

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

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

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 todo el proceso que se ha seguido en esta memoria.

Nuestro programa necesita (principalmente) 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 interfaz 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; un fichero `setup.py` que nos ayuda a publicar en `pip` y un fichero `dockerfile` para generar la imagen de Docker. Además, hemos incluido un archivo `README.md` que proporciona una descripción general e instrucciones del proyecto.

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

Se puede acceder a la aplicación directamente en <https://13vhpftnhq75225rcpsappc.streamlit.app>.

1 Lectura y representación del movimiento del par de monedas

Esta sección aborda el proceso de descarga y análisis de datos de pares de divisas a través de la API de Kraken, ilustrando cómo se visualizan las fluctuaciones de precios mediante un gráfico de velas interactivo.

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**).

```

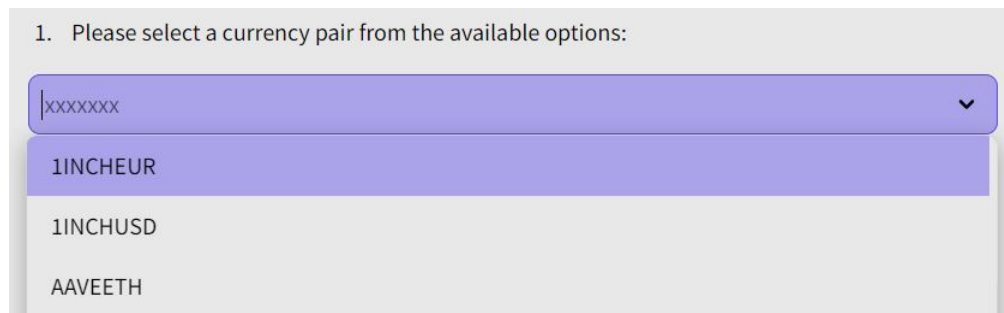
class Front:
    [...]

    # Method to create user input interfaces, including dropdowns and buttons
    def select_boxes(self):
        # 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(interval, 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 solamente acepta los intervalos de `options` y devuelve una respuesta con un límite de tamaño de 720 intervalos por consulta, por lo que buscamos el intervalo que nos permita contar con más información tras la agregación. Debido a esta limitación, al realizar ciertas agregaciones obtendremos unos pocos valores en nuestras gráficas. 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 que indica que se han realizado demasiadas consultas a la API.

```
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). Además, utilizamos el diccionario `st.session_state` para almacenar la selección del intervalo y que no se pierda su valor a lo largo del tiempo en la aplicación de `streamlit` creada.

```
class Front:
    def __init__(self):
        [...]
        # Retrieve or initialize the selected time interval for each candle
        st.session_state['selected_option'] =
            st.session_state.get("selected_option", None)

        st.session_state['is_custom_interval'] =
            st.session_state.get("is_custom_interval", None)

        # Retrieve or initialize the customized time interval for each candle
        st.session_state['custom_interval'] =
            st.session_state.get("custom_interval", None)
```

```

# Store the time interval for each candle from the session state
self.time_interval = st.session_state['selected_option']
[...]

def select_boxes(self):
    [...]

    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 different values stored in the session state
            st.session_state['selected_option'] = self.time_interval
            st.session_state['is_custom_interval'] = None
            st.session_state['custom_interval'] = None

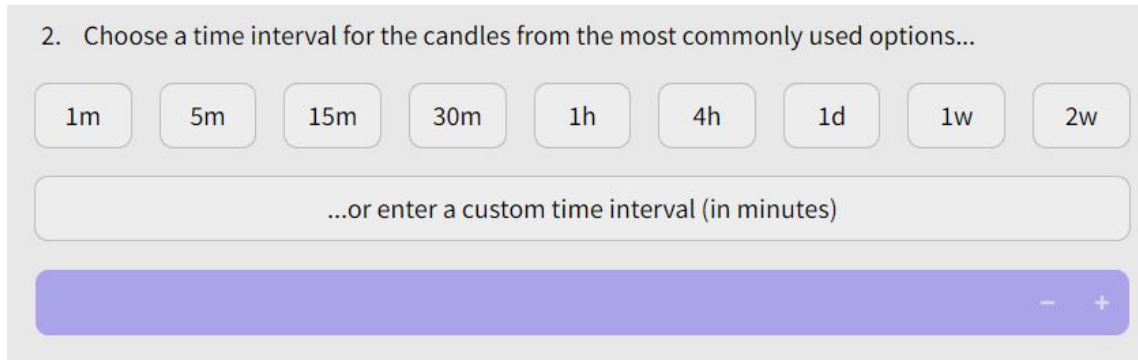
# Button for allowing custom time interval input
if st.button("...or enter a custom time interval (in minutes)", key="Other"):
    st.session_state['is_custom_interval'] =
        not st.session_state['is_custom_interval']

# Input field for custom time interval in minutes
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')

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

[...]
```

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



- **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):
        [...]
        # Initialize since and until attributes
        self.since, self.until = None, None

    def select_boxes(self):
        [...]
        st.markdown("3. Optionally, choose a time window 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`:

Con los inputs mencionados, en `def obtain_function(pair, interval, divisor, since, until)` se lleva a cabo la consulta con la que se recuperan los datos en formato JSON, con dos campos `result` 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 `result` 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). Nótese que utilizamos el decorador `@st.cache_data(ttl=300)` para guardar el resultado de esta función en la caché de `streamlit` durante 5 minutos y así no repetir innecesariamente procesos que ya se han realizado con antelación.

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 `string` a `float`.

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 más 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 `def aggregate_intervals(interval, df)` cuando el intervalo seleccionado por el usuario no se encuentre entre las opciones disponibles para realizar una consulta directa a la API de Kraken. Además, guardamos los campos necesarios para el cálculo del oscilador estocástico, la media móvil de este indicador y las señales de compra y venta para una estrategia de trading que hemos definido. Estos últimos campos serán explicados más adelante en sus respectivas secciones.

```

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

    # Resamples the DataFrame to the specified interval and aggregates key metrics
    resampled_df = df.resample(f'{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
@st.cache_data(ttl=300) # Decorator to cache the data in Streamlit for 5 minutes
def obtain_function(pair, interval, divisor, since, until):

    # 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, interval and start date
        response = k.query_public('OHLC',{ 'pair':pair, 'interval':divisor, 'since':since})
        if response['error']: # Check and raise an exception if an error is retrieved
            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:
        ohlc_data = response['result'][pair] # Extract OHLC data from the API response

        # 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 Unix timestamps to pandas 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 until is not None: # Filter data when until is not None
            cutoff_date = pd.to_datetime(until, unit='s')
            ohlc_df = ohlc_df[ohlc_df.index < cutoff_date]

```



```

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

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

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

window = 14 if ohl_df.shape[0] >= 60 else 3 # Determine window size
ohl_df['L14'] = ohl_df['Low'].rolling(window=window).min()
ohl_df['H14'] = ohl_df['High'].rolling(window=window).max()

ohl_df['%K'] = (ohl_df['Close'] - ohl_df['L14']) /
               (ohl_df['H14'] - ohl_df['L14']) * 100
ohl_df['%D'] = ohl_df['%K'].rolling(window=3).mean()

ohl_df['Buy_Signal'] = ((ohl_df['%K'] > ohl_df['%D']) &
                        (ohl_df['%K'].shift(1) < ohl_df['%D'].shift(1)) &
                        (ohl_df['%D'] < 20))

ohl_df['Sell_Signal'] = ((ohl_df['%K'] < ohl_df['%D']) &
                        (ohl_df['%K'].shift(1) > ohl_df['%D'].shift(1)) &
                        (ohl_df['%D'] > 80))

return ohl_df # Return the prepared DataFrame

class Graph:
    [...]
    # Retrieves trading data from the Kraken API and stores it in a Pandas DataFrame
    def obtain_data(self):
        return obtain_function(self.pair, self.interval,
                               self.divisor, self.since, self.until)

```

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 para el SMA y el EMA.

```

# The class Graph is designed for constructing candlestick and
# stochastic oscillator graphs with moving averages for trading analysis
class Graph:
    [...]

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

        try:
            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='',
                                         legendgroup='group', legendrank=1), 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',
                                     legendgroup='group', legendrank=2), 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',
                                     legendgroup='group', legendrank=3), secondary_y=True)

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

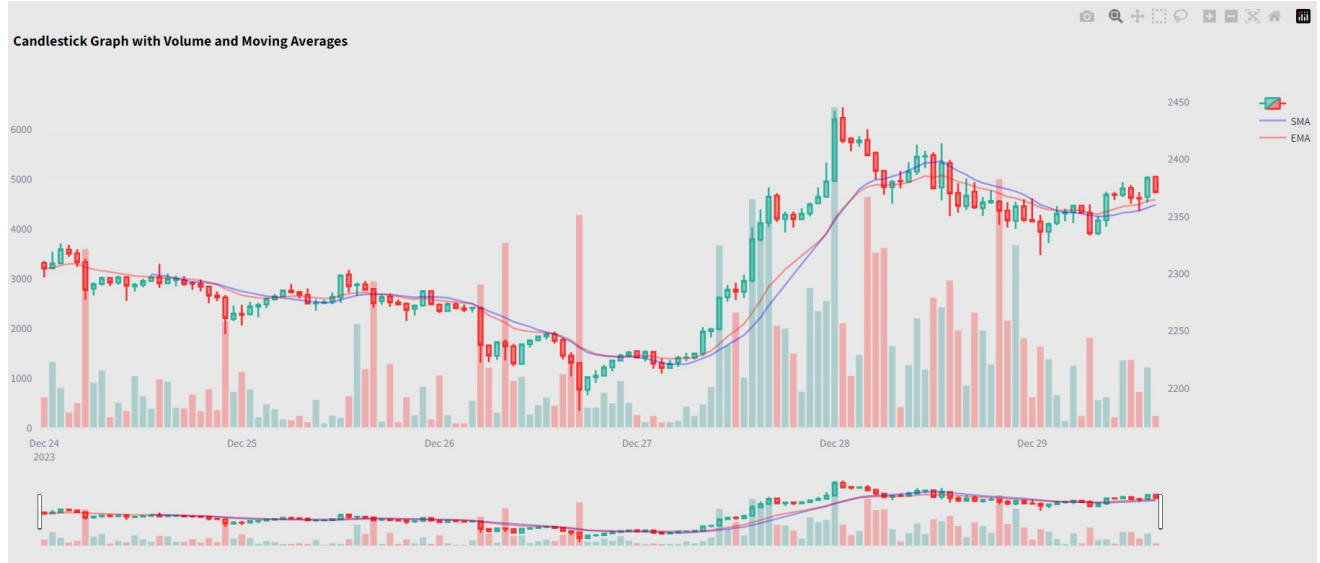
            return fig # Return the Figure object for plotting

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

            # Return an empty Plotly Figure object if an error occurs
            return go.Figure()

```

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`:



2 Indicadores técnicos

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 (de forma única y en combinación con el gráfico de velas). El VWAP es otro indicador que podría haberse añadido junto con la representación gráfica de la información OHLC gracias a que viene dado directamente por la respuesta de la API de Kraken.

2.1 Cálculo y representación gráfica del oscilador estocástico

El oscilador estocástico, desarrollado en la década de 1950, es un indicador de momentum utilizado en el análisis técnico de los mercados financieros. Su función principal es identificar condiciones de sobrecompra o sobreventa en el precio de un activo. Lo hace mostrando la posición del cierre actual del mercado en relación con el rango de precios durante un período específico que cubre un número de intervalos previos al intervalo para el que se está calculando el indicador, comúnmente se escogen 14 intervalos. Un oscilador estocástico alto indica que el precio está cerrando cerca del máximo reciente y si es bajo, un cierre cercano al mínimo reciente.

La fórmula para calcular el oscilador estocástico es la siguiente:

$$\%K_{i,range} = \frac{Close_i - \min(Low_{range})}{\max(High_{range}) - \min(Low_{range})} \times 100$$

donde:

- $Close_i$ es el último precio al que se negoció un activo en el intervalo i ,
- $\min(Low_{range})$ es el precio más bajo negociado del activo durante un período $range$ determinado,
- $\max(High_{range})$ es el precio más alto negociado del activo durante un período $range$ determinado.

Normalmente, el oscilador estocástico se muestra con una media móvil $%D$. Habitualmente, la línea $%K$ se calcula utilizando un período estándar de 14 intervalos, reflejando el nivel actual de precio en relación con los rangos de precios altos y bajos durante este período. La línea $%D$, por otro lado, es una media móvil de la línea $%K$, y típicamente se calcula sobre un periodo de 3 intervalos, lo que ayuda a suavizar las fluctuaciones de $%K$ y proporciona una mejor perspectiva de la tendencia general.

En nuestra metodología, hemos adoptado estas convenciones: utilizamos 14 intervalos para calcular la línea $%K$ y 3 intervalos para la línea $%D$. Sin embargo, también hemos incorporado un ajuste importante en situaciones donde el total de intervalos disponibles es menor a 60. En tales casos, reducimos el período de cálculo para la línea $%K$ a 3 intervalos. Gracias a esta modificación, nos aseguramos de que los indicadores calculados tengan la mayor relevancia y precisión posibles en escenarios con menos datos.

```
# Retrieves trading data from the Kraken API and stores it in a Pandas DataFrame
@st.cache_data(ttl=300) # Decorator to cache the data in Streamlit for 5 minutes
def obtain_function(pair, interval, divisor, since, until):
    [...]
    window = 14 if ohlc_df.shape[0]>=60 else 3
    ohlc_df['L14'] = ohlc_df['Low'].rolling(window=window).min()
    ohlc_df['H14'] = ohlc_df['High'].rolling(window=window).max()
    ohlc_df['%K'] = (ohlc_df['Close'] - ohlc_df['L14']) /
                    (ohlc_df['H14'] - ohlc_df['L14']) * 100
    ohlc_df['%D'] = ohlc_df['%K'].rolling(window=3).mean()
    [...]

class Graph:
    [...]

    @staticmethod # Calculate and graph the stochastic oscillator and its mobile mean
    def stochastic(df):

        try:
            data = [# Line chart for the '%D' line of the stochastic oscillator
                    go.Scatter(x=df.index, y=df['%D'], name='Smoothed Stochastic',
                               marker=dict(color='#b2b2b2'),
                               legendgroup='group', legendrank=5),

                    # Line chart for the '%K' line of the stochastic oscillator
                    go.Scatter(x=df.index, y=df['%K'], name='Stochastic Oscillator',
                               marker=dict(color='#4c4c4c'),
                               legendgroup='group', legendrank=4),

                    # Horizontal line at 20%
                    go.Scatter(x=df.index, y=[20]*len(df.index), mode='lines',
                               name='20% threshold', line=dict(color='purple',
                               width=1, dash='dash'), showlegend=False),

                    # Horizontal line at 80%
                    go.Scatter(x=df.index, y=[80]*len(df.index), mode='lines',
                               name='80% threshold', line=dict(color='purple',
                               width=1, dash='dash'), showlegend=False)]
```

```

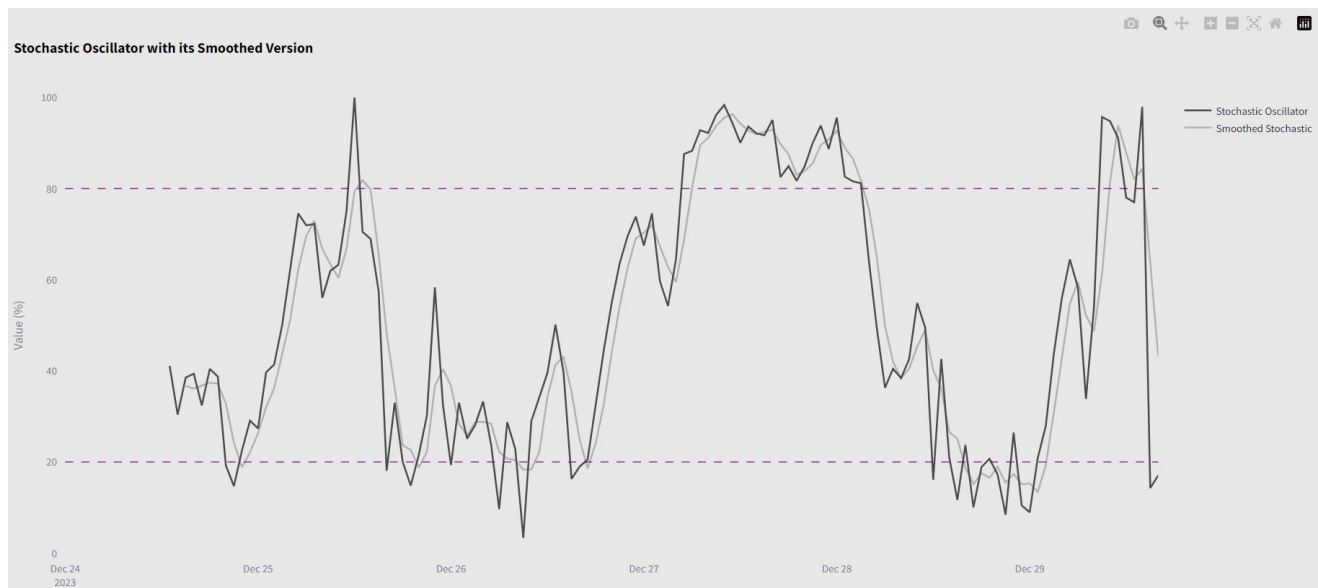
# Define the layout for the plotly figure, setting titles and axis labels
layout = go.Layout(title='Stochastic Oscillator with its Smoothed Version',
                    yaxis=dict(title='Value (%)', range=[0,100]))

# Create a Figure object with the candlestick data
fig = go.Figure(data=data, layout=layout)
fig.layout.height = 250
return fig # Return the Figure object for plotting

# Handle exceptions in chart creation and return an empty figure in case of error
except Exception as e:
    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 para el oscilador estocástico y su media móvil, se vería como sigue en la aplicación de **streamlit**:



Los indicadores %K y %D pueden utilizarse para seguir la siguiente estrategia de trading: cuando %K cruza por encima a %D y %D es inferior al 20% se genera una señal de compra, mientras que cuando %K cruza por debajo a %D y la línea %D es superior al 80% se genera una señal de venta. Esta estrategia ha sido implementada en nuestra aplicación para poder observar el rendimiento que se tiene al seguir este enfoque.

Para ello, hemos usado las señales de compra para simular una cartera en la que se compran 100 unidades de la moneda por señal y vendemos 100 unidades por cada señal de venta (siempre que la cartera cuente con dicha cantidad). La gráfica generada representa las ganancias o pérdidas que se tendrían a lo largo del periodo graficado. Si no se han generado señales de compra en el periodo, no se muestra ninguna grafica.

```
class Graph:
```

```
    [...]
```

```
    # Function to calculate the profit from trading based on buy and sell signals  
    def calculate_profit(self, df):
```

```
        try:
```

```
            # Initialize variables to track coins held and total amount spent  
            coins = 0  
            total_spent = 0
```

```
            # Buy_Price is set to the Close price where Buy_Signal==True, otherwise NaN  
            df['Buy_Price'] = np.where(df['Buy_Signal'], df['Close'], np.nan)
```

```
            # Sell_Price is set to the Close price where Sell_Signal==True, otherwise NaN  
            df['Sell_Price'] = np.where(df['Sell_Signal'], df['Close'], np.nan)  
            df['Profit'] = 0
```

```
            # Loop through each row in the DataFrame  
            for i in range(len(df)):
```

```
                # If there's a buy signal, increase coins and add to total spent  
                if df['Buy_Signal'].iloc[i]:  
                    coins += 100  
                    total_spent += df['Buy_Price'].iloc[i] * 100
```

```
                # If there's a sell signal and enough coins are held,  
                # reduce coins and subtract from total spent  
                if df['Sell_Signal'].iloc[i] and coins >= 100:  
                    coins -= 100  
                    total_spent -= df['Sell_Price'].iloc[i] * 100
```

```
                # Calculate profit: current value of held coins minus total amount spent  
                df.loc[df.index[i], 'Profit'] = (df.loc[df.index[i], 'Close'] * coins) -  
                    total_spent
```

```
            return df # Return the modified DataFrame with the 'Profit' column
```

```
        # Catching and printing any exceptions that occur during the function execution  
        except Exception as e:
```

```
            print(f"An error occurred while creating the profit data: {e}")
```

```
        # Returning an empty DataFrame in case of an exception  
        return pd.DataFrame()
```

```

# Function to create a profit graph from a DataFrame containing buy and sell signals
def profit_graph(self, df):

    try:

        # Check if there are any buy signals in the DataFrame
        if not df['Buy_Signal'].any():
            return None # Return None if there are no buy signals

        # Get the index of the first buy signal in the DataFrame
        first_buy_signal = df[df['Buy_Signal']].index[0]

        # Slice the DataFrame from the first buy signal onwards
        df = df.loc[first_buy_signal:]

        # Create a list of Scatter plots for the profit graph
        data = [
            # Line plot for cumulative profit over time
            go.Scatter(x=df.index, y=df['Profit'].cumsum(), name='Profit',
                       marker=dict(color='#0d0c52')),

            # Marker plot for points where buy signals occur
            go.Scatter(x=df[df['Buy_Signal']].index,
                       y=df['Profit'].cumsum()[df['Buy_Signal']],
                       mode='markers', marker=dict(color='#05e3a0',
                                                    size=10), name='Buy Signal'),

            # Marker plot for points where sell signals occur
            go.Scatter(x=df[df['Sell_Signal']].index,
                       y=df['Profit'].cumsum()[df['Sell_Signal']],
                       mode='markers', marker=dict(color='#f77088',
                                                    size=10), name='Sell Signal')
        ]

        # Define the layout for the graph, including axis labels and margins
        layout = go.Layout(
            yaxis=dict(title='Value'),
            margin=dict(l=40, r=40, t=20, b=40))

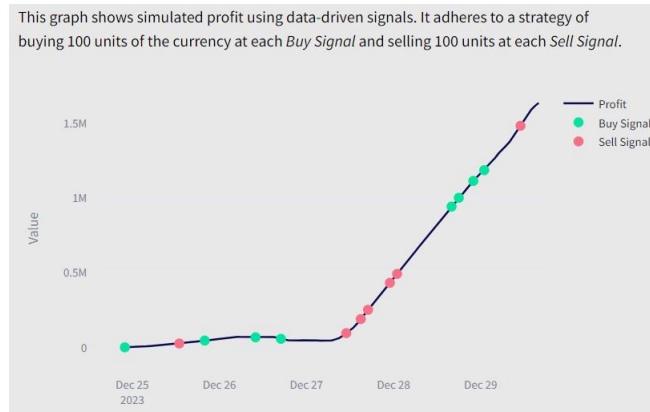
        # Create a Figure object with the defined data and layout
        fig = go.Figure(data=data, layout=layout)

        # Set the height and width of the figure
        fig.layout.height = 350
        fig.layout.width = 650
        return fig # Return the Figure object for plotting

    # Print an error message if an exception occurs and return an empty Figure object
    except Exception as e:
        print(f"An error occurred while creating the profit chart: {e}")
        return go.Figure()

```

El gráfico de la evolución del beneficio obtenido, se vería como sigue en la aplicación de **streamlit**:



2.2 Cálculo del oscilador estocástico sobre una media móvil

Como hemos expuesto en el apartado anterior, la media móvil $%D$ se utiliza para suavizar las fluctuaciones del oscilador estocástico $%K$ y para la generación de señales de compra y venta en la estrategia descrita.

Otro enfoque para calcular un oscilador estocástico suavizado sería sustituir, en la propia fórmula de su definición, el precio de cierre de un intervalo con la media móvil del cierre (esto sería el indicador SMA). Esta posibilidad sería de fácil implementación en nuestro código, pues ya contamos con un campo para dicha media móvil y la sustitución en la fórmula sería directa. Así se vería el fragmento de código modificado:

```
# Retrieves trading data from the Kraken API and stores it in a Pandas DataFrame
@st.cache_data(ttl=300) # Decorator to cache the data in Streamlit for 5 minutes
def obtain_function(pair, interval, divisor, since, until):
    [...]

    ohlc_df['%K'] = (ohlc_df['SMA'] - ohlc_df['L14']) /
                    (ohlc_df['H14'] - ohlc_df['L14']) * 100

    [...]
```

2.3 Gráfico del indicador junto con la cotización del par calculado

Finalmente, damos al usuario la opción de graficar tanto la cotización como el oscilador estocástico $%K$ y su media móvil $%D$, en una única figura. Para ello, combinamos adecuadamente los gráficos generados por separado y creamos un menú para seleccionar la gráfica que se desea visualizar.

```
class Front:
    [...]

    # Method to generate and display graphs based on user-selected parameters
    def display_graph(self):

        # Horizontal menu for selecting the type of graph to display
        self.graph_selected = option_menu(None, ["Candlestick", "Stochastic",
                                                "Combined", "Strategy"],
```

```

        icons=['bar-chart-line', 'activity', "layers"],
        menu_icon="cast", default_index=0, orientation="horizontal")

if self.graph_selected != None:

    # Conditional to verify if self.currency_pair is of NoneType
    if self.currency_pair is None:
        st.markdown('&nbsp;'*30 + 'Please, select a &nbsp;*currency pair*&nbsp;  

            to graph the corresponding data', unsafe_allow_html=True)
        return # End the execution of this method

    # Conditional to verify if self.time_interval is of NoneType
    if self.time_interval is None:
        st.markdown('&nbsp;'*30 + 'Please, choose a &nbsp;*time interval*&nbsp;  

            to graph the corresponding data', unsafe_allow_html=True)
        return # End the execution of this method

graph = Graph(pair=self.currency_pair, interval=self.time_interval,
              divisor=find_largest_divisor(self.time_interval),
              since=self.since, until=self.until)

ohlc_df = graph.obtain_data()
candlestick, stochastic = graph.candlestick(ohlc_df), graph.stochastic(ohlc_df)

if self.graph_selected == "Candlestick":
    fig = candlestick

elif self.graph_selected == "Stochastic":
    fig = stochastic

elif self.graph_selected == "Combined":
    fig = make_subplots(rows=2, cols=1, shared_xaxes=True, vertical_spacing=0.1,
                        row_heights=[0.8, 0.2], specs=[[{"secondary_y": True}], [{}]])

    fig.add_trace(candlestick['data'][0], row=1, col=1, secondary_y=True)
    fig.add_trace(candlestick['data'][1], row=1, col=1, secondary_y=False)
    fig.add_trace(candlestick['data'][2], row=1, col=1, secondary_y=True)
    fig.add_trace(candlestick['data'][3], row=1, col=1, secondary_y=True)

    fig.add_trace(stochastic['data'][0], row=2, col=1)
    fig.add_trace(stochastic['data'][1], row=2, col=1)
    fig.add_trace(stochastic['data'][2], row=2, col=1)
    fig.add_trace(stochastic['data'][3], row=2, col=1)

    fig.update_layout(
        title='Candlestick Graph with Moving Average and  

            Stochastic Oscillator',
        yaxis3_title='%K - %D',
        xaxis_rangeflider_visible=False,
        height=450, width = 650)

```

```

elif self.graph_selected == "Strategy":
    profit_df = graph.calculate_profit(ohlc_df)
    fig = graph.profit_graph(profit_df)
    if fig is not None:
        st.write("This graph shows simulated profit using data-driven signals. " +
                 "It adheres to a strategy of buying 100 units of the currency at " +
                 "each *Buy Signal* and selling 100 units at each *Sell Signal*.")
    else:
        st.write("There are no buy signals")
    return

# Convert the figure to a dictionary for Streamlit to display
fig_dict = fig.to_dict()

# Use Streamlit to display the plotly graph
st.plotly_chart(fig_dict)

```

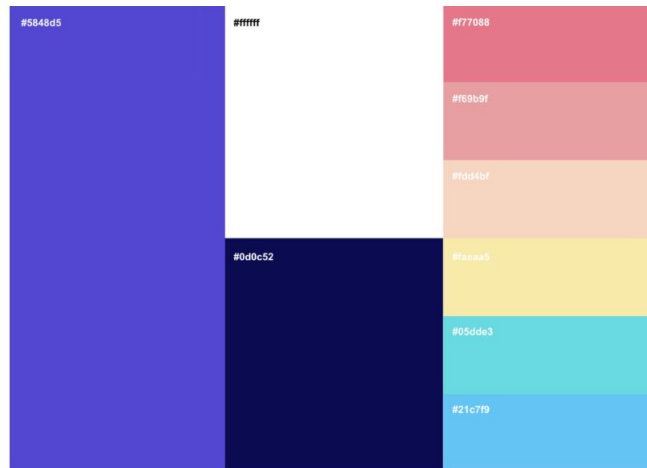
El menú con las opciones para que el usuario pueda seleccionar la gráfica que se mostrará en la interfaz y la combinación del gráfico de velas para la cotización junto con los diagramas de líneas para el oscilador estocástico, se verían como sigue en la aplicación de `streamlit`:



3 Estructuración

Nuestro proyecto necesita principalmente de tres ficheros `.py` para funcionar: `graphs.py`, `front.py` y `main.py` (que se encarga de ejecutar la aplicación de `streamlit` con las funcionalidades creadas). Además, hemos añadido el fichero `style.py` en el que hay una función `def style()` que sirve para crear el formato inicial de la aplicación. Esta función se encarga de crear un encabezado con links a la página oficial de Kraken, el repositorio de GitHub en que se encuentran todos los ficheros creados y las páginas correspondientes de nuestro proyecto tanto en PyPI como en Docker. Además, este fichero se encarga de ajustar los márgenes de la página y ocultar los elementos predeterminados de las aplicaciones de `streamlit`.

Por otro lado, hemos configurado la paleta principal de la aplicación desde el archivo `.streamlit/config.toml`. Para este objetivo hemos tomado como referencia la paleta que utiliza la propia página de Kraken.



El contenido del archivo `.streamlit/config.toml` tendría la siguiente forma:

```
[theme]
primaryColor = "#5848d5"           # Replace with your chosen primary color
backgroundColor = "#E8E8E8"        # Replace with your chosen background color
secondaryBackgroundColor = "#ABA3EA" # Replace with a secondary color
textColor = "#000000"              # Replace with your chosen text color
```

De esta manera, conseguimos generar una aplicación de `streamlit` con las funcionalidades descritas en los apartados anteriores, pero teniendo en cuenta también el apartado estético de la interfaz con la que va a interactuar el usuario final. Así se vería el apartado gráfico de la aplicación de `streamlit` que hemos creado:

3.1 Funciones y utilización de clases

Como ya se ha comentado con anterioridad, el proyecto se estructura en varios módulos y clases para facilitar la organización y la reutilización del código. Contamos con dos clases: **class** `Graph` en `graphs.py`, para la obtención de datos a través de la consulta a la API de Kraken usando la librería `krakenex` y la generación de las gráficas a través de la librería `plotly` y, por otro lado, la **class** `Front` en `front.py` que se encarga de generar los widgets necesarios para la interacción con el usuario, la recogida de los valores seleccionados y la creación de instancias de la **class** `Graph` para mostrar las figuras generadas por pantalla adecuadamente.

A continuación, mostramos de manera superficial las funciones y clases de los ficheros del proyecto.

graphs.py

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

# Retrieves trading data from the Kraken API and stores it in a Pandas DataFrame
@st.cache_data(ttl=300) # Decorator to cache the data in Streamlit for 5 minutes
def obtain_function(pair, interval, divisor, since, until):
    [...]

# The class Graph is designed for constructing all the different graphs for this project
class Graph:

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

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

    # Static method to create a candlestick chart from OHLC data using Plotly
    @staticmethod
    def candlestick(df):
        [...]

    # Calculate and graph the stochastic oscillator and its mobile mean
    @staticmethod
    def stochastic(df):
        [...]

    # Function to calculate the profit from trading based on buy and sell signals
    def calculate_profit(self, df):
        [...]

    # Function to create a profit graph from a DataFrame containing buy and sell signals
    def profit_graph(self, df):
        [...]
```

style.py

```
# Function to set the page container style
def set_page_container_style( max_width: int = 1500, max_width_100_percent: bool = False,
                             padding_top: int = 1, padding_right: int = 10,
                             padding_left: int = 1, padding_bottom: int = 10 ):
    [...]

def style():
    [...]
```

front.py

```
[...]
# Import the style function from the 'style' module to customize the app
from style import style

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

# Finds the largest duration in 'options' that is a divisor of n
def find_largest_divisor(n):
    [...]

# The class Front is designed to construct the frontend of the Streamlit application
class Front:

    def __init__(self):
        style()
        [...]

    # Method to create user input interfaces, including dropdowns and buttons
    def select_boxes(self):
        [...]

    # Method to generate and display graphs based on user-selected parameters
    def display_graph(self):
        [...]

    # Method to execute the core operations of the Streamlit application
    def run(self):

        try:
            # Invoke methods to display user input options and the selected graph
            col1, _, col2 = st.columns([100,5,95])

            with col1:
                # Displays currency pair and time interval selection options
                self.select_boxes()
```

```

        with col2:
            # Renders the graph based on user selections
            self.display_graph()

# Handle and display any exceptions that occur during execution
except Exception as e:
    # Show the error message to the user in the app
    st.error(f"An error occurred: {e}")

```

main.py

```

from front import * # Import all components from the 'front' module

# Ensure this script runs only when executed directly, not when imported as a module
if __name__ == "__main__":
    front = Front() # Instantiate the Front class from the 'front' module

    # Execute the main functionality of the Front instance, initiating the Streamlit app
    front.run()

```

3.2 Manejo de errores y excepciones

En este proyecto, la gestión de errores se realiza principalmente a través de la estructura de control de flujo `try-except`. Esta metodología nos permite identificar y tratar posibles fallos que surjan durante la ejecución del programa. Un ejemplo se observa en la función `def profit_graph(self, df)` que, dados los atributos de una instancia de la `class Graph`, crea el gráfico de rendimientos de la estrategia definida. Si se identifica algún problema, el sistema muestra un mensaje de error en la consola y devuelve un gráfico sin datos.

```

class Graph:
    [...]

    # Function to create a profit graph from a DataFrame containing buy and sell signals
    def profit_graph(self, df):
        try:
            [...]
            # Create a Figure object with the defined data and layout
            fig = go.Figure(data=data, layout=layout)

            # Set the height and width of the figure
            fig.layout.height = 350
            fig.layout.width = 650

            return fig # Return the Figure object for plotting

        # Print an error message if an exception occurs and return an empty Figure object
        except Exception as e:
            print(f"An error occurred while creating the profit chart: {e}")
            return go.Figure()

```

En este otro caso, utilizamos la función `st.error()` de `streamlit` para mostrar un mensaje de error en nuestra aplicación web. Esta herramienta es de gran utilidad en el desarrollo de aplicaciones con `streamlit`, ya que simplifica significativamente el proceso de detectar y solucionar errores.

```
class Front:
    [...]

    # Method to execute the core operations of the Streamlit application
    def run(self):
        try:
            [...]

        # Handle and display any exceptions that occur during execution
        except Exception as e:
            # Show the error message to the user in the app
            st.error(f"An error occurred: {e}")
```

En el ámbito del manejo de errores, se tiene un escenario particular al realizar la consulta a la API de Kraken en la función `def obtain_function(pair, interval, divisor, since, until)` de `graphs.py`. El resultado de la consulta es un conjunto de datos en formato JSON con dos campos: `'error'` y `'result'`.

Debido a ello, introducimos una lógica condicional para observar si el campo `'error'` es vacío y, en caso contrario, se imprime en la consola el mensaje que contiene. Además, se lanza una excepción con dicho mensaje para que sea recogida por el bloque `try-except` en el que se encuentran las acciones descritas.

```
# Retrieves trading data from the Kraken API and stores it in a Pandas DataFrame
@st.cache_data(ttl=300) # Decorator to cache the data in Streamlit for 5 minutes
def obtain_function(pair, interval, divisor, since, until):

    # 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, interval and start date
        response = k.query_public('OHLC',{ 'pair':pair, 'interval':divisor, 'since':since})

        # Check and raise an exception if an error exists 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

    [...]
```

4 Puntuación Extra

En esta sección adicional, detallaremos las pruebas de software y cobertura realizadas, la forma en que se han facilitado los mecanismos para la reproducción del entorno virtual y el proceso que se ha seguido para llevar a cabo la distribución del proyecto en plataformas como PyPI y Dockerhub.

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

En el marco de este proyecto, hemos implementado pruebas de software de tipo unit-testing e integration-testing en el fichero `tests.py`, utilizando la librería `unittest`. Esta herramienta nos ha facilitado la creación de clases de prueba, cada una con métodos que corresponden a pruebas específicas. A continuación, presentamos una muestra con varias de las pruebas que han sido integradas exitosamente en nuestro proyecto.

La función `def test_get_kraken_pairs(self)` comprueba que, al realizar una llamada a la función del fichero `front.py`, `def get_kraken_pairs()`, y recoger su resultado, se obtiene una tupla no vacía.

```
# Definition of a test case class for the 'front' module
class TestFront(unittest.TestCase):
    [...]

    # Test method to test the get_kraken_pairs function
    def test_get_kraken_pairs(self):
        # Calling the get_kraken_pairs function and storing its result
        result = get_kraken_pairs()

        # Checking if the result is a tuple and has a length greater than 0
        self.assertIsInstance(result, tuple)
        self.assertTrue(len(result) > 0)
```

La función `def test_find_largest_divisor(self)` comprueba el resultado, para varios números específicos, de la función `def find_largest_divisor(n)` del fichero `front.py`.

```
# Definition of a test case class for the 'front' module
class TestFront(unittest.TestCase):
    [...]

    # Testing the find_largest_divisor function with different inputs
    def test_find_largest_divisor(self):
        self.assertEqual(find_largest_divisor(60), 60)
        self.assertEqual(find_largest_divisor(61), 1)
        self.assertEqual(find_largest_divisor(120), 60)
```

La siguiente prueba de software se basa en la idea de comprobar que se verifica cierta propiedad que la función que se desea evaluar debería cumplir teóricamente. En este caso, estamos trabajando para comprobar la corrección de la función `def find_largest_divisor(n)`, se trata de la igualdad de los valores al aplicar la función sobre un producto $a \times b$, y al realizar el mínimo común múltiplo (`def lcm(a, b)`) con los resultados de aplicar la función a los valores de a y b , respectivamente. Esta idea que acabamos de exponer ha sido implementada para varios pares de números naturales en `def test_largest_divisor_lcm(self)`.

```

# Definition of a test case class for the 'front' module
class TestFront(unittest.TestCase):
    [...]

    # Testing that find_largest_divisor verifies a formula given different inputs
    def test_largest_divisor_lcm(self):
        test_pairs = [(6, 8), (15, 25), (9, 14)] # Sample pairs of numbers to test
        for a, b in test_pairs:
            product = a * b
            largest_divisor_product = find_largest_divisor(product)
            lcm_largest_divisors = lcm(find_largest_divisor(a), find_largest_divisor(b))

            self.assertEqual(largest_divisor_product, lcm_largest_divisors)

```

Tanto para la `class Front` como para la `class Graph` se han realizado pruebas sencillas para comprobar la correcta inicialización de las instancias. Por otro lado, hemos comprobado que al utilizar la función `def obtain_data(self)` de `graphs.py` (con los valores predeterminados) se obtiene un dataframe de `pandas` no vacío. Además, para varias de las distintas funciones que crean gráficas (`candlestick`, `stochastic` y `profit_graph`) hemos comprobado que efectivamente se genera una figura de `plotly`.

Veamos un ejemplo de estas pruebas de software con la implementación para `def candlestick(df)`.

```

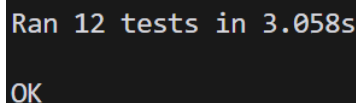
# Definition of a test case class for for the 'graphs' module
class TestGraph(unittest.TestCase):
    [...]

    # Testing the candlestick method of the Graph class
    def test_candlestick(self):
        graph = Graph(pair='XETHZUSD', interval=1440, divisor=1)
        df = graph.obtain_data()
        fig = graph.candlestick(df)
        self.assertIsInstance(fig, go.Figure))

```

Por último, hemos realizado varias pruebas para la función `def aggregate_intervals(interval, df)` de `graphs.py`. En ellas, hemos definido un dataframe de `pandas` para comprobar que al agregar los intervalos se mantienen las columnas del dataframe y que al introducir un valor no válido se recoge un error.

Para ejecutar las pruebas, se pueden utilizar dos comandos distintos: `python -m unittest` o `python tests.py`, desde el directorio principal del proyecto. Al completar este proceso, en la consola se mostrará un mensaje que confirma que el programa ha superado con éxito las pruebas de software.



```

Ran 12 tests in 3.058s
OK

```

De igual manera, para recopilar los datos sobre la cobertura del código durante la ejecución de las pruebas se pueden utilizar dos comandos distintos: `coverage run -m unittest` o `coverage run tests`. Tras ejecutar uno de estos comandos, podemos generar un reporte como el siguiente, usando el comando `coverage report`, con el que se puede ver el porcentaje de líneas recorridas de cada fichero durante la ejecución de las pruebas.

Name	Stmts	Miss	Cover
front.py	112	79	29%
graphs.py	117	21	82%
style.py	14	1	93%
tests.py	81	0	100%
TOTAL	324	101	69%

Durante la formulación de las distintas pruebas de software, hemos decidido limitar el alcance de las pruebas para la `class Front`. Dado que su función principal se centra en la interacción con el usuario y no en la lógica de negocio o en las funcionalidades clave, consideramos que un enfoque extensivo en las pruebas para esta clase no es esencial. No obstante, para garantizar la transparencia y facilitar un seguimiento adecuado, la clase ha sido incluida en los informes de cobertura.

4.2 Facilitar los mecanismos para la reproducción del entorno virtual

Para integrar Poetry en el proyecto, se ha utilizado el archivo `pyproject.toml`, ubicado en la raíz del proyecto. Este archivo es esencial para Poetry, ya que gestiona las dependencias del proyecto. Dentro de `pyproject.toml` se detallan las dependencias necesarias, la versión de Python requerida y otros metadatos relevantes del proyecto. Los contenidos de este fichero se muestran a continuación:

```
[tool.poetry]
name = "KrakenPythonMarcosRodrigo"
version = "0.1.2"
description = "A streamlit tool for rendering kraken data"
authors = ["Your Name <you@example.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.11"
plotly = "5.18.0"
streamlit = "1.29.0"
coverage = "5.5"
pandas = "2.1.4"
numpy = "1.26.2"
krakenex = "2.1.0"
streamlit-option-menu = "0.3.6"
protobuf = "3.20.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Para reproducir el entorno virtual utilizando Poetry, se deben seguir los siguientes pasos:

1. Instalación de Poetry. Puedes hacerlo siguiendo las instrucciones en la documentación oficial de Poetry.
2. Clonar el repositorio del proyecto.

3. Navegar hasta el directorio del proyecto.
4. Ejecutar el comando `poetry install`. Este comando instalará todas las dependencias especificadas en el archivo `pyproject.toml`.
5. Para activar el entorno virtual creado por Poetry, ejecuta el comando `poetry shell`.

4.3 Reproducción del entorno con pip

Puedes instalar manualmente el entorno con los siguientes pasos:

1. Creamos un entorno virtual: `python -m venv krakenpythonmarcosrodrigo`
2. Activamos el entorno: `source krakenpythonmarcosrodrigo/bin/activate`
3. Instalamos los requisitos: `pip install -r requirements.txt`

Después de instalar los requisitos ya puedes ejecutar el proyecto con el comando `streamlit run main.py`.

4.4 Distribución del proyecto a través de PyPI

1. Primero se han instalado las dependencias de desarrollo Hatch y Twine.

```
$pip install hatch  
$pip install pip twine
```
2. Se crea una cuenta en PyPI, adicionalmente se necesita activar la autenticación de doble factor y se debe utilizar el token de la API en Twine.
3. Para crear el distribuible se utiliza el comando: `$hatch build`.
4. Para publicar el paquete se utiliza el comando: `$twine upload dist/*` (con tu cuenta de PyPI).
5. Una vez que el paquete esté publicado en PyPI, otros pueden instalarlo utilizando `$pip install`.

4.5 Distribución del proyecto a través de DockerHub

Para crear la imagen que se ha subido a DockerHub se han seguido los siguientes pasos:

1. Creamos un Dockerfile que describa el contenedor:

```
# Use an official Python runtime as a parent image  
FROM python:3.11-slim-buster  
  
# Set the working directory in the container to /app  
WORKDIR /app  
  
# Add the current directory contents into the container at /app  
ADD . /app
```

```
# Install Poetry
RUN pip install poetry

# Use Poetry to install dependencies
RUN poetry install

# Make port 8501 available to the world outside this container
EXPOSE 8501

# Run main.py when the container launches
CMD ["poetry", "run", "streamlit", "run", "main.py"]
```

2. Construimos la imagen de Docker con: `$docker build -t dixrow/krakenpythonmarcosrodrigo .`
3. Creamos una cuenta de Docker y la enlazamos con: `$docker login`
4. Subimos la imagen que hemos creado con: `$docker push dixrow/krakenpythonmarcosrodrigo:latest`
5. Para bajarse la imagen de DockerHub se usa: `$docker pull dixrow/krakenpythonmarcosrodrigo`
6. Para ejecutar la imagen se usa: `$docker run -p 8501:8501 dixrow/krakenpythonmarcosrodrigo`
7. Por último se puede acceder a través del navegador con el enlace: 127.0.0.1:8501