UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Progetto Big Data Engineering

Anno Accademico 2022-2023

Simone D'Orta

M63001283

Indice

Traccia

La previsione dei prezzi delle azioni utilizzando i big data e l'intelligenza artificiale (AI) è emersa come un potente strumento per investitori, trader e analisti finanziari. Si tratta di analizzare grandi quantità di dati storici e in tempo reale, tra cui tendenze di mercato, indicatori economici, performance aziendali e articoli di cronaca, per identificare modelli e prevedere i prezzi futuri delle azioni. L'uso di algoritmi avanzati di apprendimento
e modelli predittivi aiuta gli investitori a prendere decisioni informate, a minimizzare i rischi e a massimizzare i rendimenti. In questo modo, i big data e l'IA stanno trasformando il panorama degli investimenti tradizionali, offrendo agli investitori nuove opportunità di sfruttare le tecnologie emergenti per ottenere migliori performance di mercato.

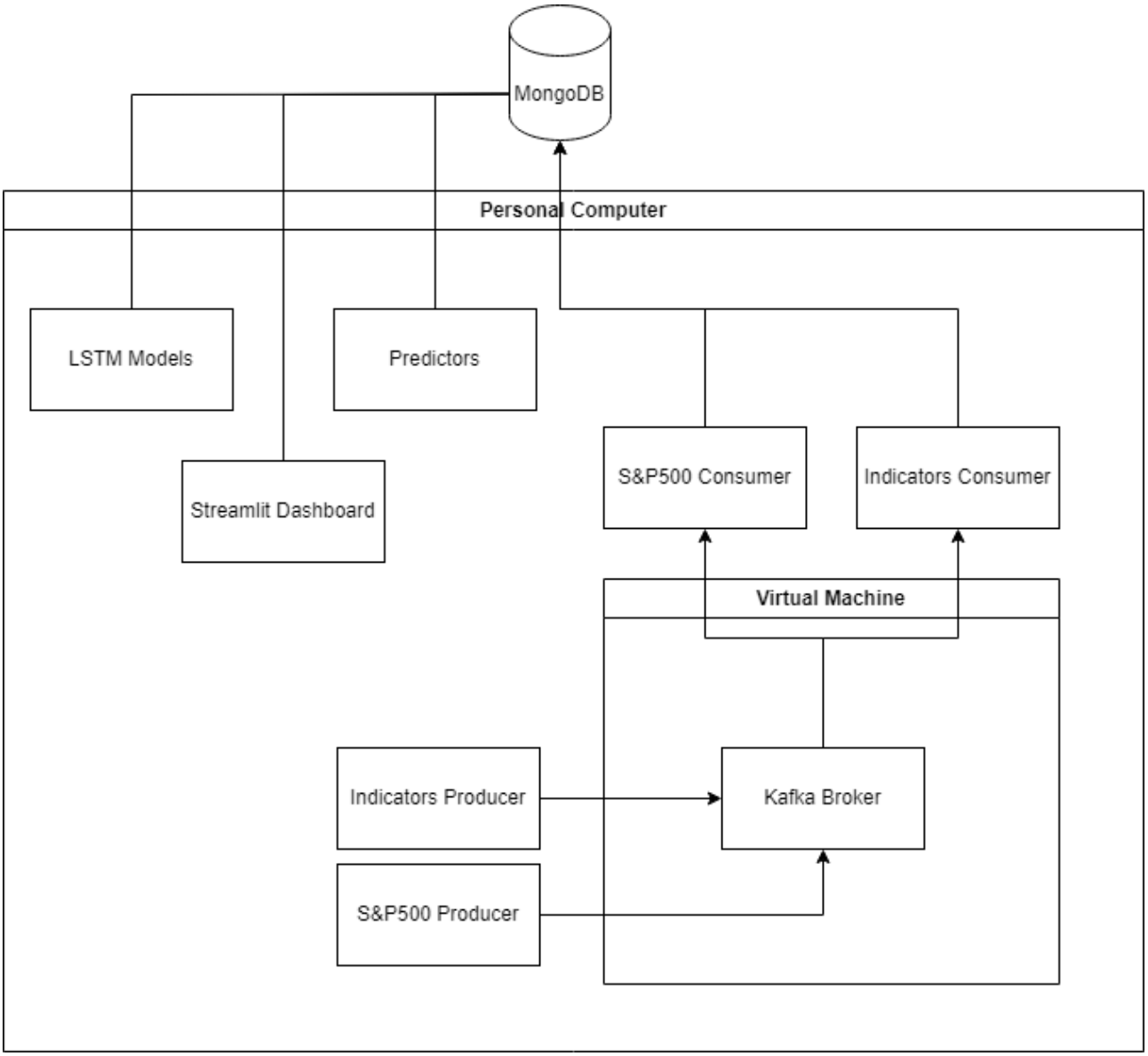
Capitolo 1: Architettura e setup

Nel progetto in questione, si è deciso di utilizzare strumenti avanzati per la gestione di big data e reti neurali al fine di prevedere il prezzo dell'indice azionario S&P500.

Nello specifico, si è optato per l'utilizzo di Kafka per gestire lo streaming continuo dei dati relativi al prezzo dell'indice azionario e agli indicatori economici. Tali dati sono stati salvati e gestiti tramite il database NoSQL MongoDB. Infine, i dati salvati vengono utilizzati per addestrare un modello LSTM in grado di predire i prezzi fino a tre giorni di distanza.

1.1 Architettura

Come già accennato in precedenza, per gestire correttamente il flusso di dati relativi all'indice S&P500 e agli indicatori economici, si è scelto di utilizzare Kafka. Tale scelta è stata motivata anche dalla possibilità futura di estendere l'analisi ad altre azioni o indici. Considerando l'enorme mole di dati che un sistema simile potrebbe dover gestire nel tempo e l’importanza di effettuare analytics sugli stessi, si è adottato un database NoSQL, nello specifico MongoDB. Di seguito viene presentata una bozza dell’architettura utilizzata.



1.2 Setup

Kafka è stato installato su una macchina virtuale con sistema operativo Ubuntu 22.04. Di seguito viene descritta la procedura seguita per eseguire correttamente la configurazione . Dopo aver installato Kafka sulla macchina virtuale Linux è stato identificato l'hostname eseguendo il comando "hostname" nella shell:

$ hostname

simone-VirtualBox.localdomain

Successivamente, sono state apportate modifiche alle proprietà del broker Kafka (kafka.properties). Sono state impostate le seguenti configurazioni:

listeners=PLAINTEXT://0.0.0.0:9092
advertised.listeners=PLAINTEXT://simone-VirtualBox.localdomain:9092

Nel sistema operativo Windows (host), è stato modificato il file hosts di Windows per aggiungere l'hostname della macchina virtuale ottenuto nel passaggio precedente e assegnare l'indirizzo IP 127.0.0.1 a tale hostname:

127.0.0.1 simone-VirtualBox.localdomain

In VirtualBox, sono state aperte le impostazioni di rete della macchina virtuale e sono state aggiunte nuove regole di port forwarding per la porta del broker Kafka, ovvero la porta 9092 (impostazione predefinita, se non è stata modificata).
Con questa configurazione, è stato possibile connettersi ad Apache Kafka installato sulla macchina virtuale Linux da una macchina virtuale Windows e pubblicare i messaggi. Questo setup consente anche di connettere più macchine virtuali guest per emulare un cluster multinodo.
Sono stati creati i seguenti topic con un replication-factor di 1 e 1 partizione ciascuno: "HOUST", "2INDPRO", "PCE", "PI", "PPIACO", "UNRATE", "sp500", "sp500_pred". Dunque, è stato creato un topic per ogni indice utilizzato ed uno ulteriore per gestire le predizioni.
Per quanto riguarda MongoDB Atlas, è stato creato un cluster su MongoDB Atlas, una piattaforma di database cloud che semplifica la creazione e la gestione del database. Essendo il software creato ancora un prototipo, è stato permesso l’accesso al cluster da qualsiasi ip. Inoltre, è stato creato un database chiamato "Stock", che comprende le seguenti collezioni: "HOUST", "2INDPRO", "PCE", "PI", "PPIACO", "UNRATE", "sp500", "sp500_pred".

Capitolo 2: Produttori e consumatori

Nel seguente capitolo sono mostrati tutti i produttori e consumatori utilizzati per recuperare i dati relativi all'indice S&P500 e agli indicatori economici.

2.1 S&P500 produttore e consumatore

Il codice relativo al produttore S&P500 utilizza il modulo yfinance per scaricare i dati relativi all'indice S&P 500 a partire dalla data '1982-04-20' e li memorizza el dataframe data. Per ogni riga del dataframe, i dati vengono convertiti in formato JSON utilizzando il metodo to_json() e vengono inviati al topic Kafka specificato utilizzando il metodo send() del produttore. Dopo l'invio di ogni mille messaggi, il produttore Kafka viene "flushato", per assicurarsi che i messaggi siano stati inviati correttamente.

```
# Initialize Kafka producer
producer = KafkaProducer(bootstrap_servers='simone-VirtualBox')

# Retrieve S&P 500 data
data = yf.download("^GSPC", start="1982-04-20")  # before this date the opening price is 0

# Reset index to numeric index
data.reset_index(inplace=True)

# Convert the DataFrame rows to JSON and send them to Kafka
topic = 'sp500'
count = 0
for index, row in data.iterrows():
    message = row.to_json()  # Convert the row to JSON format
    producer.send(topic, value=message.encode('utf-8'))
    count = count + 1
    if count % 1000 == 0:
        producer.flush()
producer.flush()

producer.close()
```

Il consumatore S&P500 legge i messaggi dal topic 'sp500'. I messaggi sono convertiti da formato JSON e salvati nel database MongoDB chiamato 'Stock' nella collezione 'sp500'.

```
uri = "mongodb+srv://simone:ciaociao12@cluster0.irvdcqp.mongodb.net/?retryWrites=true&w=majority"
# Create a new client and connect to the server
client = MongoClient(uri)
db = client['Stock']
collection = db['sp500']

consumer = KafkaConsumer(
    'sp500',
    bootstrap_servers=['simone-VirtualBox']
)

# Consume messages
for message in consumer:
    message_value = message.value.decode('utf-8')
    data = json.loads(message_value)

    # Convert the date field back to a datetime object in UTC
    timestamp = int(data['Date']) / 1000
    date = datetime.utcfromtimestamp(timestamp).replace(tzinfo=None)
    data['Date'] = date
    # Insert the document into MongoDB
    collection.insert_one(data)

# Close the Kafka consumer and MongoDB client
consumer.close()
client.close()
```

2.2 Indicatori economici consumatore e produttore

Il codice inizializza l'API Fred e un produttore Kafka, quindi recupera dati storici per diversi indicatori economici da Fred utilizzando il simbolo associato a ciascun indicatore. I dati vengono memorizzati in un dataframe e inviati come messaggi JSON al broker Kafka usando il topic relativo al simbolo.

Fred (Federal Reserve Economic Data) è un database di dati economici gestito dalla Federal Reserve Bank di St. Louis. Fornisce accesso a una vasta gamma di dati macroeconomici, tra cui indicatori del mercato del lavoro, produzione industriale, prezzi, spese dei consumatori e altro ancora. Nel seguente codice è utilizzata l'API che consente agli sviluppatori di accedere e recuperare dati economici per analisi e applicazioni.

```python
# Initialize Fred API
fred = Fred(api_key='a79fc0c8bf4e63effeb304267ec33dfc')
# Initialize Kafka producer
producer = KafkaProducer(bootstrap_servers='simone-VirtualBox')

start_date = '1982-04-20'
# Symbols for the economic indicators
symbols = ['UNRATE', 'INDPRO', 'PPIACO', 'HOUST', 'PI', 'PCE']
# Create an empty DataFrame to store the data
economic_data = {}
# Retrieve data for each economic indicator and add it to the DataFrame
for symbol in symbols:
    data = fred.get_series(symbol, start_date)
    data = data.reset_index()
    data = data.rename(columns={"index": "Date", 0: "Value"})
    economic_data[symbol] = data

    # Convert the DataFrame rows to JSON and send them to Kafka
    topic = symbol
    count = 0
    for index, row in economic_data[symbol].iterrows():
        message = row.to_json()  # Convert the row to JSON format
        producer.send(topic, value=message.encode('utf-8'))
        count = count + 1
        if count % 1000 == 0:
            producer.flush()
    producer.flush()

producer.close()
```

Gli indicatori economici utilizzati sono:

UNRATE: Questo indicatore rappresenta il tasso di disoccupazione negli Stati Uniti, ovvero la percentuale della forza lavoro che è disoccupata.

INDPRO: L'INDPRO, o produzione industriale, indica la produzione totale delle industrie manifatturiere, minerarie e di servizi pubblici negli Stati Uniti. È spesso utilizzato come misura dell'attività economica nel settore manifatturiero.

PPIACO: Il PPIACO, o indice dei prezzi al produttore per tutte le merci, rappresenta la variazione dei prezzi dei beni prodotti negli Stati Uniti. Misura l'inflazione a livello di produttore.

HOUST: Questo indicatore si riferisce al numero di nuove costruzioni residenziali iniziate negli Stati Uniti. È un indicatore di attività nel settore edilizio e immobiliare.

PI: L'indice dei prezzi, noto anche come deflatore del PIL, è una misura dell'inflazione a livello di consumatore. Rappresenta il cambiamento medio dei prezzi di un paniere di beni e servizi acquistati dai consumatori.

PCE: Il PCE, o spese per consumi personali, è una misura delle spese dei consumatori negli Stati Uniti. Include una vasta gamma di categorie di spesa, come alimentari, abbigliamento, abitazione, trasporti e altro ancora. Il PCE viene spesso utilizzato come indicatore chiave della crescita economica e dell'inflazione.

Il consumatore si connette al broker Kafka e consuma i messaggi provenienti dai topic definiti in precedenza. I messaggi ricevuti vengono quindi salvati nel database ‘Stock’ e nella collection relativa allo specifico topic.

```
uri = "mongodb+srv://simone:ciaociao12@cluster0.irvdcqp.mongodb.net/?retryWrites=true&w=majority"
# Create a new client and connect to the server
client = MongoClient(uri)
db = client['Stock']

consumer = KafkaConsumer(
    bootstrap_servers=['simone-VirtualBox']
)

topics = ['UNRATE', 'INDPRO', 'PPIACO', 'HOUST', 'PI', 'PCE', 'GC-F', 'CL-F', 'HG-F']
consumer.subscribe(topics=topics)

# Consume messages
for message in consumer:
    message_value = message.value.decode('utf-8')
    data = json.loads(message_value)
    # Convert the date field back to a datetime object in UTC
    timestamp = int(data['Date']) / 1000
    date = datetime.utcfromtimestamp(timestamp).replace(tzinfo=None)
    # Update the date field in the data dictionary with the BSON date
    data['Date'] = date
    collection = db[message.topic]
    # Insert the document into MongoDB
    collection.insert_one(data)

# Close the Kafka consumer and MongoDB client
consumer.close()
client.close()
```

Capitolo 3: Modelli predittivi

Nel seguente capitolo viene mostrato come è stato possibile addestrare i tre modelli predittivi in grado di prevedere il prezzo delle stock fino a tre giorni di distanza e come sono state gestite e salvate le predizioni effettuate da tali modelli.

3.1 Addestramento modelli

Per addestrare correttamente i modelli in questione, è stato prima necessario recuperare i documenti relativi all'indice S&P500 e agli indicatori economici. Successivamente, questi documenti sono stati convertiti in dataframe e uniti tra loro. Per addestrare un modello LSTM, è stato quindi necessario creare finestre di 128 istanze e associare a ciascuna finestra l'etichetta corretta, nel caso del primo modello è il prezzo di chiusura relativo al giorno successivo all'ultima istanza della finestra.

```
# Fetch sp500
documents = collection.find()
# Convert documents to a list of dictionaries
data = [doc for doc in documents]
# Create a DataFrame from the data
data = pd.DataFrame(data)
data.drop('_id', axis=1, inplace=True)
data.drop('Adj Close', axis=1, inplace=True)

collections = ['UNRATE', 'INDPRO', 'PPIACO', 'HOUST', 'PI', 'PCE']
for x in collections:
    collection = db[x]
    documents = collection.find()
    # Convert documents to a list of dictionaries
    temp = [doc for doc in documents]
    # Create a DataFrame from the data
    temp = pd.DataFrame(temp)
    temp.drop('_id', axis=1, inplace=True)

    # Merge the DataFrames based on year and month
    data = pd.merge(data, temp, left_on=data['Date'].dt.to_period('M'), right_on=temp['Date'].dt.to_period('M'))
    # Forward-fill the missing values
    data[f'{x} Value'] = data['Value'].ffill()
    data = data.drop('Value', axis=1)
    data = data.drop(['key_0', 'Date_y'], axis=1)
    data = data.rename(columns={'Date_x': 'Date'})
```

```
# Extract the 'Close' column
close_column = data['Close']
# Convert the 'Date' column to int
data['Date'] = data['Date'].astype(np.int64)
data.drop('Close', axis=1, inplace=True)

# Normalize the input data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
date_scaler = StandardScaler()
date_scaler.fit(data['Date'].values.reshape(-1, 1))
# Normalize the labels
label_scaler = StandardScaler()
scaled_labels = label_scaler.fit_transform(close_column.values.reshape(-1, 1))

# Set the number of days used for prediction
prediction_days = 128
# Initialize empty lists for training data input and output
x_train = []
y_train = []
# Iterate through the scaled data, starting from the prediction_days index
for x in range(prediction_days, len(scaled_data)):
    # Append the previous 'prediction_days' values to x_train
    x_train.append(scaled_data[x - prediction_days:x])
    # Append the current value to y_train
    y_train.append(scaled_labels[x])
# Convert the x_train and y_train lists to numpy arrays
x_train, y_train = np.array(x_train), np.array(y_train)
# Split the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(x_train, y_train, test_size=0.15, random_state=10)
```

```
# Initialize a sequential model
model = Sequential()
model.add(LSTM(units=16, return_sequences=True, input_shape=(x_train.shape[1], x_train.shape[2])))
model.add(LSTM(units=16, return_sequences=True))
model.add(LSTM(units=16))
model.add(Dense(units=1))

model.summary()
model.compile(optimizer='adam', loss='mse')

# Define the filepath for saving the best model
checkpoint_filepath = 'best_model.h5'
# Create a ModelCheckpoint callback
checkpoint = ModelCheckpoint(checkpoint_filepath, monitor='val_loss', save_best_only=True, mode='min', verbose=2)
# Create an EarlyStopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=20, mode='min', verbose=2)
# Train the LSTM model with the ModelCheckpoint and EarlyStopping callbacks
history = model.fit(x_train, y_train, epochs=100, batch_size=32, validation_split=0.15, callbacks=[checkpoint, early_stopping])

# Loads the weights
model.load_weights(checkpoint_filepath)
# Evaluate the model on the testing set
test_loss = model.evaluate(x_test, y_test)
print("Test Loss:", test_loss)
```

L’unica differenza nel codice per gli altri due modelli in grado di predire il prezzo dei giorni successivi al prossimo, è nella costruzione delle finestre e nell’associazione dell’etichetta alla finestra. Associerò ad ogni finestra, non il valore di chiusura relativo al giorno successivo all'ultima istanza della finestra, ma quello relativo a due o tre giorni dopo.

Modello 2:

```
# I'M PREDICTING THE PRICE FOR 2 DAYS AFTER
# Iterate through the scaled data, starting from the prediction_days index
for x in range(prediction_days, len(scaled_data)-1):
    x_train.append(scaled_data[x - prediction_days:x])
    y_train.append(scaled_labels[x+1])
```

Modello 3:

```
# I'M PREDICTING THE PRICE FOR 3 DAYS AFTER
# Iterate through the scaled data, starting from the prediction_days index
for x in range(prediction_days, len(scaled_data)-2):
    x_train.append(scaled_data[x - prediction_days:x])
    y_train.append(scaled_labels[x+2])
```

In tutti e tre i modelli, oltre ad utilizzare i valori relativi all’S&P500, sono stati utilizzati degli indici economici per cercare di migliorare le prestazioni. L’introduzione di questi indici, non ha portato però a delle differenze sostanziali nelle prestazioni dei modelli.
La struttura della rete neurale e la dimensione della finestra sono il risultato di numerose prove effettuate. Infatti, la precedente struttura della rete e dimensione della finestra sono risultate le più performanti in termini di test loss, in particolare:

Test loss modello 1: 0.00036

Test loss modello 2: 0.00065

Test loss modello 3: 0.00095

3.2 Predizioni

Per predire il prezzo dell'indice S&P500 utilizzando i tre modelli addestrati in precedenza bisogna inanzitutto recuperare i documentati dal database MongoDB ed unirli tra loro. L'unico problema è l'ultimo valore relativo agli indici economici è di Maggio, dunque è necessario replicare questo valore per tutti i giorni successivi:

```
# Fetch sp500
documents = collection.find()
# Convert documents to a list of dictionaries
initial_data = [doc for doc in documents]
# Create a DataFrame from the data
initial_data = pd.DataFrame(initial_data)
initial_data.drop('_id', axis=1, inplace=True)
initial_data.drop('Adj Close', axis=1, inplace=True)

data = initial_data
collections = ['UNRATE', 'INDPRO', 'PPIACO', 'HOUST', 'PI', 'PCE']
for x in collections:
    collection = db[x]
    documents = collection.find()
    # Convert documents to a list of dictionaries
    temp = [doc for doc in documents]
    # Create a DataFrame from the data
    temp = pd.DataFrame(temp)
    temp.drop('_id', axis=1, inplace=True)

    # Merge the DataFrames based on year and month
    data = pd.merge(data, temp, left_on=data['Date'].dt.to_period('M'), right_on=temp['Date'].dt.to_period('M'))
    # Forward-fill the missing values
    data[f'{x} Value'] = data['Value'].ffill()
    data = data.drop('Value', axis=1)
    data = data.drop(['key_0', 'Date_y'], axis=1)
    data = data.rename(columns={'Date_x': 'Date'})
```

```
# UNRATE, INDPRO,...,PCE values stops in MAY so I replicate them.
# Filter the relevant data from initial_data
filtered_initial_data = initial_data[initial_data['Date'] > data["Date"].iloc[-1]]
# Get the last values of the columns to replicate in data
last_values = data.iloc[-1][['UNRATE Value', 'INDPRO Value', 'PPIACO Value', 'HOUST Value', 'PI Value', 'PCE Value']]
# Append the filtered initial_data to data
data = pd.concat([data, filtered_initial_data[['Date', 'High', 'Volume', 'Low', 'Close', 'Open']]], ignore_index=True)
# Fill the NaN values in the additional columns with the last values
data[['UNRATE Value', 'INDPRO Value', 'PPIACO Value', 'HOUST Value', 'PI Value', 'PCE Value']] = \
    data[['UNRATE Value', 'INDPRO Value', 'PPIACO Value', 'HOUST Value', 'PI Value', 'PCE Value']].fillna(last_values)

# Dataframe containing all the prediction and the actual value
df_pred = pd.DataFrame()
df_pred["Date"] = data["Date"]
df_pred["Value"] = data['Close']
# Preparing the dataframe to receive the predictions
# Get the last date from the existing data
last_date = df_pred["Date"].iloc[-1]
# Add three new rows to df_pred
df_pred.loc[len(df_pred)] = [last_date + pd.DateOffset(days=1), np.nan]
df_pred.loc[len(df_pred)] = [last_date + pd.DateOffset(days=2), np.nan]
df_pred.loc[len(df_pred)] = [last_date + pd.DateOffset(days=3), np.nan]
df_pred["Value"].iloc[-3] = np.nan
```

Vengono costruite le finestre da 128 istanze a partire dalle quali vengono predetti i valori per ogni data utilizzando i modelli addestrati in precedenza.

```
# Extract the 'Close' column
close_column = data['Close']
# Convert the 'Date' column to int
data['Date'] = data['Date'].astype(np.int64)
data.drop('Close', axis=1, inplace=True)

scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
date_scaler = StandardScaler()
date_scaler.fit(data['Date'].values.reshape(-1, 1))
# Normalize the labels
label_scaler = StandardScaler()
scaled_labels = label_scaler.fit_transform(close_column.values.reshape(-1, 1))
# Set the number of days used for prediction
prediction_days = 128

for i in range(1, 4):
    x_test = []
    # Iterate through the scaled data, starting from the prediction_days index
    for x in range(prediction_days, len(scaled_data)+1):
        # Append the previous 'prediction_days' values to x_train
        x_test.append(scaled_data[x - prediction_days:x])

    # Convert the x_test and y_test lists to numpy arrays
    x_test = np.array(x_test)
```

```
    # Load the model from the saved weights
    loaded_model = load_model(f'best_model{i}.h5')
    # Perform new predictions on new data
    y_pred = loaded_model.predict(x_test)

    # Denormalize the predicted values
    y_pred_denormalized = label_scaler.inverse_transform(y_pred)
    # Extract the last date for each window in x_test
    last_dates = x_test[:, -1, 0]
    # Denormalize the last_dates using the date_scaler
    denormalized_last_dates = date_scaler.inverse_transform(last_dates.reshape(-1, 1))
    # Convert the denormalized last_dates to datetime and add i day
    last_dates = pd.to_datetime(denormalized_last_dates.flatten()) + pd.DateOffset(days=i)

    # Create the 'temp' DataFrame
    temp = pd.DataFrame({f'Pred{i}': y_pred_denormalized.flatten()})
    # Create a new DataFrame with NaN values
    nan_values = pd.DataFrame({f'Pred{i}': [np.nan] * (127 + i)})
    # Concatenate the new DataFrame with the temp DataFrame
    temp = pd.concat([nan_values, temp], ignore_index=True)
    # Merge 'temp' with 'df_pred' based on their index
    df_pred[f'Pred{i}'] = temp[f'Pred{i}']
```

I diversi valori predetti sono stati salvati in un dataframe, a partire dal quale sono estratte le righe che vengono caricate sul database.

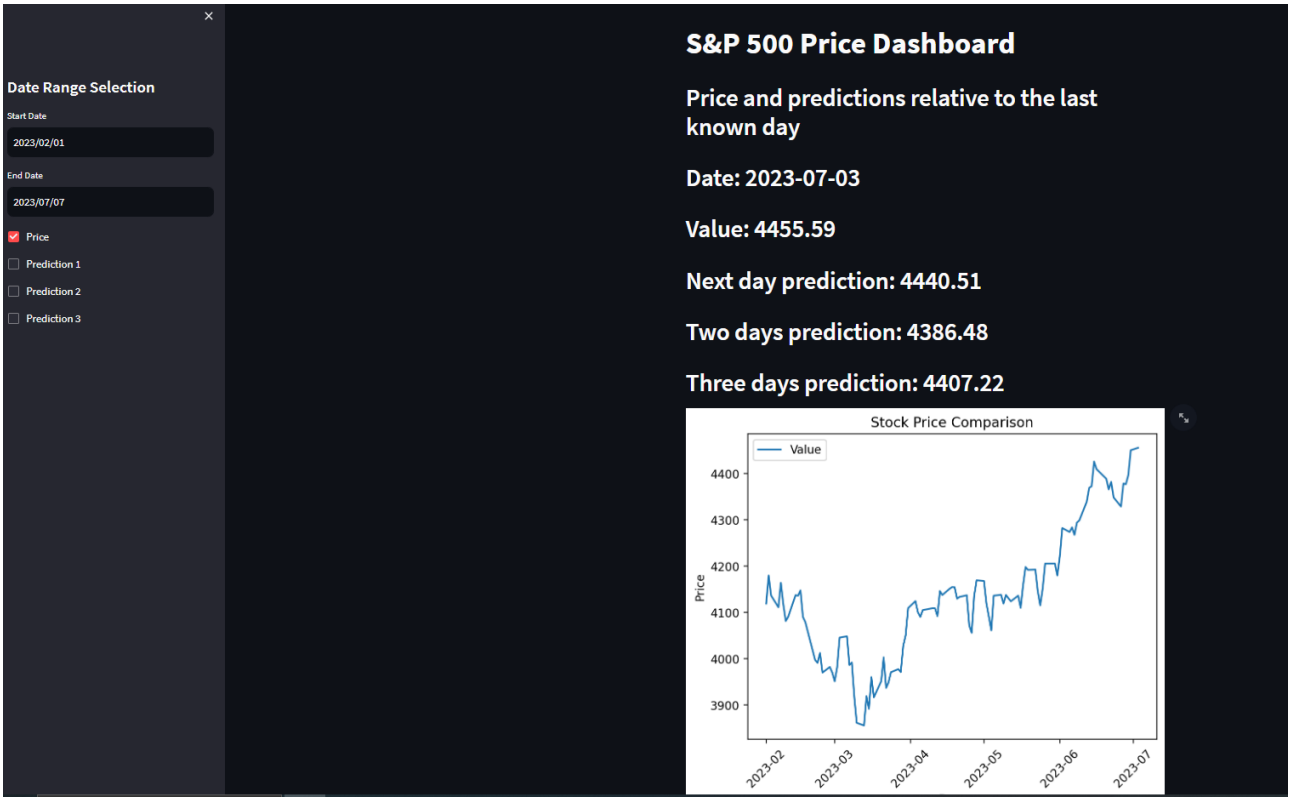
```
# Save predictions in my database
collection = db['sp500_pred']
# Collect documents as a list of dictionaries
documents = df_pred.to_dict(orient='records')
# Insert the documents into the collection
collection.insert_many(documents)
```

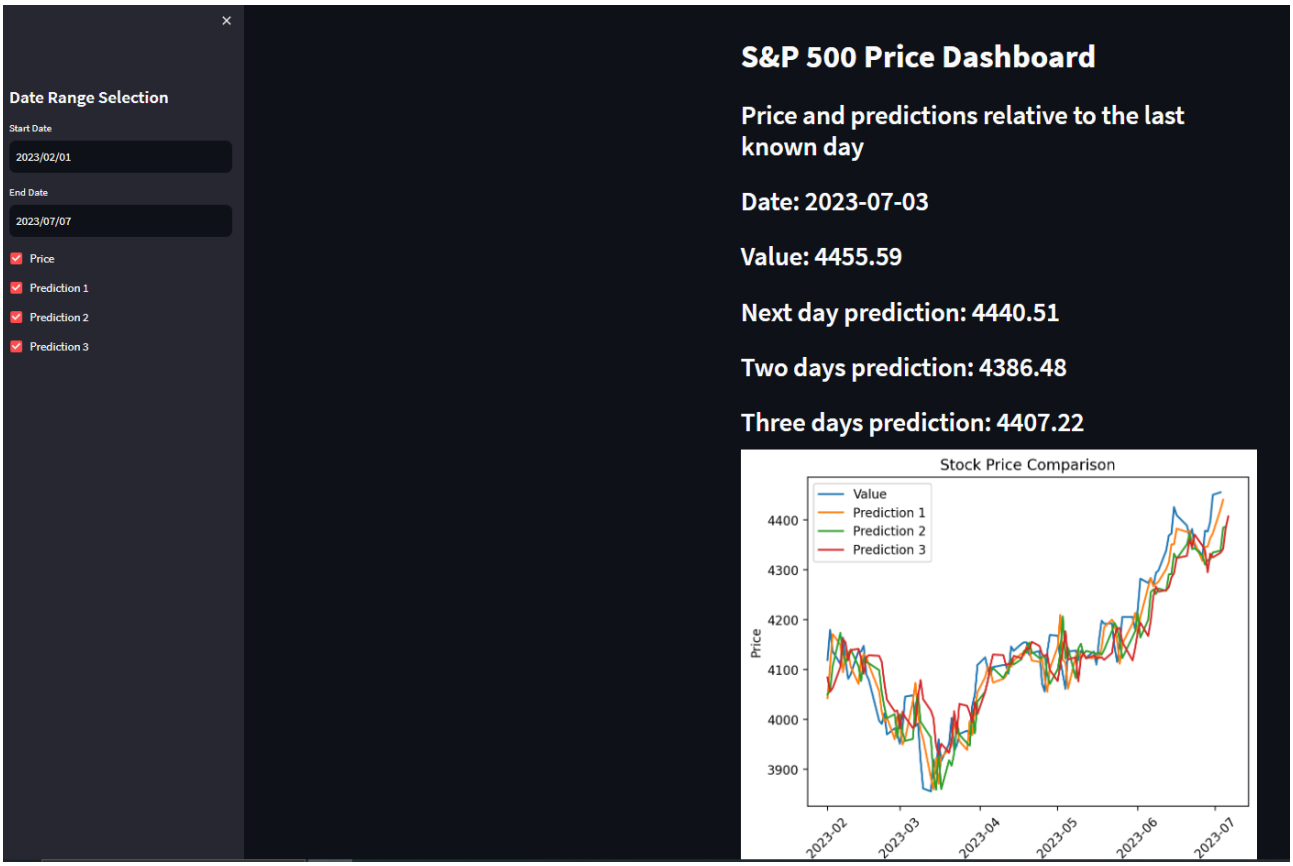
Capitolo 4: Visualizzazione dati

Nel seguente capitolo è descritta la dashboard stramlit utilizzata per visualizzare graficamente e testualmente i valori reali e predetti dell'indice S&P500.

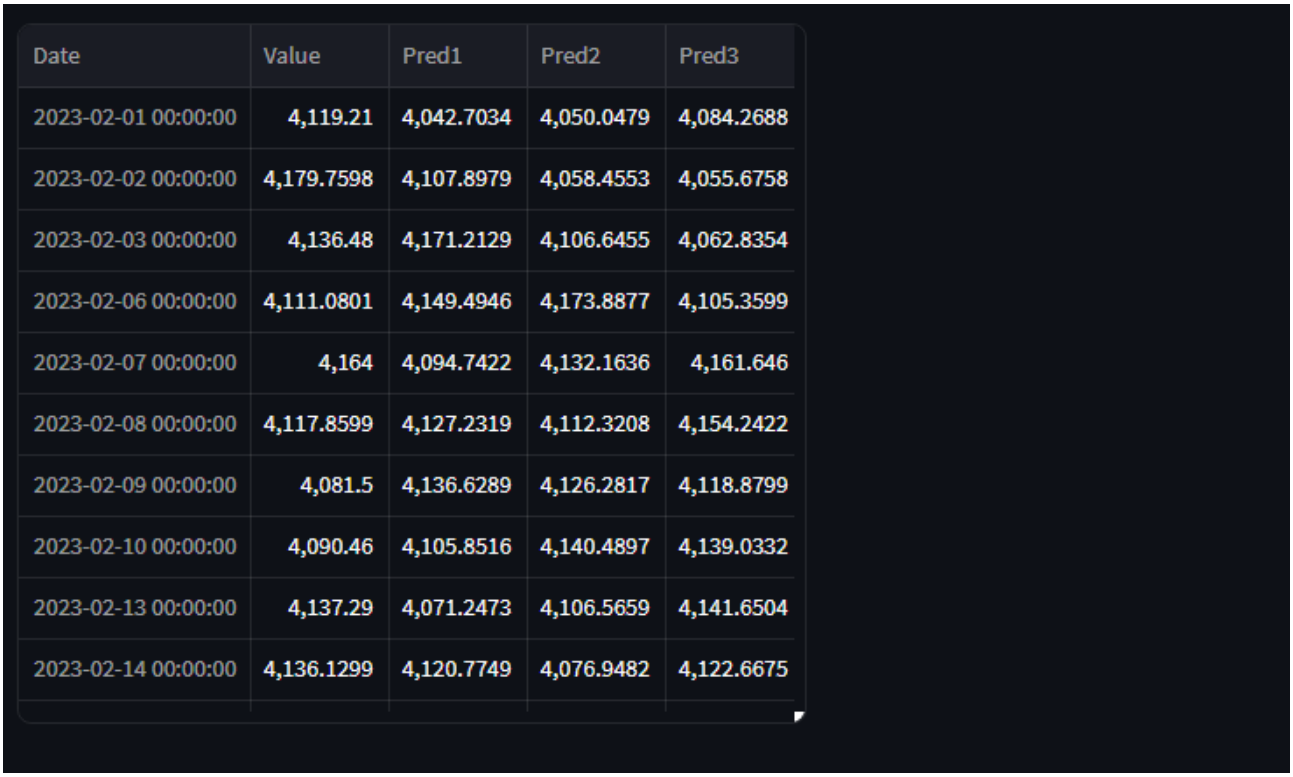
4.1 Streamlit dashboard

La dashboard sviluppata mostra testualmente la data e il prezzo dell'ultima istanza della collection "sp500_pred", mostrando contestualmente anche la predizione per i successivi 3 giorni. Tramite il menu laterale è possibile modificare il grafico che mostra nel tempo il prezzo reale e le tre differenti predizioni.





Infine è anche mostrato il dataset filtrato in base alle date in input in forma tabellare.



Date	Value	Pred1	Pred2	Pred3
2023-02-01 00:00:00	4,119.21	4,042.7034	4,050.0479	4,084.2688
2023-02-02 00:00:00	4,179.7598	4,107.8979	4,058.4553	4,055.6758
2023-02-03 00:00:00	4,136.48	4,171.2129	4,106.6455	4,062.8354
2023-02-06 00:00:00	4,111.0801	4,149.4946	4,173.8877	4,105.3599
2023-02-07 00:00:00	4,164	4,094.7422	4,132.1636	4,161.646
2023-02-08 00:00:00	4,117.8599	4,127.2319	4,112.3208	4,154.2422
2023-02-09 00:00:00	4,081.5	4,136.6289	4,126.2817	4,118.8799
2023-02-10 00:00:00	4,090.46	4,105.8516	4,140.4897	4,139.0332
2023-02-13 00:00:00	4,137.29	4,071.2473	4,106.5659	4,141.6504
2023-02-14 00:00:00	4,136.1299	4,120.7749	4,076.9482	4,122.6675

Il codice sviluppato per creare questa dashboard è il seguente:

```python
# Connect to MongoDB
uri = "mongodb+srv://simone:ciaociao12@cluster0.irvdcqp.mongodb.net/?retryWrites=true&w=majority"
client = pymongo.MongoClient(uri)
db = client['Stock']
collection = db['sp500_pred']


@st.cache_data
def load_data():
    # Retrieve all documents from the MongoDB collection
    data = list(collection.find())

    # Convert the MongoDB documents to a pandas DataFrame
    df = pd.DataFrame(data)
    df.drop("_id", axis=1, inplace=True)
    # Set the "Date" field as the DataFrame index
    df.set_index("Date", inplace=True)
    return df


def filter_data(df, start_date, end_date):
    # Filter the DataFrame based on the selected date range
    filtered_df = df.loc[start_date:end_date]
    return filtered_df


# Set the title of the Streamlit app
st.title("S&P 500 Price Dashboard")

# Load all data from MongoDB collection
df = load_data()
```

```python
# Add date range selection sidebar
st.sidebar.title("Date Range Selection")
default_start_date = datetime.datetime(2023, 5, 1)
start_date = st.sidebar.date_input("Start Date", default_start_date)
end_date = st.sidebar.date_input("End Date")

# Add checkbox selection for different values
show_value = st.sidebar.checkbox("Price", value=True)
show_pred1 = st.sidebar.checkbox("Prediction 1")
show_pred2 = st.sidebar.checkbox("Prediction 2")
show_pred3 = st.sidebar.checkbox("Prediction 3")

# Filter data based on the selected date range
filtered_df = filter_data(df, start_date, end_date)
# Rearrange column order
filtered_df = filtered_df[["Value", "Pred1", "Pred2", "Pred3"]]

selected_rows = df.iloc[[-4, -3, -2, -1]]
# Write the selected rows
date_format = "%Y-%m-%d"
st.header("Price and predictions relative to the last known day")
st.header(f"Date: {selected_rows.index[-4].strftime(date_format)}")
st.header(f"Value: {round(selected_rows['Value'].iloc[0], 2)}")
st.header(f"Next day prediction: {round(selected_rows['Pred1'].iloc[1], 2)}")
st.header(f"Two days prediction: {round(selected_rows['Pred2'].iloc[2], 2)}")
st.header(f"Three days prediction: {round(selected_rows['Pred3'].iloc[3], 2)}")
```

```
# Plot the selected values over time
fig, ax = plt.subplots()
if show_value:
    ax.plot(filtered_df["Value"], label="Value")
if show_pred1:
    ax.plot(filtered_df["Pred1"], label="Prediction 1")
if show_pred2:
    ax.plot(filtered_df["Pred2"], label="Prediction 2")
if show_pred3:
    ax.plot(filtered_df["Pred3"], label="Prediction 3")
ax.set_xlabel("Date")
ax.set_ylabel("Price")
ax.set_title("Stock Price Comparison")
plt.xticks(rotation=45)
ax.legend()
st.pyplot(fig)

# Display the filtered data in the Streamlit app
st.write(filtered_df)
```