Technical Specification

Cryptocurrency Price Predictor

Vincent Achukwu (17393546) & Joseph Lyons (16485216)

Supervisor: Dr. Martin Crane

Submission Date: 24/04/2022

Table of contents

Abstract

Our project is a cryptocurrency price predictor web application that allows users to view future price predictions for selected cryptocurrencies as well as view the market sentiment, search crypto news, convert between cryptocurrencies and fiat currencies, view price movements in different graphs formats, add coins to their portfolio, and display the fear and greed index. This application serves the purpose of providing users with various tools to help them determine what trades they should make based on the performance of the cryptocurrency market. Not only does this application predict future prices of cryptocurrencies, but it allows users to view the market sentiment to better understand and predict the future outcome when entering positions for their investments.

1. Introduction

Upon opening the web application, the user will not have to log in/sign up. We decided not to require users to create an account for the app since there is nothing in the app that would require the user to have an account and save their preferences. Although the portfolio feature may require something like this, we managed to figure out how to save user preferences via Flask sessions.

The home page of the application displays a cryptocurrency converter with shortcuts to other app features as well as a market overview section displaying real-time price changes and other market statistics. The app has a future price predictor feature where the user can specify a cryptocurrency and the number of days into the future to predict. It then displays a line chart along with a buy/sell signal based on the prediction. Given that the crypto market is volatile, we also implemented market sentiment features to give the user a better understanding of what the overall sentiment is with the market (bullish or bearish). The Twitter sentiment feature allows users to specify a cryptocurrency and display a graph describing whether the sentiment for that selected coin is positive, negative, or neutral. The app also allows users to specify which coins to view in the portfolio section which summarises market statistics based on the selected coins, and users can also use the search bar to search for crypto news. Lastly, the fear and greed index section of the app displays the fear and greed index for the current day, the previous day, the previous week, and the previous month. We tried to provide the user with as much relevant market sentiment data as possible rather than just relying on the predictions. The app also has a footer tape that displays real-time prices for the top 25 cryptocurrencies along with their 24-hour percentage change.

The navigation throughout the web application was made to be as easily navigable as possible by having back buttons to allow users to go back to the appropriate page depending on what part of the app they are on, as well as shortcut buttons on the home page. The navbar also has access to the various features of the app, but we included the shortcut buttons on the home page too if the user operates the application on a smaller device as it would lead to the navbar collapsing and adds an extra layer of navigation.

The frontend was designed using HTML, CSS, and JavaScript, and the backend was developed using Flask in Python. The frontend communicates with the Flask API backend by sending inputs to the Flask API endpoints, after which the Flask route is called for that endpoint and those inputs are handled. Once the backend script gets a result (whether it's calling another API or predicting future prices for a selected coin and classifying the crypto data for the frontend to read), it uses the render_template function to display the appropriate HTML page with any parameters needed for the frontend to read via JINJA.

2. Motivation

Our motivation for this project was based on the rising trend of cryptocurrencies last few years. With the elevation of decentralised systems, there has been significant growth in the number of people joining the cryptocurrency market. Therefore, we thought we would take this opportunity to develop a cryptocurrency web application to allow users to follow the market trend and to obtain a prediction for a specified cryptocurrency they would like to see predictions for. We designed this application in a simple and easy-to-use user interface that would also accommodate users who are new to the cryptocurrency space and will not be overwhelmed by the technical analysis side of things. Initially, we thought it might have been a good idea to include the stock market in this app too. This may have accommodated a more diverse group of traders who are both into stocks and crypto, however, we wanted to keep things simple and stick with one field rather than mixing things around.

We wanted to make something that would accommodate traders who are in the cryptocurrency market and aim to maximise profits based on the indicated price prediction. We realised how volatile the market is, hence, we wanted to emphasise how this application is not for financial advice, but rather a way to measure the future trend and provide users with a good indication of bullish or bearish signals which could guide traders for their next trade. We decided to make this into a web app rather than a mobile app since it is more feasible to navigate between tabs to stay up-to-date with cryptocurrency news, exchanges, and sources while having our app open rather than having to struggle navigating between different apps on a mobile device.

3. Research

Throughout the development of this project, there was quite a bit of research involved as we were working with areas we haven't, or not as often, touched on before. We mainly relied on Google when researching areas related to cryptocurrencies and what type of features would typically be present in other similar existing applications. We haven't developed an application fully relying on machine learning before, but our supervisor gave us very helpful ideas and advice with regards to how to approach this project. Initially, we thought about going with the LSTM model, but our supervisor advised us to look into the ARIMA model. Prior to focusing on the frontend side of things, we had to figure out how we could have the application work for predicting multiple cryptocurrencies rather than just Bitcoin for instance. We had to figure out how to make the app train on not just one cryptocurrency dataset, but on many. We figured that the ARIMA library in Python has an auto arima function that determines the best p, d, and q values based on the dataset and makes a prediction based on them.

We also had to research areas revolving around how to obtain tweets, obtain real-time cryptocurrency prices, implement search functionality for reading crypto news articles, and how to save user-selected cryptocurrencies when storing the coins in the user's portfolio during a session. We also tried to stay up to date with trends that are happening in the

crypto world. We looked at how high profile people could affect the price fluctuation of certain coins. For example, if Elon Musk tweeted about a certain coin, it would have massive effects on the price movement of the coin. This made it very unpredictable as to where the prices of the coins would go so we decided to have our app more of a guidance of when to buy and sell.

4. Design

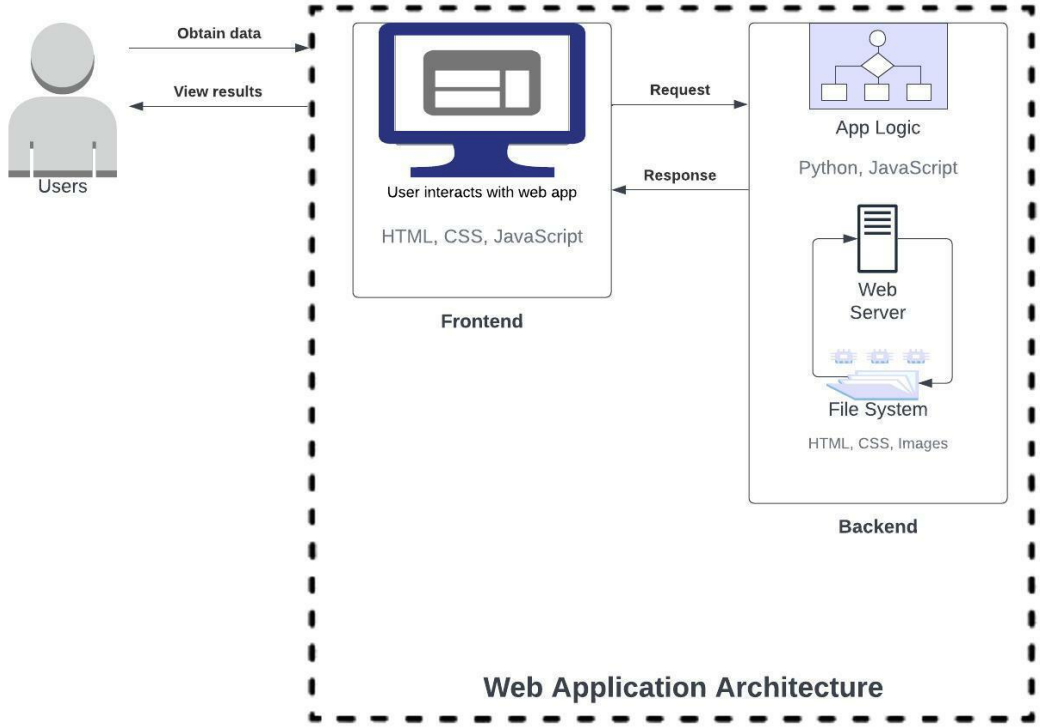
Our project was developed through a Python Flask server for the backend, and HTML, CSS, and JavaScript for the frontend. We aimed to keep the frontend design fairly simple and easy to use since this project mainly focuses on the technical and sentiment aspects of cryptocurrencies and aimed to accommodate users who are also new to this space. We came up with a simple name for the app, CoinCast, which suggests that this application is intended for cryptocurrency coin forecasting. We added a simple logo to the app also which is present in the tab on the browser. We chose a simple background image as well as a consistent and formal colour theme for the navbar. Since this was developed with Flask, we chose to add anything that should be present in all pages to base.html which can then be extended to other HTML pages that should include the frontend features too. For instance, the navbar, background, and the price ticker-tape are present across all pages of the application. We used Bootstrap CSS for adding better frontend functionality and improving the design of the application, for instance, the jumbotron div can be seen throughout the application which has a white background with some slight transparency to blend into the background image of the application.

For maintaining the navigation throughout the application, we included back buttons when the user decides to view cryptocurrency predictions or the market sentiment so as to prevent the user from having to click on the navbar link or click the back button on the browser. Flask also has a sessions library that can store and save variables during a session. We incorporated this into our application for the portfolio section so that the user will not have to log in or sign up in order to save their portfolio preferences to a database.

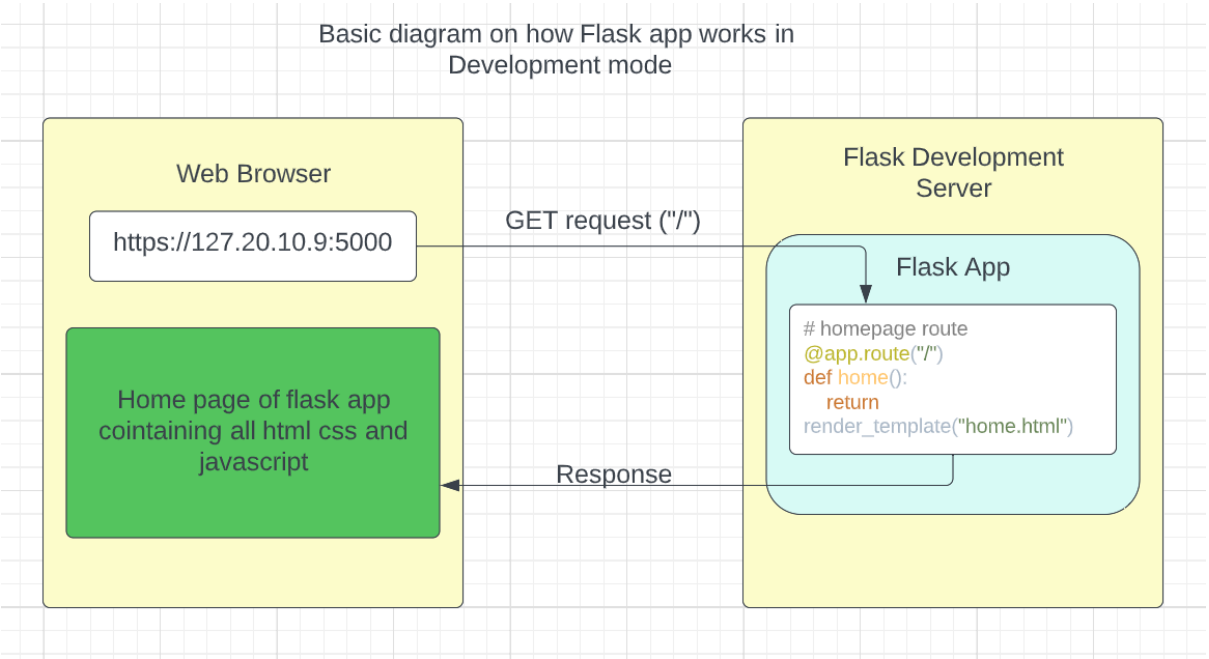
Upon searching for cryptocurrency news, by default, the application will display the most relevant search results. The user is welcome to apply search filters based on the current search query such as “most relevant”, “most popular”, and “latest”. We decided to restrict the news results to 100 results since we felt that this would be enough results based on the users’ search query. Instead of showing the exact date of when a news article was published, we decided to use the timeago library in Python. Essentially, instead of showing dates in a date format, it will show something along the lines of “x days ago” depending on how long ago something was published. This makes it more readable when scrolling through search results.

The diagram below shows a high-level architecture diagram of the interactions taking place between the user and the application.

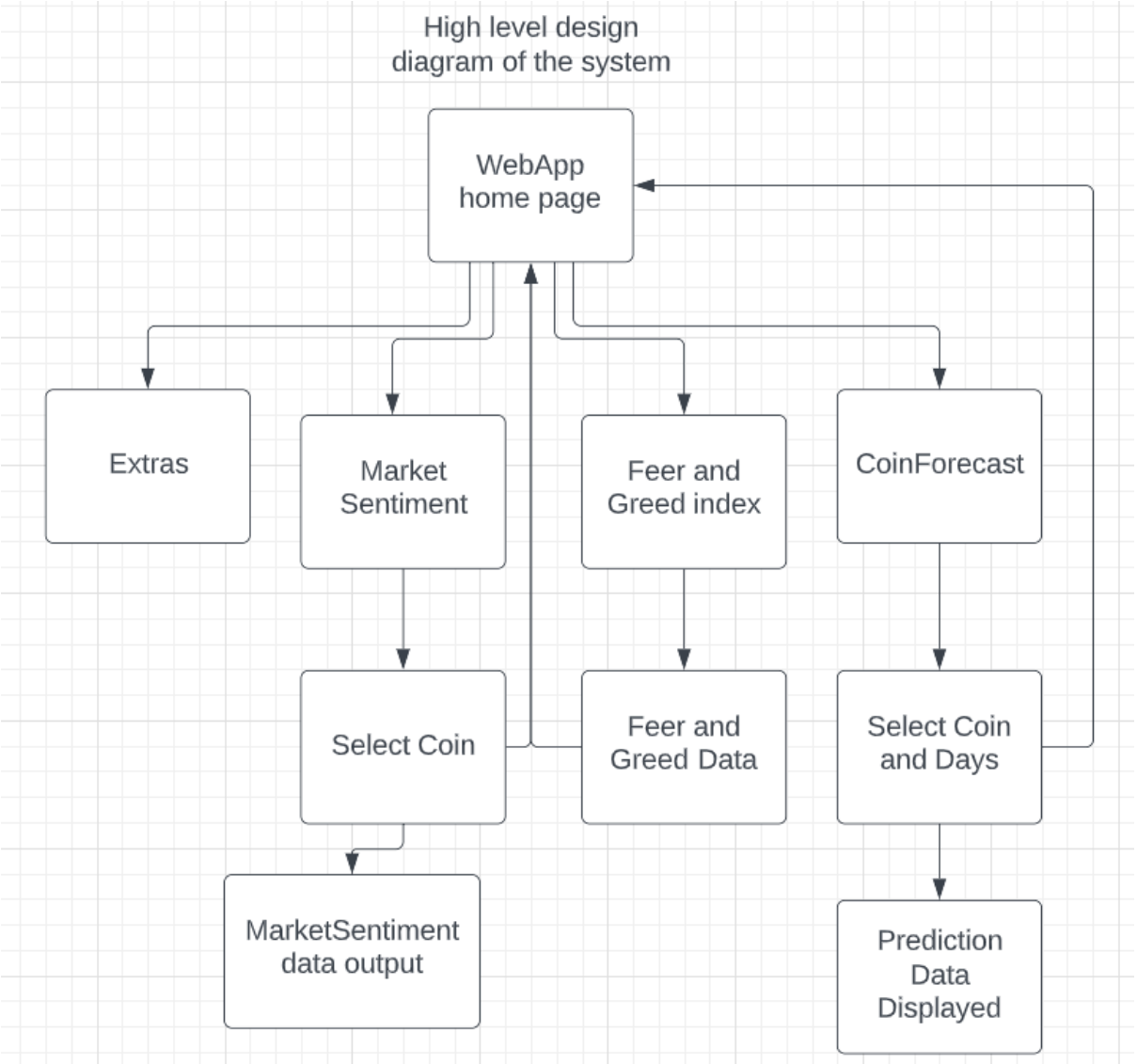
4.1 Architecture Diagram



As seen above, the user accesses the web application through a browser and upon querying for some data (whether it's obtaining predictions, market sentiment, displaying the fear and greed index, etc), the frontend inputs are passed to the backend where the application logic is handled. Depending on the request, the backend could be communicating with another API to obtain data based on the user's inputs, or it could be classifying the data if the user wants to view the predictions for a certain cryptocurrency. After that, it sends data to the frontend and the page is updated.



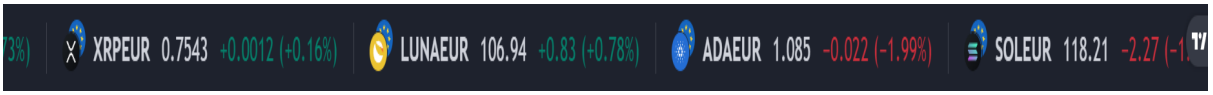
A basic example of how the web application works using the example of the homepage.



As you can see from the above diagram, there are very distinct paths the user can take. Each command leads to the next HTML page with the option to go back to the home page with the use of a back button.

4.2 Frontend Design Choices

In terms of the front end design choices, We decided to keep the theme of the web app simple. Each page would be very clear on what is being displayed. With the use of a base.html file, this was made simple. With the use of the bootstrap library and pre-made GUIs, we decided on the theme to be simple but crisp. The use of the dark grey header with white writing makes it easily readable for all groups of people. We originally had a different colour scheme which did not look as good and was not as easy for the user to view. Next, we decided to add a price ticker to the footer of each page, we thought this would emphasise the concept of our app as being based on cryptocurrencies. This was a simple way to allow the user to have an easy way to view the current price of whatever coin they are interested in.



Making the app easy to navigate played a key role when designing the front end design. Allowing the user to easily view and navigate to the part of the web app they desire to use is crucial in the front end design. The navigation bar at the top of each page clearly states which parts of the app you are going to, making sure when each page is loaded that the header changes accordingly. Also, making sure that each button is routed to the right HTML page allowed for the web app to run smoothly.

In terms of displaying the data, we create we came into issues with firstly the market sentiment data. We initially were just displaying a static image of a graph of the tweets. This was not appealing to the user. We decided to display the data in an interactive javascript graph which we talked about in part 5 implementation. This allows the user to clearly see with the use of a table and pie chart the market sentiment from Twitter. When the pie chart was initially made the colour scheme did not reflect the data being displayed. We decided to display the positive and negative segments in green and red. This allows for the user to easily identify which is positive and negative, as colour can have a significant impact on our perceptions and emotions.

5. Implementation

Programming Languages:
We developed our app using python as it has a large library of additional packages. It is a relatively lightweight programming language, which would make the experience of the user on older hardware more tolerable and would not hinder the user's interactions with the web application. We also used chart.js, a javascript library to display some of our data on the HTML pages. This made them more reader-friendly for the user.

Language	Version	Usage
Python	3.9	Interface, Backend code
JavaScript		Charts
HTML,CSS	HTML5	Front end design and implementation

Initially, for the backend, we were testing and playing around with various ways of going about the implementation of the cryptocurrency price prediction by searching for example

implementations online. In the end, we decided to go with the ARIMA model given that our supervisor advised us to implement this approach for predicting cryptocurrency prices. We saved additional Python files to the res directory to show how we initially went about predicting cryptocurrency prices via the ARIMA model along with how we read the tweets using the tweepy library in Python.

```python
# determining if the data is stationary or not
from statsmodels.tsa.stattools import adfuller
def adf_test(dataset):
    dftest = adfuller(dataset, autolag="AIC")
    print("1. ADF: ", dftest[0])
    print("2. P-Value: ", dftest[1])
    print("3. Num Of Lags: ", dftest[2])
    print("4. Num Of Observations Used For ADF Regression:", dftest[3])
    print("5. Critical Values:")
    for key, val in dftest[4].items():
        print("\t", key, ": ", val)

# # if p < 0.05 it is stationary, else it's non-stationary
adf_test(df["close"])
# P-Value: 0.9862380601118591 -> therefore it's non-stationary, need to make it stationary

# differencing once to make it stationary
df["Close First Difference"] = df["close"] - df["close"].shift(1)
adf_test(df["Close First Difference"].dropna())
plt.title("Close First Difference")
df["Close First Difference"].plot()
plt.show()
# now P-Value: 1.2770080148370808e-10 -> therefore it's stationary

# using auto_arima to determine the best model from the order values (p, d, q)
#   - p = AR model lags (best done with PACF)
#   - d = differencing
#   - q = MA lags (best done with ACF)
stepwise_fit = auto_arima(df['close'], trace=True, suppress_warnings=True)
print(stepwise_fit.summary())
# result we get is (2, 1, 2)

# differencing the dataset for PACF and ACF tests
fig = plt.figure(figsize=(12, 8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(df["Close First Difference"].iloc[2:], lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(df["Close First Difference"].iloc[2:], lags=40, ax=ax2)
plt.title("Close First Difference")
plt.show()
```

This code snippet in the image above (located in res/coinForecast.py) is what we had initially when testing the ARIMA model to better understand how it works and how one would determine the best P, D, and Q values for making a prediction on the dataset. P represents the AR model lags (best done with PACF), D represents differencing, and Q represents the MA (moving average) lags (best done with ACF). It requires a stationary dataset, so by running through some tests, you can view the various graphs such as the ACF and PACF to determine what the best P, D, and Q values should be. ACF (autocorrelation) estimates and

plots the average correlation between data points in a time series data and previous values of the series measured for different lag lengths. PACF (partial autocorrelation) on the other hand is similar to an ACF with the exception that each partial correlation controls for any correlation between observations of a shorter lag length. It finds a correlation between the residuals.

In the end, in the prediction() function of our code in app.py, we implemented it such that the user from the frontend selects a coin and the number of days to predict from the drop-down.

```python
# flask route for displaying the graph for the selected cryptocurrency
@app.route("/prediction", methods=["GET", "POST"])
def prediction():
    # clearing the graph prior to showing the predictions for selected coin
    plt.clf()
    if request.method == "POST":
        # obtaining the selected coin from the dropdown
        coinName = request.form.get("selectedCoin")
        daysSelected = int(request.form.get("daysSelected"))
        # using dictionary to get the corresponding ticker (for readability on the graph e.g Bitcoin (BTC))

        coinTicker = crypto[coinName]
        # retrieving the dataset from TIINGO
        df = getCryptoData(coinTicker)
        # retrieving prediction via passing dataset to predict function
        prediction = predict(df, daysSelected)
        # obtaining the current price and the predicted price to determine buy/sell signal
        currentPrice = df.close[-1]
        predictedPrice = prediction.values[-1]
        percentageDifference = "{:.4f}".format((abs(currentPrice - predictedPrice) / currentPrice) * 100)

        # configuring the plot via matplotlib for now (planning to upgrade to JavaScript chart)
        ax = df["close"].plot(color="blue", label="Close")
        preds = prediction.plot(ax=ax, color="red", label="Predicted Close").get_figure()
        plt.title("{} ({}-EUR) Prediction".format(coinName, coinTicker))
        plt.legend()

        # saving the plot so it can be passed to the frontend
        plt.savefig("static/images/predictionPlot.png")

        # passing the plot to coinForecastGraph.html
        return render_template("coinForecastGraph.html", coinName=coinName, currentPrice=currentPrice, predictedPrice=predictedPrice, percentageDifference=
```

This data is then read via Flask requests and the historic price of that selected coin is obtained from the Tiingo API.

```python
# helper function for getting historic price of selected cryptocurrency via TIINGO API
def getCryptoData(coinTicker):
    # setting default crypto-fiat currency pair to EUR
    againstCurrency = "EUR"

    # specifying an early start date for the selected coin so the API will retrieve the earliest start
    # date available for historic prices if it doesn't match the specified start date
    start = dt.datetime(2015, 1, 1)
    # end date is the current date
    end = dt.datetime.now()

    data = web.get_data_tiingo("{}{}".format(coinTicker, againstCurrency), start, end, api_key=TIINGOKEY)
    # dropping any N/A values in case
    data = data.dropna()
    # removing multi-index via resetting the index to just the date column
    data = data.reset_index()
    data = data.set_index("date")
```

Once the dataset is obtained, it begins determining the best ARIMA model parameters (p, d, and q) via the auto_arima() function. We used the auto_arima() function since results may not have been that great if we hardcoded the same parameters for every cryptocurrency. Therefore, the auto_arima() function would try to determine the best p, d, and q values based on the coin selected.

To display the graphs on the front end of the web application we figured out an easy way to send over live data graphs made through mathplot and BTYEIO.

```
    plt.title("{} ({}-EUR) Close First Difference".format(coinName, coinTicker))

    buf = io.BytesIO()
    # saves photo to bytes
    preds.savefig(buf, format='png')
    buf.seek(0)
    buffer = b''.join(buf)
    # changes it to 64bit
    b2 = base64.b64encode(buffer)
    # allows it to be passed over to the front end using preds2
    preds2 = b2.decode('utf-8')
    # use of mpld3
    plt_html = mpld3.fig_to_html(preds)
```

Here you can see that it saves the photo “preds” in buf which turns it into bytes. Then using base64 format it allows the photo to be saves in 64 bytes. This allowed for the images to be easliy displayed on the front end of the application.

```
# helper function for obtaining the prediction for the selected coin
def predict(coinDataset, daysSelected):

    # running auto_arima on the selected coin closing price to get its best p, d, q values
    # predictions not looking so good long term, perhaps short term predictions would suit
    stepwise_fit = auto_arima(coinDataset["close"], trace=True, suppress_warnings=True, test="adf")
    order = stepwise_fit.get_params().get("order")

    # training the model now based on entire dataset to make future predictions
    model = ARIMA(coinDataset["close"], order=order)
    model = model.fit()
    # print(coinDataset.tail())  # checking what the last date is, then predict from this day onward

    # predicting from the current date onwards
    startDate = dt.datetime.now()

    # for testing, let's predict the x days into the future (depending on how many days user selected)
    indexFutureDates = pd.date_range(start=startDate.strftime("%Y-%m-%d"), end=(startDate + relativedelta(days=+daysSelected)).strftime("%Y-%m-%d"))
    prediction = model.predict(start=len(coinDataset), end=len(coinDataset) + daysSelected, typ="levels")
    # like before, we're handling dataset for indexing so we can plot it
    prediction.index = indexFutureDates
    # converting to pandas dataframe and creating columns (this MIGHT be needed for JavaScript chart)
    # predictionToDf = pd.DataFrame({"date": prediction.index, "predictedClose": prediction.values})
    # print(predictionToDf)

    return prediction
```

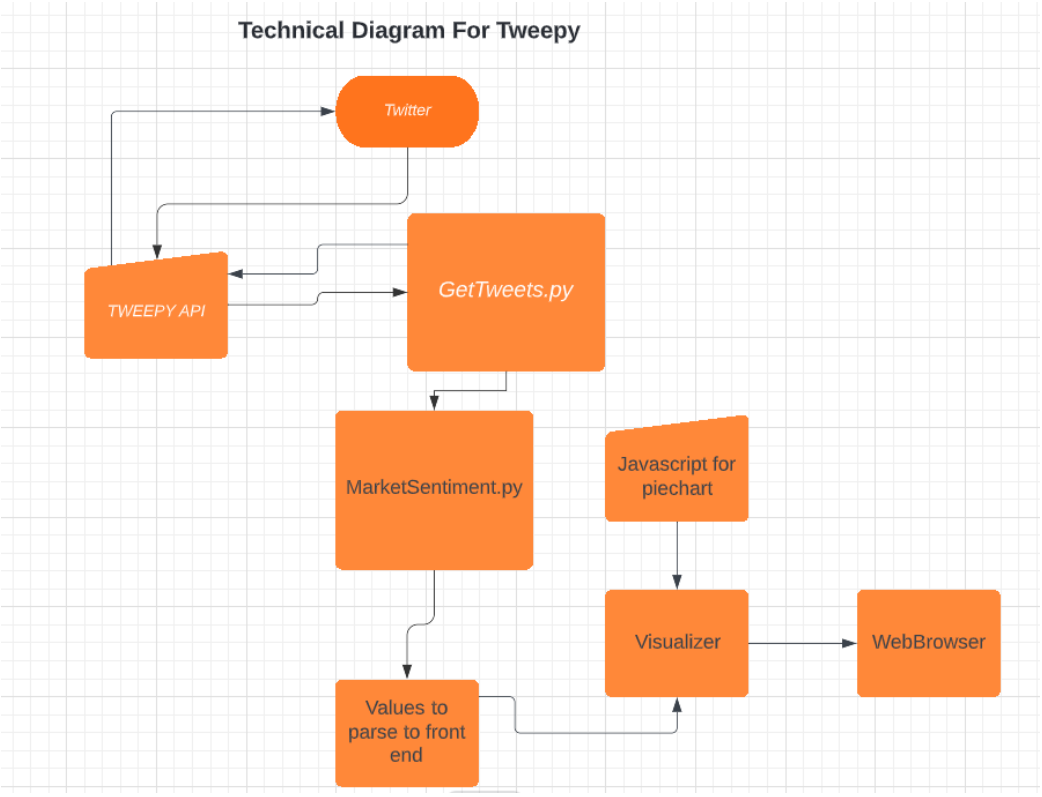
We also included the initial Twitter sentiment analysis code in the res directory to show how we went about it at first. The final version is of course in app.py. Similar to the prediction functionality, the user from the frontend would select the cryptocurrency from the dropdown menu that they would like to view the Twitter sentiment for and the selected coin would be passed to the getTweets() function.

```python
# market sentiment displayed in this route
@app.route("/sentiment", methods=["GET", "POST"])
def sentiment():

    if request.method == "POST":
        # obtaining the selected coin from the dropdown
        coinName = request.form.get("selectedCoin")
        # using dictionary to get the corresponding ticker (for readability on the graph e.g Bitcoin (BTC))
        coinTicker = crypto[coinName]
        # passing the coin name to tweepy to obtain its market sentiment
        positive, negative,  = getTweets(coinName)[0], getTweets(coinName)[1]
        neutral=abs(100-(positive+negative))
        # passing the plot to coinSentimentGraph.html
        return render_template("coinSentimentGraph.html", coinName=coinName, coinTicker=coinTicker,
                               positive=positive, neutral=neutral, negative=negative)

    cryptoCoins = list(crypto.keys())
    return render_template("coinSentimentSelector.html", cryptoCoins=cryptoCoins)
```

The tweets are obtained based on the user-selected coin by applying some filters to the tweets obtained from the Twitter API. Tweepy then takes the coin that is selected by the user and starts the filtering process. The filtering mechanism removes any tags, hashtags, and URLs which may deem the tweets to be irrelevant for the sentiment analysis. The polarity score is obtained from the tweets after which we determine if the polarity is positive or negative. This allows us to take the most recent 100 tweets and store them into two attributes either positive or negative. These totals are then parsed over to the front end so that they can be displayed within the pie chart. When parsed to the front end the numbers are stored within the contents table of the piechart, which then displays them.

```python
# helper function for obtaining tweets
def getTweets(coin):

    # looking at ordinary tweets rather than retweets for now
    search = f"#{coin} -filter:retweets"
    # .items() used for specifying how many tweets we want to obtain
    tweetCursor = tweepy.Cursor(twitterAPI.search_tweets, q=search, lang="en", tweet_mode="extended").items(100)
    # list of tweets in text form
    tweets = [tweet.full_text for tweet in tweetCursor]

    # converting to dataframe
    tweets_df = pd.DataFrame(tweets, columns=["Tweets"])
    # cleaning the data (removing tags, hashtags, etc)
    for _, row in tweets_df.iterrows():
        row["Tweets"] = re.sub("http\S+", "", row["Tweets"])
        row["Tweets"] = re.sub("#\S+", "", row["Tweets"])
        row["Tweets"] = re.sub("@\S+", "", row["Tweets"])
        row["Tweets"] = re.sub("\\n", "", row["Tweets"])    # double backslash so it makes \n a null char

    # performing sentiment analysis per tweet and assigning their polarity score
    tweets_df["Polarity"] = tweets_df["Tweets"].map(lambda tweet: textblob.TextBlob(tweet).sentiment.polarity)
    # if polarity is greater than 0, positive, else negative
    tweets_df["Result"] = tweets_df["Polarity"].map(lambda pol: "+" if pol > 0 else "-")

    # count all tweets where result is positive and negative
    positive = tweets_df[tweets_df.Result == "+"].count()["Tweets"]
    negative = tweets_df[tweets_df.Result == "-"].count()["Tweets"]

    # returning the positive & negative market sentiment via list so we can read it from sentiment()
```

The other remaining features of the application provide additional market sentiment information, along with the portfolio section that allows users to save/edit their selected coins in their portfolio which displays technical and analytical information about the selected coins.



Here is a technical diagram of how the tweepy API obtains the tweets from Twitter and the process of how it takes the data and displays it on the front end.

Search News:

```python
# route to search results page
@app.route("/searchResults", methods=["GET", "POST"])
def results():
    # if user makes a POST request
    if request.method == "POST":
        # access data inside
        query = request.form.get("query")
        selectedFilter = request.form.get("selectedFilter")
        # if the query is not empty, pass it to the news api
        if query:
            # save the current query to a global variable if the user wants to
            global cachedQuery
            cachedQuery = query

            # by default, sorting by relevancy (if param not specified, newsapi
            response = newsAPI.get_everything(
                                        q=query,
                                        language="en",
                                        sort_by="relevancy",
                                        page_size=100
                                    )

            # formatting the date to timeAgo format
            refactoredResponse = publishedDateFormatter(response)

            # bring them to results.html displaying query results
            return render_template("results.html",
                                   response=response["articles"],
                                   query=query,
                                   filterOptions=list(filterOptions.keys())
```

This code snippet from the backend shows how the user search query is obtained from the frontend and passed to the news API. By default, we want relevant results to be displayed first, and we limited the search results to 100. If we had more time to work on the project, we could have approached this by implementing some sort of pagination for the user and display more search results based on the user query. We chose to stick with limiting it to 100 search results as we believe that this is more than enough for the user to scroll through.

```
265         # else if the query was entered but a filter was applied
266         elif query is None:
267 
268             # restore the original query and apply the filter to it
269             query = cachedQuery
270 
271             # reordering the temp list of filters so the selected filter will be di
272             # know which filter they selected)
273             selectedFilterValue = filterOptions[selectedFilter]
274             filterOptionsReordered = list(filterOptions.keys())
275             filterOptionsReordered.remove(selectedFilter)
276             filterOptionsReordered.insert(0, selectedFilter)
277 
278             # sorting by the filter that the user selected
279             response = newsAPI.get_everything(
280                                         q=cachedQuery,
281                                         language="en",
282                                         sort_by=selectedFilterValue,
283                                         page_size=100
284                                     )
285 
286             # formatting the date to timeAgo format
287             refactoredResponse = publishedDateFormatter(response)
288 
289             # bring them to results.html displaying query results
290             return render_template("results.html",
291                                    response=response["articles"],
292                                    query=query,
293                                    selectedFilter=selectedFilter,
294                                    filterOptions=filterOptionsReordered
295                                )
296 
297         # else the query is empty, (newsapi raises exception if empty query is pass
298         return render_template("results.html", query=query)
```

Upon searching the news, the user can also select a search filter from 3 options: most relevant (default), latest, and most popular. Depending on the option, the user's original search query is saved in a temporary variable and the selected search filter is passed to the news API parameters and the page is refreshed once the search results are available.

```python
# helper function for results() to reformat the date/time entry of when article w
def publishedDateFormatter(response):
    # obtain current date for timeAgo functionality (for comparing current date w
    now = dt.datetime.now()
    date_format = "%Y-%m-%dT%H:%M:%SZ"

    # changing the "publishedAt" date to a timeAgo format
    # (e.g instead of showing a date, it'll show something like "x hours ago", et
    for result in response["articles"]:
        publishedDate = dt.datetime.strptime(result["publishedAt"], date_format)
        timeAgoFormat = timeago.format(publishedDate, now)
        result["publishedAt"] = timeAgoFormat

    return response
```

The purpose of the date format is to improve the readability of the application. Instead of showing the published news article date in the form DD/MM/YYYY, we thought it would be better to show how long ago something was published given that this format can give the user a better understanding of how long ago something happened or was published.

Fear and Greed:

```python
# fng = Fear And Greed
# obtaining fear and greed for current date, previous day, previous week, and previous month
@app.route("/fng")
def fearAndGreed():

    # first tile (in fearAndGreed.html) showing fng of today via graph by passing current date to API to
    # obtain the image graph
    currDate = dt.datetime.now()
    # Replace "-" with "#" if using Windows (see here: https://www.kite.com/python/answers/how-to-remove-leading-0's-in-a-date-in-pytho
    currDateString = currDate.strftime("%Y-%#m-%#d")
    currDateURL = "https://alternative.me/images/fng/crypto-fear-and-greed-index-{}.png".format(currDateString)

    # now we're storing the historic fng values (including todays) for the second tile
    historicValues = {}
    # limit=32 because we want to see fng over the last month
    fngIndexes = requests.get("https://api.alternative.me/fng/?limit=32")
    fngIndexesJson = fngIndexes.json()["data"]

    # obtaining historic fng values and their corresponding classification
    # (format of dictionary) historicValues{"timeOfMonth": [fng value, fng meaning]}
    historicValues["Today"] = [int(fngIndexesJson[0]["value"]), fngIndexesJson[0]["value_classification"]]
    historicValues["Yesterday"] = [int(fngIndexesJson[1]["value"]), fngIndexesJson[1]["value_classification"]]
    historicValues["Last Week"] = [int(fngIndexesJson[7]["value"]), fngIndexesJson[7]["value_classification"]]
    historicValues["Last Month"] = [int(fngIndexesJson[30]["value"]), fngIndexesJson[30]["value_classification"]]

    # converting timestamp to int for frontend to convert the time left in seconds into hours, minutes, and secs
    secondsUntilUpdate = int(fngIndexesJson[0]["time_until_update"])

    # passing generated fng images to html page to display it
    return render_template("fearAndGreed.html", todaysGraph=currDateURL, historicValues=historicValues, secondsLeft=secondsUntilUpdate)
```

The fear and greed route calls the fear and greed API based on the current date. The API generates an image of the fear and greed index and displays what it currently is. It also shows the fear and greed index of the previous day, previous week, and previous month. This will also allow the user to better understand the market sentiment. There is also a timer that shows when the fear and greed index will refresh.

Portfolio:

```
346 # idea: have a separate route that basically determines whether to go to the selector or the summary it
347 # ofc on first access, you select coins, but when you press "back", THAT'S when you can edit the portfo
348 @app.route("/portfolio")
349 def portfolio():
350
351     # if user wants to visit portfolio given that they already selected the coins initially, display it
352     if session.get("selectedCoinsList"):
353         oldSelectedCoinsList = session.get("selectedCoinsList")
354         tradingViewSymbols = [["BINANCE:{}EUR".format(crypto[coin])] for coin in oldSelectedCoinsList]
355         return render_template("portfolioSummary.html", tradingViewSymbols=tradingViewSymbols)
356
357     # else, let them edit portfolio/create a new one if visiting site on a new session
358     # (in the URL, it'll still say "/portfolio" instead of "/portfolioSelector" since we're
359     # calling a function from "/portfolio" (unless we click on the "edit portfolio" button))
360     return portfolioSelector()
```

The portfolio route displays the user-selected coins in a TradingView widget which the user can use to easily draw trend lines and display other technical information about the coins they selected. With the use of flask sessions, if the user accesses the portfolio route for the first time, they will be asked to select which coins to add to their portfolio after which these coins are stored in the session. This saved us from having to use a database of some sort that would require users to sign up/log in just to be able to view their portfolio. Otherwise, the user already saved their coins to their portfolio and this has been saved in the current Flask session, so the application will display their portfolio.

```
362 # user selects coins to add to portfolio summary
363 @app.route("/portfolioSelector")
364 def portfolioSelector():
365
366     cryptoCoins = list(crypto.keys())
367     return render_template("portfolioSelector.html", cryptoCoins=cryptoCoins)
368
```

If the user wants to edit their portfolio or create their portfolio from a new session, calling the portfolioSelector() route will essentially display all the coins the user can select from to add to their portfolio.

```
373 # display statistical summary based on the user-selected coins
374 @app.route("/portfolioSummary", methods=["GET", "POST"])
375 def portfolioSummary():
376     # if user makes a POST request
377     if request.method == "POST":
378         # obtaining list of user-selected coins
379         selectedCoinsList = request.form.getlist("selectedCoins")
380         # save the user-selected coins in current session so user can avoid having to re-add coins
381         # current portfolio per session
382         session["selectedCoinsList"] = selectedCoinsList
383         # then converting list to the following format: "BINANCE:BTCEUR" (for tradingView chart JS)
384         tradingViewSymbols = [["BINANCE:{}EUR".format(crypto[coin])] for coin in selectedCoinsList]
385
386     # passing selected coins as well as all the top 20 coins (for mapping selected coin with ticker
387     return render_template("portfolioSummary.html", tradingViewSymbols=tradingViewSymbols)
```

The portfolio summary route simply just obtains the user-selected coins stored in the current Flask session and displays them.

6. Problems and Resolution

6.1 Prediction Accuracy

Given that we never developed a machine learning application before, we at first found it difficult to find the best approach to starting this project. We received advice from our supervisor as to what way we could go about it and we were advised to use the ARIMA model. We realised that no matter what model we would use, it would be very difficult to predict the future prices of cryptocurrencies especially given that they are very volatile. The ARIMA model does show a prediction but they are not as accurate as we liked, hence, why we focused a lot on implementing some market sentiment features too so that users would not fully rely on predictions. In the end, the application was intended to be more of a guide for traders and investors to use for buy and sell signals based on predictions and market sentiment.

6.2 Displaying of Data

When it came to the displaying of our data, We initially just had the graphs being displayed as static photos which were saved in the directory. We wanted to try and display the data in a live and interactive way. Firstly we tackled the market sentiment graph. We initialy had the data displayed in a bar chart which didnt look great on the web page. We decided to go with a interactive pie chart made using javascript. With the use of great resources out there we used a pie chart template on chart.js which had set values making up the chart. We altered the colours and size of the graph to fit our needs.
We wanted to be able to parse the real time tweets totals across from the tweepy api to the front end to display them. This was done with the use of {{}} double curly braces.

```
    positive, negative, = getTweets(coinName)[0], getTweets(coinName)[1]
    neutral = abs(100 - (positive + negative))
    # passing the plot to coinSentimentGraph.html
    return render_template("coinSentimentGraph.html", coinName=coinName, coinTicker=coinTicker,
                           positive=positive, neutral=neutral, negative=negative)
```

This took the value that had been obtained from tweepy and parsed it across to be used in the javascript chart. As you can see from the snippet below {{posivite}} contained the values for the count of positive tweets and same for negative and neutral.

```
                    <table id="chartData">


<tr>
  <th>Positive Or Negative</th><th>{{coinName}}</th>
 </tr>

<tr style="color: green">
  <td>Positive</td><td>{{positive}}%</td>
</tr>

<tr style="color: red">
  <td>Negative</td><td>{{negative}}%</td>
</tr>
                              <tr style="color: black">
  <td>Neutral</td><td>{{neutral}}%</td>
</tr>



</table>
```

6.4 Rate Limit on Tweets

When it comes to obtaining the tweets, we sometimes ran into the issue of trying to deal with rate limits. This is because Twitter only allows a certain number of tweets to be processed at a time, so we decided to stick with processing 100 tweets and at the same time, provide the user with the link to view all tweets associated with the coin they selected when displaying the market sentiment.

7. Results

Most of the features we planned initially have been implemented and overall, they give the results that we expected. Of course, however, given that we have never developed a machine learning application before, we found it difficult at first to approach the project and this was a major limitation throughout the development of our application. Our application has a prediction feature just as we initially wanted, and it also has the market sentiment feature which was the first sentiment feature we added to the application. From there onwards, we added more and more features to the application to help make the app feel like everything is in one place rather than having to have multiple applications and browser tabs open. The search feature was implemented how we wanted it initially, allowing users to

search for anything related to cryptocurrency news. The converter/calculator feature ended up better than we expected given that it is a widget that allows users to enter values and select various currencies while also showing the real-time price based on the inputs. The portfolio and fear and greed features seemed tricky to implement at first as we were not sure what should be shown in those parts of the application but we managed to figure it out in the end without overcomplicating the application. One of the main aims of our project was to keep it simple and easy to use without overwhelming the user so as to accommodate as many users as possible.

8. Future Work

As for future work, there are some options. One thing we could do differently is to implement the cryptocurrency price prediction feature with an LSTM model or some other machine learning algorithm. It was our first time developing an application to this scale with machine learning, and the ARIMA model did work well but also could have experimented with another predictive model to better determine how to predict the future prices of cryptocurrencies.

We would also aim to plan more in advance and get more hands-on experience with machine learning if we were to expand this application further beyond where it is currently at. We also could include stock market technicals along with the cryptocurrency statistics so as to further expand the target audience for this application. We initially planned on doing so, but due to time constraints and design choices, we stuck with cryptocurrencies. If we had more time, we could have further improved the prediction feature frontend and backend-wise.