

Python para el Análisis de Datos. Proyecto Final (MEBDS)

Realizado por: *Rodrigo de la Nuez Moraleda y Marcos Castro Cacho*

Descripción de cómo se ha afrontado el proyecto y por qué.

Este proyecto se ha llevado a cabo con el objetivo de aplicar y profundizar en los conocimientos adquiridos en la asignatura *Python para el Análisis de Datos*. Combinando el aprendizaje académico con nuestra propia investigación de librerías y herramientas externas, hemos creado una aplicación que permite analizar datos reales y actualizados de los precios de distintos pares de divisas.

Para ello se ha creado una interfaz interactiva que permite crear un gráfico de velas, junto con varios indicadores bursátiles, a partir de la información OHLC de un par de divisas específico, el intervalo de tiempo usado para la agregación y, opcionalmente, un marco temporal que acote los resultados.

El proceso que se ha seguido para afrontar el proyecto y llevar a cabo el desarrollo del programa creado se puede dividir en varias fases. En primer lugar, estudiamos los datos devueltos por la API de Kraken para el par ETH/USDT con el objetivo de comprender la estructura de los mismos, así como la mejor manera de manipularlos para lograr nuestros objetivos. Tras ello, utilizamos *webscraping* para obtener todos los pares de divisas disponibles en la API y tener capacidad de elección entre estas opciones.

Para la representación gráfica de la cotización de las divisas y los indicadores calculados, se hicieron pruebas con distintas librerías (`matplotlib`, `mplfinance`); pero finalmente optamos por utilizar `plotly` para ello. Por último, estudiamos como utilizar `streamlit` para ofrecer al usuario una experiencia interactiva. Tras este punto, centramos nuestros esfuerzos en refinar nuestro código (añadir nuevas funcionalidades, mejorar el proceso y la estructura, comentar su funcionamiento, etc.), agregar pruebas que nos permitan evaluar la corrección de nuestro programa, estudiar la reproducción de nuestro entorno a través de distintas herramientas y documentar de todo el proceso en esta memoria.

Descripción de la estructura del código y estructura de ficheros.

Nuestro programa necesita de tres ficheros distintos para su correcta ejecución:

- `graphs.py`, que contiene la `class Graph` con métodos para la obtención de los datos y la representación gráfica de los mismos (junto con los indicadores asociados), según el par de divisas, el intervalo de agregación y el marco temporal seleccionados.
- `front.py`, que contiene la `class Front`. Esta crea la interaz de usuario con la que se puede interactuar, de forma que la información mostrada coincida con la seleccionada, haciendo uso de los *widgets* y botones que `streamlit` proporciona para la rápida y fácil creación de aplicaciones web.
- `main.py`, que contiene el código necesario para la ejecución de la aplicación.

Estos ficheros deben encontrarse en el mismo directorio para que el programa pueda acceder a las distintas funcionalidades que han sido creadas sin la aparición de excepciones. Contamos, además, con un fichero `tests.py` para la ejecución de las pruebas de software creadas; un fichero `requirements.txt` para poder instalar fácilmente todas las librerías necesarias, con las versiones adecuadas. para la ejecución del código; un fichero `setup.py` ([añadir descripción](#)) y un fichero `dockerfile` ([añadir descripción](#)).

Descripción de la forma de ejecución del código.

El comando `streamlit run main.py` permite la ejecución del programa desde su directorio.

1 Lectura y representación del movimiento del par de monedas

(añadir descripción)

1.1 Descarga de los datos

Para la descarga de los datos contábamos con las siguientes posibilidades: utilizar la librería **krakenex**, hacer una descarga directa a través de **pandas** o descargar directamente el **.csv** y utilizar el archivo de forma local. Debido a que de estas tres opciones la única que permite recoger información de distintos pares de divisas para crear una herramienta interactiva es la primera de ellas, nuestro programa lleva a cabo esta fase del proceso a través de una consulta a la API de Kraken usando la librería **krakenex**.

En la **class** **Graph** del fichero **graphs.py**, inicializamos el cliente de Kraken para hacer una llamada al endpoint publico OHLC con la función **def obtain_data(self)** y almacenamos en un dataframe de **pandas** toda la información recogida en la consulta para la selección escogida por el usuario.

Esta clase se inicializa como se muestra en el siguiente fragmento del código:

```
class Graph:

    # Constructor for initializing a Graph instance
    def __init__(self, pair='XETHZUSD', interval=1440, divisor=1,
                  since=None, until=None):

        self.pair = pair          # The currency pair to be analyzed
        self.interval = interval  # Time interval for each data point in minutes
        self.divisor = divisor    # Divisor for interval adjustment
        self.since = since        # Start of the time window
        self.until = until        # Time limit for the time window
```

Para la obtención de datos a través de la consulta a la API, se requieren los siguientes argumentos:

- **pair**. Denota el identificador del par de divisas para el cual se extraerán datos. Para ofrecer al usuario la posibilidad de elegir entre todos los pares de divisas para los que Kraken tiene información, hemos usado la librería **requests** para crear en **front.py** una función **def get_kraken_pairs()** que guarda en la tupla **kraken_pairs** todos los pares de divisas disponibles a través del endpoint público [AssetPairs](#).

```
# Import the requests library for HTTP request handling
import requests

# Retrieves all available currency pairs from the Kraken API
def get_kraken_pairs():

    # Endpoint URL for fetching Kraken currency pairs
    url = 'https://api.kraken.com/0/public/AssetPairs'

    # Send a GET request to the Kraken API
    response = requests.get(url)

    # Convert the response to JSON format
    response_json = response.json()
```

```

# Extract currency pair identifiers from the JSON data
pairs = response_json['result'].keys()

# Return the currency pairs as a tuple
return tuple(pairs)

# Get and store the list of currency pairs available on Kraken
kraken_pairs = get_kraken_pairs()

```

Para seleccionar un par de divisas de entre los disponibles, hemos utilizado **streamlit** para generar un menú dropdown en la aplicación (importamos esta librería como sigue: **import streamlit as st**).

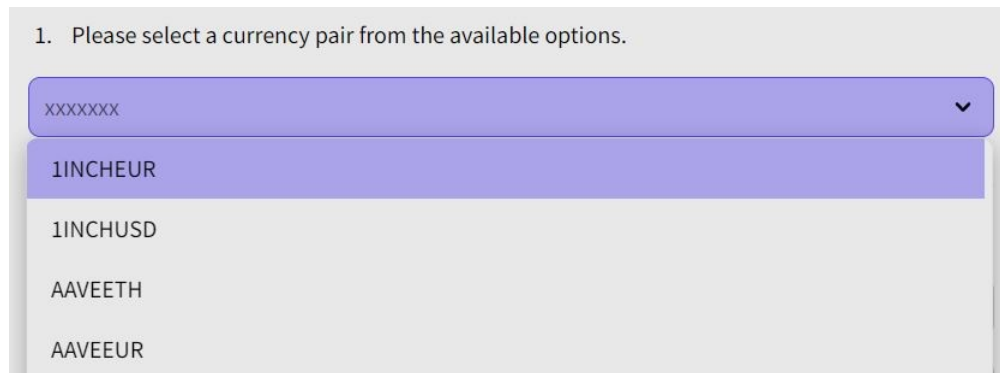
```

# Prompt user to select a currency pair from the pairs retrieved
st.write("1. Please select a currency pair from the available options:")

# Dropdown menu for selecting a currency pair
self.currency_pair = st.selectbox(
    label = 'placeholder',           # Streamlit's selectbox requires a label
    options = kraken_pairs,         # List of currency pairs from Kraken
    index = None,                   # Index of the preselected option
    placeholder = "xxxxxxx",       # Placeholder text in the dropdown
    label_visibility = "collapsed"
)

```

Así se vería el menú dropdown en la aplicación creada con **streamlit**:



- **interval**. Denota el intervalo de tiempo (en minutos) que se utiliza para la agregación, es decir, cada vela del gráfico generado contará con datos de tantos minutos como el valor de **interval**. Para este atributo ofrecemos dos opciones al usuario: seleccionar entre los intervalos de tiempo más comunes (para los que **krakenex** ofrece datos con una consulta directa y así se indica en la [documentación](#)) o escribir el valor del intervalo que desea seleccionar como un número entero entre 1 y 43200 (1 mes).

```

# A dictionary mapping time intervals to their durations in minutes
intervals = {"1m":1, "5m":5, "15m":15, "30m":30, "1h":60, "4h":240,
             "1d":1440, "1w":10080, "2w":21600}

# Separate lists of interval labels and their corresponding durations
keys, options = intervals.keys(), intervals.values()

```

Si el intervalo toma un valor a elección del usuario, utilizamos la función `def find_largest_interval(n)` en `front.py` para realizar la consulta con el mayor divisor de entre los intervalos predeterminados del entero introducido. La función `def aggregate_intervals(self, df)` en `graphs.py` permite la agregación de los distintos campos según el intervalo que el usuario ha seleccionado desde la aplicación.

```
# Finds the largest duration in 'options' that is a divisor of n
def find_largest_divisor(n):
    # Filters durations that are divisors of n
    valid_divisors = [d for d in options if n % d == 0]

    # Returns the largest divisor found
    return max(valid_divisors)
```

Llevamos a cabo este proceso ya que la consulta a la API de Kraken devuelve una respuesta con un límite de tamaño de 720 intervalos por consulta. Debido a ello, al realizar ciertas agregaciones obtendremos unos pocos valores en nuestra gráfica. Este problema se podría solucionar realizando múltiples consultas a la API y juntando las respuestas a las distintas consultas antes de realizar la agregación. Sin embargo, esto no es posible pues, al hacer varias consultas de forma muy seguida, recogemos el siguiente mensaje de error:

```
There was an error with the API call
An error occurred: ['EGeneral:Too many requests']
```

Para la selección del intervalo entre las distintas opciones, hemos creado un conjunto de botones y un objeto `number_input` de `streamlit` que únicamente acepta números enteros entre 1 y 43200 (ambos incluidos).

```
class Front:
    def __init__(self):
        < código de la función >

        # Retrieve or initialize the selected time interval for each candle
        st.session_state.selected_option = st.session_state.get("selected_option")
        st.session_state.is_custom_interval = st.session_state.get("is_custom_interval")

        # Store the time interval for each candle from the session state
        st.session_state.custom_interval = st.session_state.get("custom_interval")
        self.time_interval = st.session_state.selected_option

        < código de la función >

    def select_boxes(self):
        < código de la función >

        st.markdown("2. Choose a time interval for the candles from
                                the most commonly used options..." +
                                f"""
                                <style>
                                div.stButton > button {{
                                    width: 100%;
                                }}
                                </style>""", unsafe_allow_html=True )
```

```

# Generate a row of buttons for selecting time intervals
columns = st.columns(len(keys))
for i, key in enumerate(keys):
    with columns[i]:
        button_key = f"button--{key}"
        if st.button(key, key=button_key):
            # Sets the selected time interval
            self.time_interval = int(intervals[key])
            # Updates the session state
            st.session_state.selected_option = self.time_interval

# Button for allowing custom time interval input
if st.button("...or enter a custom time interval (in minutes)", key="Other"):
    # Input field for custom time interval in minutes
    st.session_state.is_custom_interval = not st.session_state.is_custom_interval

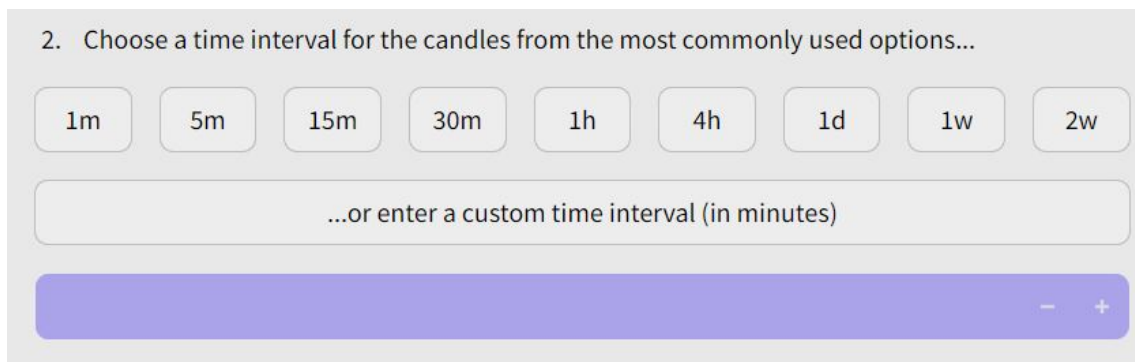
if st.session_state.is_custom_interval:
    st.session_state.custom_interval = st.number_input('Custom interval',
        min_value=1, max_value=43200, step=1,
        value=None, label_visibility='collapsed')

if st.session_state.custom_interval is not None:
    self.time_interval = st.session_state.custom_interval
    # Update the time interval with the custom input
    st.session_state.selected_option = st.session_state.custom_interval

```

< código de la función >

Así se verían el conjunto de botones y el `number_input` en la aplicación creada con `streamlit`:



The screenshot shows a web interface titled "2. Choose a time interval for the candles from the most commonly used options...". It features a row of nine buttons labeled "1m", "5m", "15m", "30m", "1h", "4h", "1d", "1w", and "2w". Below these buttons is a text input field with the placeholder text "...or enter a custom time interval (in minutes)". At the bottom, there is a purple bar with a minus sign and a plus sign, likely for adjusting the input value.

- **since.** Denota el tiempo para el cual se toman los primeros datos en la consulta, es un número entero que se corresponde con un timestamp de Unix. Este valor es el número de segundos transcurridos desde la medianoche UTC del 1 de enero de 1970. Por ejemplo, el timestamp Unix "1548111600" se correspondería con la fecha y hora "2019-01-21 23:00:00 UTC". Cabe destacar que si el tiempo introducido es anterior al primero de los 720 intervalos devueltos por la API, esta funcionalidad no empezará en el tiempo indicado sino en el primero de esos 720 intervalos.

Además, hemos buscado acotar la última fecha para la que se muestran datos y, dado que `krakenex` no cuenta con dicha funcionalidad, hemos añadido un atributo `until` a la `class Graph` con el que aplicamos un filtro tras la obtención de los datos. Para la selección de los valores de `since` y `until`, hemos usado las funcionalidades de `streamlit` para crear dos objetos `expander` con un `date_input` cada uno.

```
class Front:
    def __init__(self):
        < código de la función >

        # Initialize since and until attributes
        self.since, self.until = None, None

    # Method to create user input interfaces, including dropdowns and buttons
    def select_boxes(self):
        < código de la función >

        st.markdown("3. Optionally, choose a time interval within
                      the range of the original selection:")

        # Date picker for selecting the start date
        columns0, columns1 = st.columns([1, 1])
        with columns0:
            with st.expander("Start Date", expanded=True):
                self.since = st.date_input('start', value=None,
                                           label_visibility = "collapsed")
                if self.since is not None:
                    # Convert date to datetime
                    self.since = datetime.datetime.combine(self.since,
                                                           datetime.datetime.min.time())
                    self.since = datetime.datetime.strptime(str(self.since),
                                                             "%Y-%m-%d %H:%M:%S").timestamp()

        with columns1:
            with st.expander("End Date", expanded=True):
                self.until = st.date_input('end', value=None,
                                           label_visibility = "collapsed")
                if self.until is not None:
                    # Convert date to datetime
                    self.until = datetime.datetime.combine(self.until,
                                                           datetime.datetime.min.time())
                    self.until = datetime.datetime.strptime(str(self.until),
                                                             "%Y-%m-%d %H:%M:%S").timestamp()
```

Así se verían los objetos `expander` con el `date_input` en la aplicación creada con `streamlit`:

3. Optionally, choose a time interval within the range of the original selection:

Start Date
YYYY/MM/DD

End Date
YYYY/MM/DD

Con los inputs mencionados, en la función `def obtain_data(self)` se lleva a cabo la consulta con la que se recuperan los datos en formato JSON, con dos campos `response` y `error`. En caso de haber algún problema con la llamada a la API, se recoge la excepción en el campo `error` y, de no ser así, el campo `response` a su vez guarda la información de los campos `Time` (tiempo de apertura del intervalo), `Open` (precio de apertura del intervalo), `High` (precio más alto del intervalo), `Low` (precio más bajo del intervalo), `Close` (precio de cierre del intervalo), `VWAP` (valor del indicador para el intervalo), `Volume` (volumen acumulado del intervalo) y `Count` (número de transacciones en el intervalo).

Transformamos los datos a un dataframe de `pandas` y prescindimos de los campos `VWAP` y `Count`, pues no nos son de utilidad. Cambiamos los timestamps de Unix al tipo `datetime` de `pandas`, usamos el campo `Time` como índice del dataframe, filtramos los datos cuando se haya introducido un valor para `until` y transformamos el tipo del resto de campos de `float` a `string`.

A continuación, se han calculado dos indicadores bursátiles para su inclusión en el gráfico de velas: el SMA o *Simple Moving Average* (media móvil del precio de cierre) y el EMA o *Exponential Moving Average* (similar al SMA pero dando más peso a los datos mas recientes, lo que hace que sea más sensible a cambios a corto plazo).

En el cálculo de medias móviles, como es este caso, existe la convención de escoger una ventana de 14 intervalos y, aunque esta elección puede variar según el análisis que se busque hacer sobre los datos, hemos decidido adoptar este estándar para nuestra aplicación. No obstante, hemos tenido en cuenta casos en los que la consulta no devuelve datos para muchos intervalos y, en tal escenario, se toma una ventana de únicamente 3 intervalos para poder calcular los indicadores en un mayor porcentaje de los intervalos obtenidos. Por último, aplicamos la función de agregación cuando el intervalo seleccionado por el usuario no se encuentre entre las opciones disponibles para realizar una consulta directa a la API de Kraken.

class Graph:

```
# This function aggregates data into custom time intervals
# that are not natively provided by the Kraken API to make queries
def aggregate_intervals(self, df):

    # Resamples the DataFrame to the specified interval and aggregates key metrics
    resampled_df = df.resample(f'{self.interval}T').agg({
        'Open': 'first',
        'High': 'max',
        'Low': 'min',
        'Close': 'last',
        'SMA': 'mean',
        'EMA': 'mean',
        'Volume': 'sum'})

    return resampled_df
```

```

# Retrieves trading data from the Kraken API and stores it in a Pandas DataFrame
def obtain_data(self):

    # Initializing a Kraken API client and querying data within a try-except block
    try:
        k = krakenex.API() # Initialize the Kraken client

        # Query for OHLC data for the specified currency pair and interval
        response = k.query_public('OHLC', {'pair':self.pair,
                                           'interval':self.divisor,
                                           'since':self.since})

        # Check and raise an exception if errors exist in the response
        if response['error']:
            print(f"There was an error with the API call")
            raise Exception(response['error'])

    # Catch and print any exceptions during the data retrieval process
    except Exception as e:
        print(f"An error occurred: {e}") # Print the specific error message
        print("Error while retrieving data") # Indicate a data retrieval error

    # Process the retrieved data if no exceptions occur
    else:
        # Extract OHLC data from API response
        ohlc_data = response['result'][self.pair]

        # Convert OHLC data to a DataFrame and remove unnecessary columns
        ohlc_df = pd.DataFrame(ohlc_data, columns=["Time", "Open", "High", "Low",
                                                  "Close", "VWAP", "Volume", "Count"]).drop(['VWAP', 'Count'], axis=1)

        # Convert timestamps to datetime format and set as DataFrame index
        ohlc_df["Time"] = pd.to_datetime(ohlc_df["Time"], unit='s')
        ohlc_df.set_index(pd.DatetimeIndex(ohlc_df["Time"]), inplace=True)
        if self.until is not None: # Filter data when until is not None
            cutoff_date = pd.to_datetime(self.until, unit='s')
            ohlc_df = ohlc_df[ohlc_df.index < cutoff_date]

        # Convert all price and volume data to float type for calculations
        ohlc_df["Open"] = ohlc_df["Open"].astype(float)
        ohlc_df["High"] = ohlc_df["High"].astype(float)
        ohlc_df["Low"] = ohlc_df["Low"].astype(float)
        ohlc_df["Close"] = ohlc_df["Close"].astype(float)
        ohlc_df["Volume"] = ohlc_df["Volume"].astype(float)

        # Add Simple Moving Average (SMA) and Exponential Moving Average (EMA)
        window = 14 if ohlc_df.shape[0] >= 60 else 3
        ohlc_df['SMA'] = ohlc_df['Close'].rolling(window=window).mean()
        ohlc_df['EMA'] = ohlc_df['Close'].ewm(span=window, adjust=False).mean()

```



```

# Aggregate data into custom intervals if needed
if self.interval not in (1, 5, 15, 30, 60, 240, 1440, 10080, 21600):
    ohlc_df = self.aggregate_intervals(ohlc_df)

return ohlc_df # Return the prepared DataFrame

```

1.2 Grabado de las cotizaciones

Las cotizaciones de un par de divisas suelen ser representadas haciendo uso de un gráfico de velas para la información OHLC (Open - High - Low - Close). Para llevar a cabo esta representación creamos una figura `candlestick` con `plotly`, a partir de los datos obtenidos en la consulta a la API de Kraken. Adicionalmente, hemos incluido la información del volumen a lo largo del tiempo con un diagrama de barras que refleja el cambio en el precio con el mismo color que la vela correspondiente pero con una opacidad menor para que ambas gráficas puedan coexistir y complementarse. Por otro lado, hemos tenido en cuenta la información de cierre para añadir diagramas de líneas que representen los indicadores SMA y EMA. (cálculo del EMA)

class Graph:

```

# Static method to create a candlestick chart from OHLC data using Plotly
@staticmethod
def candlestick(ohlc_df):

    try:
        size = size = ohlc_df.shape[0]
        if size <= 14:
            df = ohlc_df[:] # Take the whole dataframe when it has small size
        else:
            df = ohlc_df[14:] # Remove the first intervals so that SMA is defined

        colors = ['#008080' if close >= open else 'red' for open, close
                  in zip(df['Open'], df['Close'])]
        fig = make_subplots(specs=[[{"secondary_y": True}]]

        # Include candlestick with range selector
        fig.add_trace(go.Candlestick(x=df.index, open=df['Open'], high=df['High'],
                                     low=df['Low'], close=df['Close'], name=''), secondary_y=True)

        # Bar diagram displaying the Volume data
        fig.add_trace(go.Bar(x=df.index, y=df['Volume'], marker_color=colors,
                             opacity=0.25, showlegend=False), secondary_y=False)

        # Line chart displaying the computed SMA values
        fig.add_trace(go.Scatter(x=df.index, y=df['SMA'],
                                 marker=dict(color='#0000FF'), opacity=0.35, name='SMA'),
                      secondary_y=True)

        # Line chart displaying the computed EMA values
        fig.add_trace(go.Scatter(x=df.index, y=df['EMA'],
                                 marker=dict(color='#FF0000'), opacity=0.35, name='EMA'),
                      secondary_y=True)

```

```

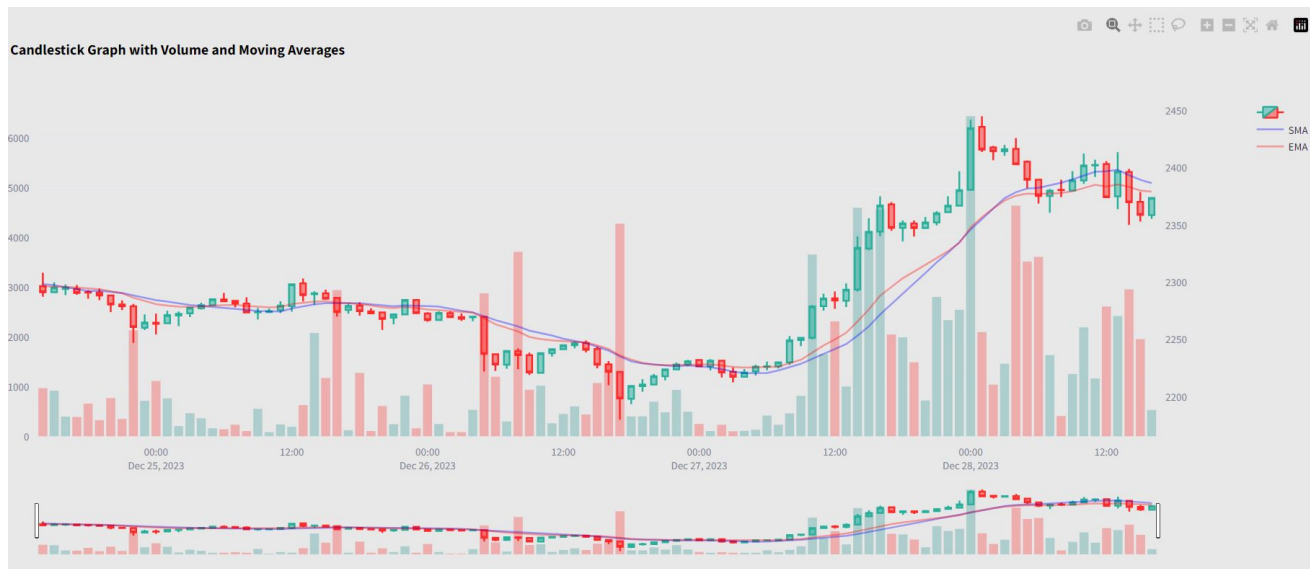
fig.layout.yaxis2.showgrid = False
fig.layout.title = 'Candlestick Graph with Moving Average'
fig.layout.height = 400
fig.layout.width = 650

return fig # Return the Figure object for plotting

except Exception as e:
    # Handle exceptions in chart creation and return an empty figure in case of error
    print(f"An error occurred while creating the candlestick chart: {e}")
    return go.Figure() # Return an empty Plotly Figure object if an error occurs

```

El gráfico de velas resultante, con el diagrama de barras para el Volumen y los diagramas de líneas para los indicadores SMA y EMA, se vería como sigue en la aplicación creada con **streamlit**:



Calculadora de ganancias. Damos también la opción de hacer una pequeña simulación utilizando los datos que vemos en las gráficas. Para ello cogemos las señales de compra y simulamos una cartera en la que compramos 100 unidades de la moneda por señal y vendemos 100 unidades por cada señal de venta. La gráfica representa las ganancias o pérdidas que se tendrían en el periodo. No se muestra ninguna gráfica si no se han generado señales de compra en el periodo.

2 Indicadores técnicos

(añadir descripción) Como ya hemos explicado en el apartado anterior, se han calculado y representado en el gráfico de velas los indicadores SMA (Simple Moving Average) y EMA (Exponential Moving Average). Sin embargo, el indicador principal de esta práctica ha sido el oscilador estocástico, el cual explicaremos a continuación como ha sido calculado y su representación gráfica en la aplicación creada.

2.1 Cálculo y representación gráfica del oscilador estocástico

El estocástico es un indicador de momentum desarrollado en los años 50. Se utiliza para mostrar la ubicación del cierre actual en relación con el rango alto/bajo durante un número determinado de períodos. La fórmula

para calcular el estocástico es la siguiente: Donde:- Precio de cierre actual: es el último precio al que se negoció un activo.- Mínimo más bajo: es el precio más bajo negociado del activo durante un período determinado.- Máximo más alto: es el precio más alto negociado del activo durante un período determinado. El estocástico se representa con dos líneas, %K y %D. La línea %D es una media móvil de %K. Cuando la línea %K cruza por encima de la línea %D, se genera una señal de compra. Cuando la línea %K cruza por debajo de la línea %D, se genera una señal de venta. TODO comprimir este párrafo a una o dos frases o borrar directamente. Explicación de nuestra forma de calcularlo: Primero, calcula el oscilador estocástico (%K) y su media móvil (%D) utilizando los precios de cierre, alto y bajo de un DataFrame de pandas. Luego, genera señales de compra y venta basadas en las condiciones especificadas en las líneas 117 y 118.

La fórmula $(df['Close'] - df['L14']) / (df['H14'] - df['L14']) * 100$ es la implementación de la fórmula del oscilador estocástico que mide la relación entre el precio de cierre actual y el rango de precios en un número determinado de períodos.

La línea `df['%D'] = df['%K'].rolling(window=3).mean()` calcula la media móvil de 3 períodos de %K, que se conoce como %D. La media móvil se utiliza para suavizar las fluctuaciones de %K y generar una línea de señal para las señales de compra y venta.

También generamos las señales de compra y venta que utilizaremos en la simulación de beneficios.

- Las de compra se generan cuando la línea %K cruza por encima de la línea %D y el valor de %D es menor que 20
- Las de venta se generan cuando la línea %K cruza por debajo de la línea %D y el valor de %D es mayor que 80

2.2 Cálculo del oscilador estocástico sobre una media móvil

2.3 Gráfico del indicador junto con la cotización del par calculado

Graficar los dos juntos

Finalmente damos al usuario la opción de graficar las anteriores opciones juntas. TODO captura gráficas juntas

Adicionalmente aquí damos la opción de simular ganancias que será explicada en su sección más adelante. TODO es esta línea necesaria?

(a) Media móvil exponencial

Es otro tipo de media móvil que da más peso a los datos más recientes, lo que hace que sea más sensible a cambios a corto plazo. TODO confirmame que esto es correcto?

(b) Media móvil simple

TODO tenemos esto siquiera?, no es el smoothed stochastic oscillator?

3 Estructuración

(añadir descripción)

3.1 Funciones

3.2 Utilización de clases

3.3 Manejo de errores y excepciones

4 Puntuación Extra

(añadir descripción)

4.1 Testeo y cobertura (unit-testing, integration-testing)

Durante todo el proceso hemos buscado evitar casos que puedan dar pie a errores por incongruencia con el tipado de las variables o la no selección de alguno de los atributos necesarios par el cálculo de los distintos tipos de gráficas.

4.2 Facilitar los mecanismos para la reproducción del entorno virtual

4.3 Distribución del proyecto a través de PyPi