una volta che tutti i batch sono stati elaborati, abbiamo passato al modello l'intero set di training e abbiamo completato un'epoca.
All'addestramento segue il processo che porterà alla creazione delle predizioni attraverso il comando predict che prende in input x_test già ridimensionato con un semplice reshape:
`x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))`
Abbiamo bisogno di ridimensionare i dati in quanto il modello LSTM si aspetta dei dati tridimensionali di cui la prima dimensione è definita dal numero dei campioni "x_test.shape[0]", Il numero di timesteps: "x_test.shape[1]" ed infine il numero di features: 1.

```
predictions = model.predict(x_test)
Predictions = scaler.inverse_transform(predictions)
```

Attraverso la variabile Predictions, che è stata riportata nella forma originaria attraverso lo scaler.inverse, siamo stati finalmente in grado di realizzare la prima predizione per entrambe le serie storiche analizzate, realizzando un grafico che raffigura la porzione di training set e test set predetta (già definite nella fase di pre-elaborazione attraverso la creazione di due dataset complementari con proporzione 90/10 tra training e test set):





Andare a commentare visivamente i grafici ottenuti risulta poco efficace ai fini delle nostre analisi di performance, pertanto è necessario affidarsi alle metriche che abbiamo elencato all'inizio del nostro documento importando la libreria: “`sklearn.metrics`”.

```
EUR_USD
RMSE: 0.008167479659334127
MAE: 0.006518994913756391
MSE: 6.670772398563671e-05
R^2: 0.7291743505692445
GBP_USD
RMSE: 0.011005214179799414
MAE: 0.008419315182343687
MSE: 0.0001211147391432581
R^2: 0.7376095720746615
```

Come possiamo notare, le performance mostrano risultati abbastanza promettenti,con un buon livello di accuratezza nella previsione. Per Euro Dollaro l’RMSE di 0.00817 e un R^2 di 0.7291 indicano che il modello riesce a catturare gran parte della variabilità del prezzo, con errori relativamente bassi ed anche il valore del Mean Absolute Error conferma che il modello ha una discreta capacità di generalizzazione.
Per GBP_USD, le metriche sono leggermente peggiori, con un RMSE ed un MAE più alti rispettivamente: (0.011) e (0.00841) un R^2 inferiore (0.7376), suggerendo di fatto che il modello ha più difficoltà a catturare i pattern di questa serie temporale.
In generale quindi possiamo notare come il modello si comporti meglio su EUR_USD rispetto a GBP_USD, il che potrebbe essere dovuto a una maggiore stabilità della serie EUR_USD.
Tuttavia, il livello di accuratezza ottenuto potrebbe essere ulteriormente migliorato con modifiche mirate dei parametri e delle features, pertanto la seconda parte del nostro progetto è stata totalmente dedicata all’ottimizzazione delle performance attraverso un tuning dei parametri per ogni modello.
Il primo passo è stato quello di impostare uno strumento fondamentale per prevenire overfitting e migliorare la generalizzazione del modello, ovvero il Dropout. Abbiamo deciso di inserirlo tra i due layer Encoder-Decoder con un rate relativamente basso, pari a 0.2, ciò significa che il 20% dei neuroni nel layer specificato verrà disattivato casualmente a ogni iterazione di addestramento. Tuttavia le performance dei modelli sono sorprendentemente calate in ogni metrica di qualche punto percentuale ed abbiamo dunque ripristinato i valori di partenza.
La nostra analisi si è dunque spostata sulla compilazione del modello ed abbiamo deciso di sostituire sia la loss function dal mse ad una funzione più robusta: 'huber', sia l’ottimizzatore ‘adam’ con ‘RMSprop’ per verificare se si potesse migliorare la convergenza, ma per entrambi gli strumenti abbiamo ricevuto solo risposte negative in termini di errore di performance quindi anche in questo caso siamo tornati ai parametri di partenza.
Il primo miglioramento significativo l’abbiamo ottenuto aumentando il numero di epoche e di batch nella fase di addestramento e dopo vari tentativi di tuning abbiamo individuato i valori ottimali per entrambi i parametri pari a 50 epoche con un batch size pari a 32.
E per bilanciare la complessità computazionale raggiunta a causa dell’aumento del numero di epoche e batch e migliorare di conseguenza l’efficienza del modello, abbiamo voluto inserire

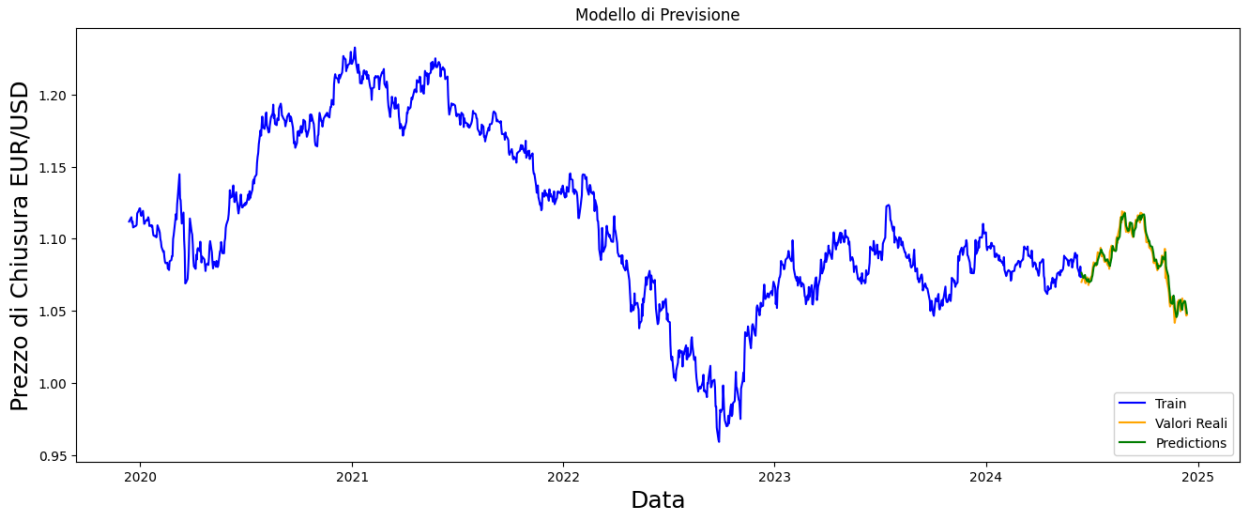
una tecnica chiamata ‘EarlyStop’ che interrompe automaticamente l’addestramento se il modello smette di migliorare durante la fase di addestramento, evitando overfitting e spreco di risorse computazionali:

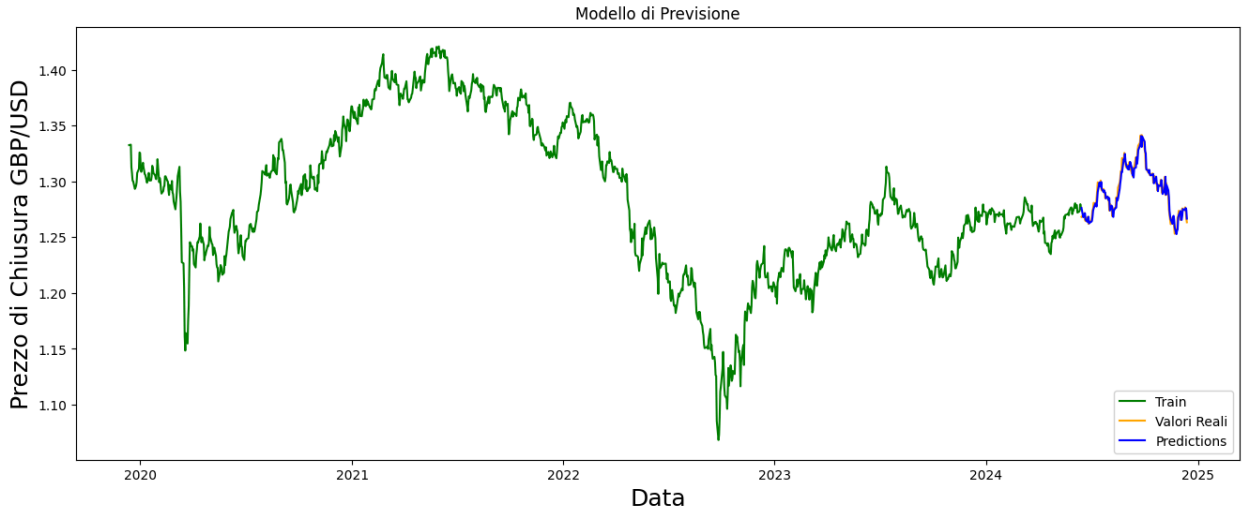
early_stop = EarlyStopping(monitor='loss', patience=5, restore_best_weights=True)

In questo caso, il criterio utilizzato per decidere quando interrompere l'addestramento è la loss, ovvero l'errore calcolato sul training set. La patience è impostata a 5, il che significa che se la loss non migliora per 5 epoche consecutive, il training viene interrotto automaticamente e prende in considerazione, grazie all’ultimo parametro inserito, solo il peso migliore tra le epoche considerate.
I risultati ottenuti dalla combinazione di questi parametri sono stati sorprendenti:

```
EUR_USD
RMSE: 0.004492002073032218
MAE: 0.003585077029512133
MSE: 2.0178082624125737e-05
R^2: 0.9508946591754173
GBP_USD
RMSE: 0.005473932465093496
MAE: 0.0043547184281676685
MSE: 2.9963936632404554e-05
R^2: 0.9350842827972848
```

Anche in questo caso possiamo osservare una differenza di performance, seppur sottile, tra i due titoli valutari. Il modello mostra risultati leggermente migliori su EUR_USD, con un RMSE di 0.00449 e un R^2 di 0.9509, rispetto a GBP_USD, dove l'RMSE è di 0.00547 e l'R^2 è di 0.9351. Questo suggerisce, come già anticipato, che il modello riesce a catturare meglio le dinamiche di EUR_USD rispetto a GBP_USD.





XGBoost

Passiamo adesso all'analisi del secondo modello XGBoost che, come anticipato all'inizio, si basa sul concetto di Gradient Boosting, un metodo che combina più alberi decisionali per migliorare la capacità predittiva del modello in maniera simile ai metodi di boosting che abbiamo approfondito nel corso, dove ogni modello (in questo caso il decision tree) viene costruito in modo sequenziale in base agli errori del modello precedente.
In questa versione di ensemble learning, invece di correggere gli errori modificando i pesi dei dati, si utilizza la discesa del gradiente per ottimizzare la funzione di perdita e l'XGBoost rappresenta una implementazione ottimizzata del Gradient Boosting che migliora le prestazioni e l'efficienza del modello attraverso due caratteristiche fondamentali:

1. Parallelizzazione: suddivide i dati in più blocchi ed esegue calcoli paralleli, riducendo i tempi di addestramento.
2. Regolarizzazione: aiuta a ridurre la complessità del modello ed evitare l'overfitting.

Essendo inoltre un modello di apprendimento supervisionato e non una rete neurale ricorrente come LSTM, non è in grado di lavorare direttamente con dati sequenziali quindi dobbiamo trasformare la serie temporale in un problema di regressione, utilizzando i valori passati come feature:

```
data_xgb = data1.copy()
data_xgb['Lag_1'] = data_xgb['Ultimo'].shift(1)
data_xgb['Lag_7'] = data_xgb['Ultimo'].shift(7)
data_xgb['Lag_30'] = data_xgb['Ultimo'].shift(30)
```

Il primo passaggio in assoluto è stato quello di prendere la serie temporale originale (nel nostro caso data1 corrisponde a EUR_USD) che contiene solo la colonna 'Ultimo' con i prezzi di chiusura e creare nuove colonne che contengono i valori passati del prezzo:

- Lag_1 = valore del giorno precedente

- Lag_7 = valore di 7 giorni fa
- Lag_30 = valore di 30 giorni fa

Ora il dataset non è più una serie temporale, bensì una tabella con feature e target definiti dal seguente codice:

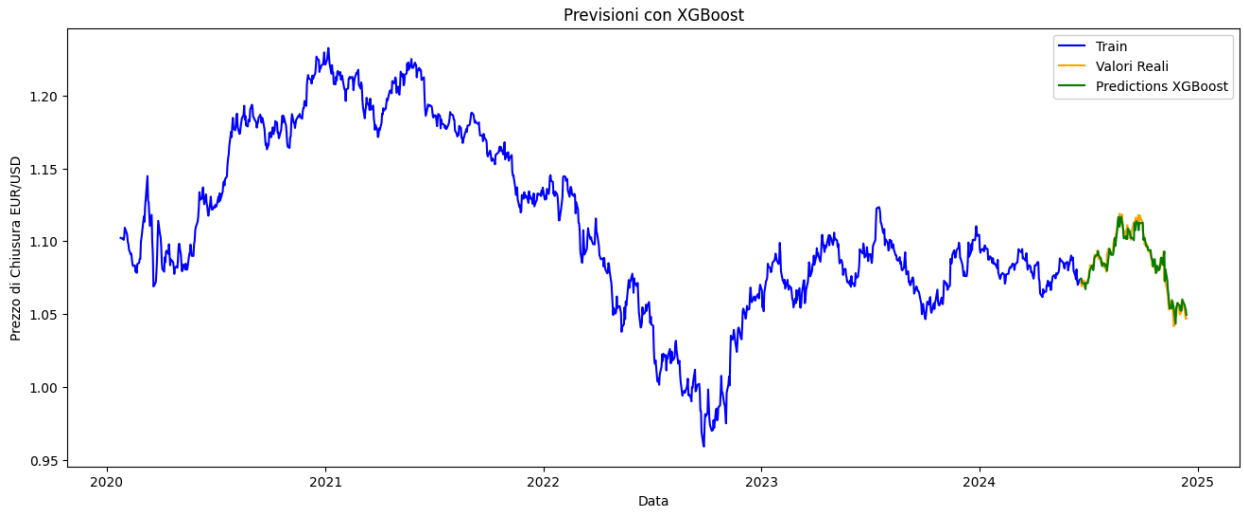
```
x_train_xgb = train_xgb.drop(columns=['Ultimo'])
y_train_xgb = train_xgb['Ultimo']
x_test_xgb = test_xgb.drop(columns=['Ultimo'])
y_test_xgb1 = test_xgb['Ultimo']
```

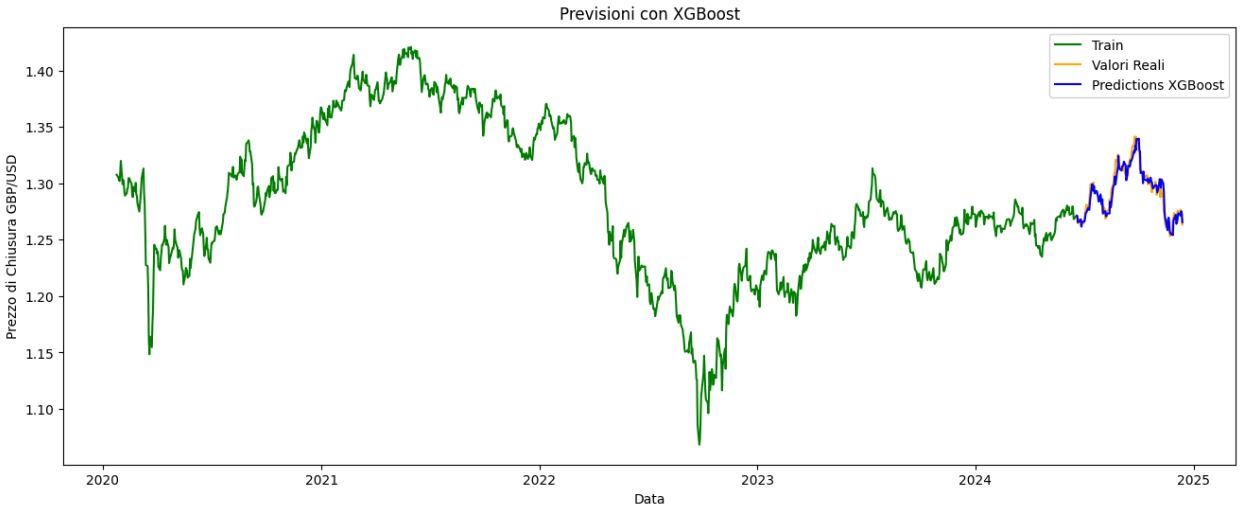
Andiamo quindi ad implementare il nostro modello:

```
model_xgb =
xgb.XGBRegressor(objective='reg:squarederror',n_estimators=100,
learning_rate=0.05)

model_xgb.fit(x_train_xgb, y_train_xgb)
```

Prendendo in input i dati di addestramento '`x_train_xgb, y_train_xgb`', XGBoost costruisce una serie di alberi decisionali, ciascuno migliorando le previsioni dell'albero precedente, impostando come parametri 100 alberi decisionali con un learning rate pari a 0.05 ovvero il tasso di apprendimento che determina quanto velocemente il modello apprende dalle iterazioni precedenti e il valore 0.05 significa che ogni nuovo albero contribuisce solo per il 5% alla previsione finale. Di fatto dopo numerosi tentativi di tuning siamo giunti a questo equilibrio tra tasso di apprendimento e numero di alberi decisionali e ciò rappresenta un buon compromesso per evitare l'overfitting e allo stesso tempo rendere il modello efficiente da un punto di vista computazionale.





Riporteremo qui le metriche finali di performance per fare un confronto diretto con la rete neurale ricorrente:

```
EUR_USD
RMSE: 0.00461224559346158
MAE: 0.0034774828448891713
MSE: 2.1272809414405762e-05
R^2: 0.9489168568989249

GBP_USD
RMSE: 0.006340694904695475
MAE: 0.004855062934756278
MSE: 4.020441187443116e-05
R^2: 0.9127266017796535
```

Possiamo quindi osservare una similitudine nei risultati ottenuti dai due modelli implementati. Tuttavia, la rete neurale ha mostrato una leggera superiorità in termini di prestazioni per entrambi i dataset, registrando un errore inferiore di qualche punto percentuale rispetto a XGBoost.
La differenza sostanziale emersa durante l'intero processo di implementazione, è stata, senza dubbio, il tempo di addestramento dei due modelli.
In particolare, il modello di Gradient Boosting è stato scelto per la sua rapidità, con un tempo di addestramento che si attesta intorno a ~0,5 secondi. Al contrario, la rete neurale ricorrente, a causa del numero di epoche e della dimensione del batch impostati, richiede un tempo significativamente maggiore. Grazie alla presenza del parametro Early Stop, il processo di addestramento si interrompe in anticipo non appena le prestazioni sul set di validazione smettono di migliorare. Di conseguenza, la media di epoche processate si attesta a circa 40 e nonostante ciò il tempo di addestramento non è mai sceso sotto i due minuti.
Possiamo quindi concludere che, sebbene ogni modello presenti i propri pregi e difetti, è indispensabile conoscere in anticipo le caratteristiche di ogni modello in modo tale da

selezionare l’algoritmo più adatto in base alle esigenze specifiche del problema. Tuttavia il confronto diretto di modelli totalmente differenti, aiuta a comprendere meglio le peculiarità di ciascun approccio e a valutare a posteriori, in base ai dati ottenuti, il miglior modello in funzione delle prestazioni, dell’interpretabilità e della complessità computazionale.