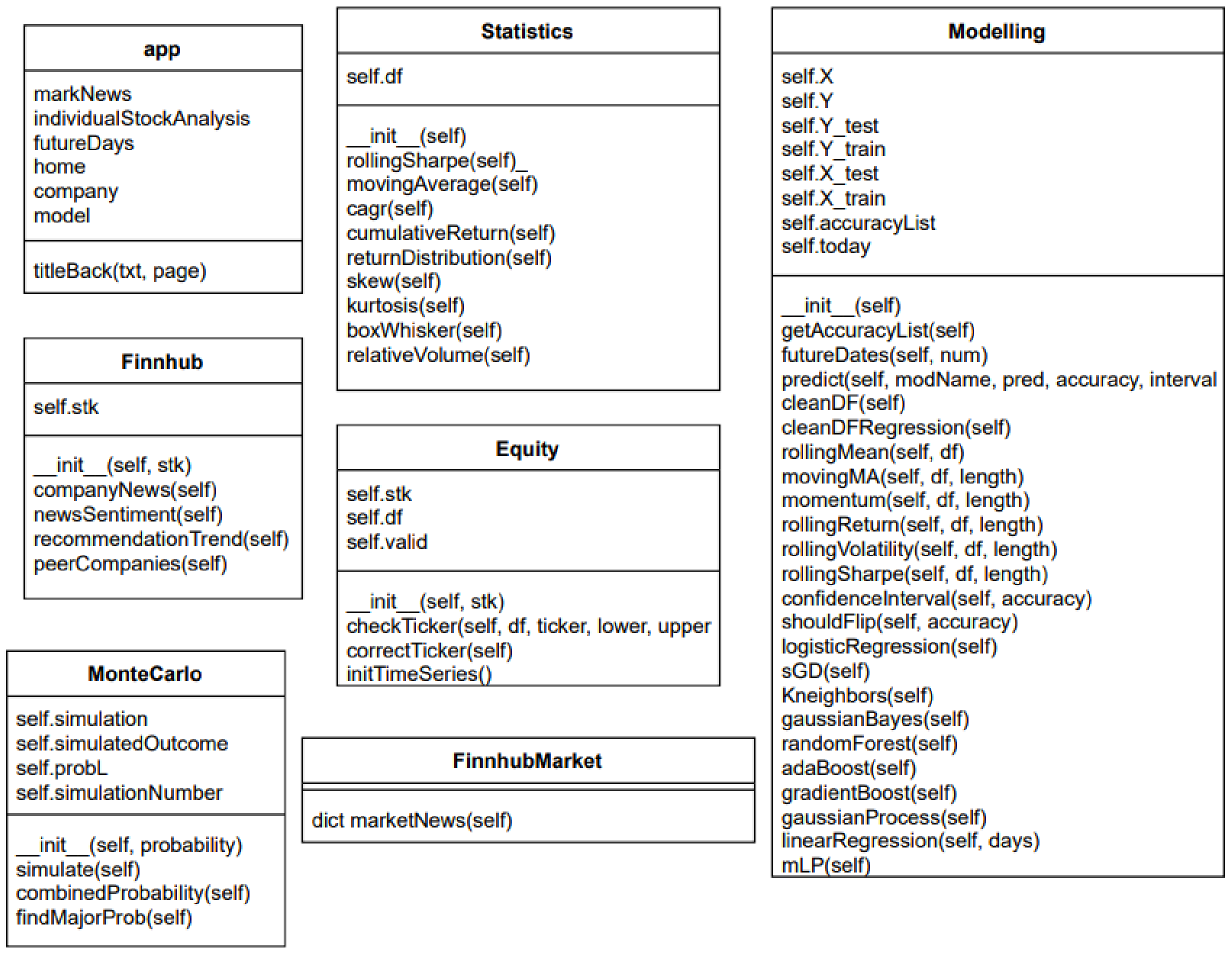
Criterion C: Development

Word Count: 945

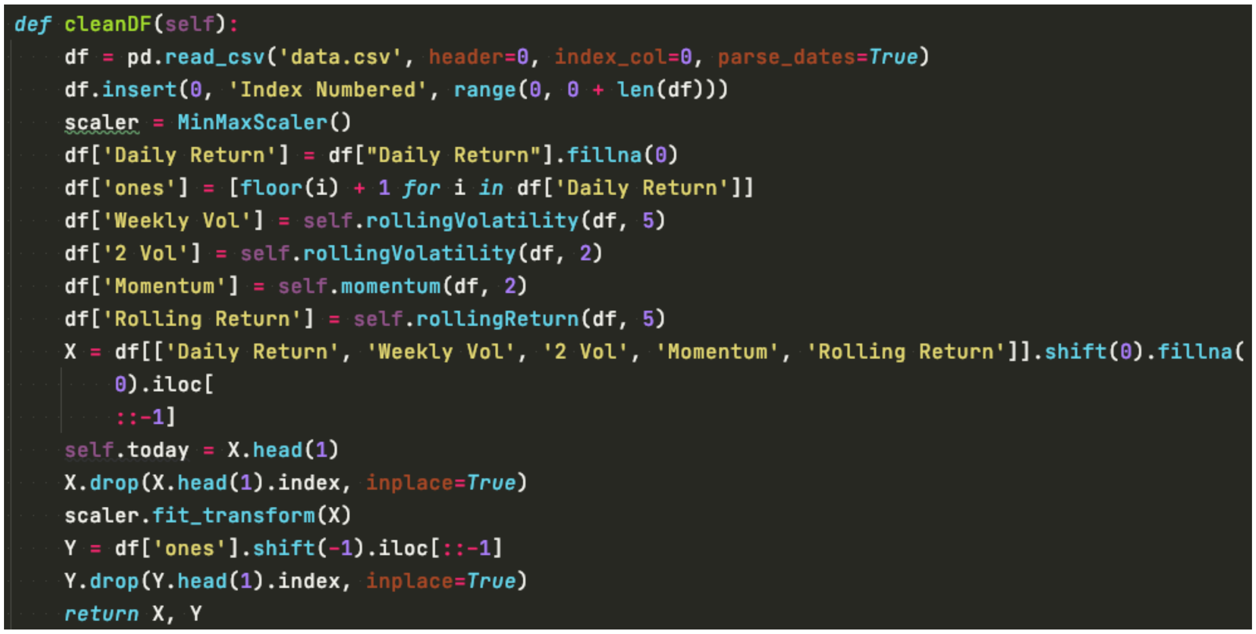
UML Representation of Class and Methods



Modeling Class - File Reading, Machine Learning, Data Processing/Manipulation

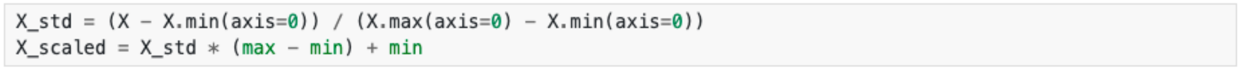
Cleaning Data

The modeling class involves four crucial parts of machine learning: preparing data, feature engineering, training the model, and testing the model.

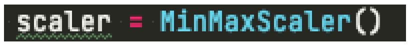


```
def cleanDF(self):
    df = pd.read_csv('data.csv', header=0, index_col=0, parse_dates=True)
    df.insert(0, 'Index Numbered', range(0, 0 + len(df)))
    scaler = MinMaxScaler()
    df['Daily Return'] = df["Daily Return"].fillna(0)
    df['ones'] = [floor(i) + 1 for i in df['Daily Return']]
    df['Weekly Vol'] = self.rollingVolatility(df, 5)
    df['2 Vol'] = self.rollingVolatility(df, 2)
    df['Momentum'] = self.momentum(df, 2)
    df['Rolling Return'] = self.rollingReturn(df, 5)
    X = df[['Daily Return', 'Weekly Vol', '2 Vol', 'Momentum', 'Rolling Return']].shift(0).fillna(
        0).iloc[
        ::-1]
    self.today = X.head(1)
    X.drop(X.head(1).index, inplace=True)
    scaler.fit_transform(X)
    Y = df['ones'].shift(-1).iloc[::-1]
    Y.drop(Y.head(1).index, inplace=True)
    return X, Y
```

This method reads in a DataFrame and cleans the data. The first step is to normalize all the data using MinMaxScaler(). This preprocessing method normalizes the based on the following formula:



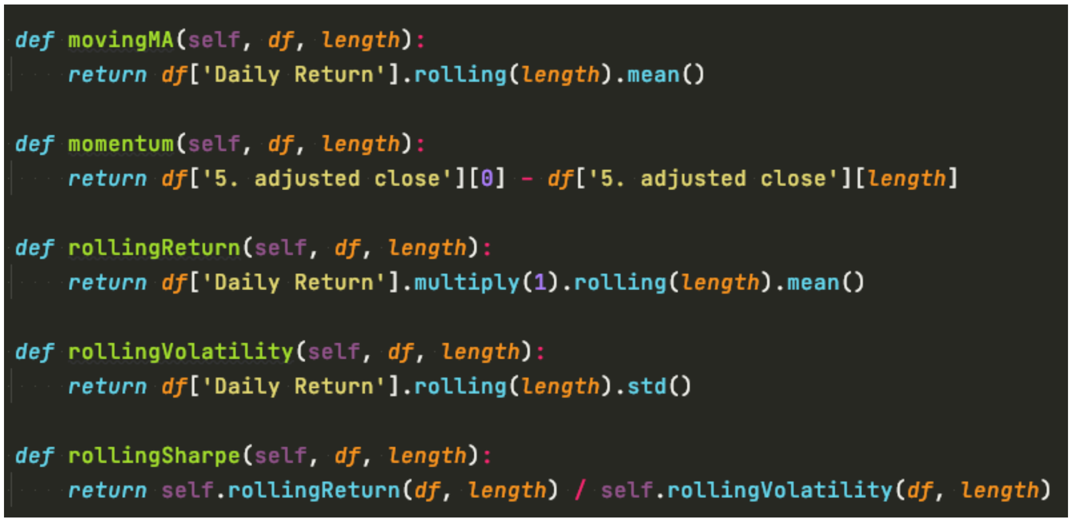
```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```



```
scaler = MinMaxScaler()
```

Normalizing data makes it easier for the machine learning models to accurately and precisely recognize patterns and train the model with a higher prediction accuracy.

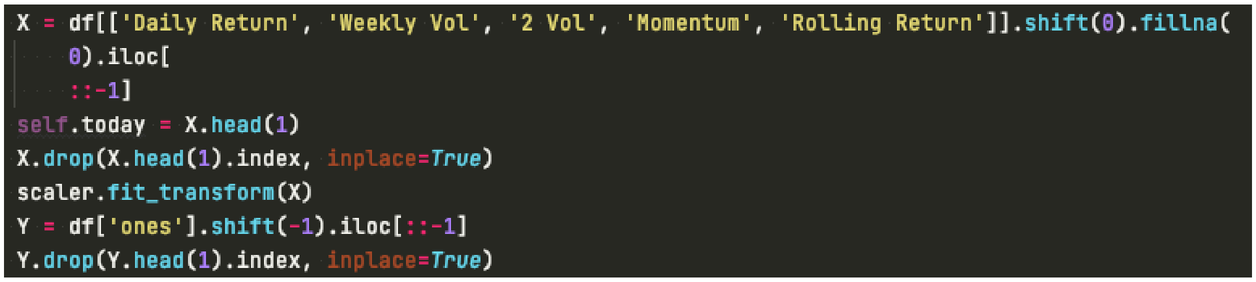
Once normalized, technical analysis features are created in the DataFrame.



```python
def movingMA(self, df, length):
    return df['Daily Return'].rolling(length).mean()

def momentum(self, df, length):
    return df['5. adjusted close'][0] - df['5. adjusted close'][length]

def rollingReturn(self, df, length):
    return df['Daily Return'].multiply(1).rolling(length).mean()

def rollingVolatility(self, df, length):
    return df['Daily Return'].rolling(length).std()

def rollingSharpe(self, df, length):
    return self.rollingReturn(df, length) / self.rollingVolatility(df, length)
```

These methods correctly manipulate the DataFrame rows and using pandas generates rolling windows of data. Stationarity is the property that a set of data has a constant mean or is bounded to a value. Stationarity is crucial as this ensures that the statistical properties of the data do not change over time, and hence the model can accurately recognize patterns and make decisions. Therefore, each feature created must be stationary and bounded to 0.

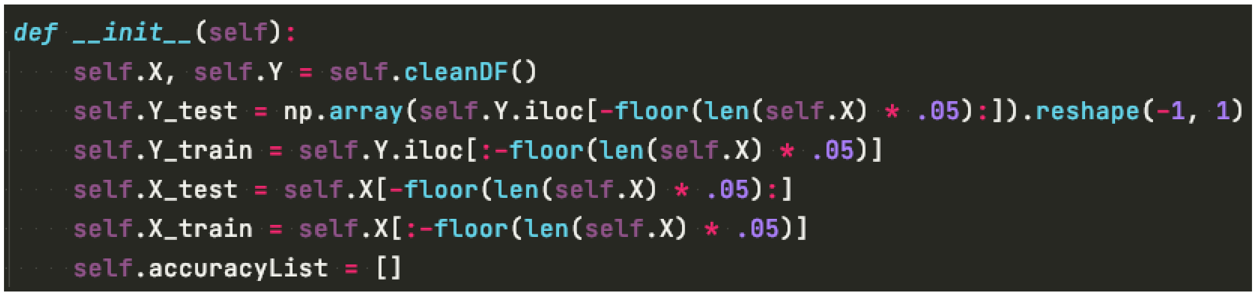
The final step of processing data is creating the X features and Y feature:



```python
X = df[['Daily Return', 'Weekly Vol', '2 Vol', 'Momentum', 'Rolling Return']].shift(0).fillna(
    0).iloc[
    ::-1]
self.today = X.head(1)
X.drop(X.head(1).index, inplace=True)
scaler.fit_transform(X)
Y = df['ones'].shift(-1).iloc[::-1]
Y.drop(Y.head(1).index, inplace=True)
```

The X feature represents all the features the model can learn from. The Y feature is a series of 1s and 0s representing if the market the next day went up or down. It is important to convert the daily returns into binary as this program uses classification algorithms. The data is cleaned and ready.

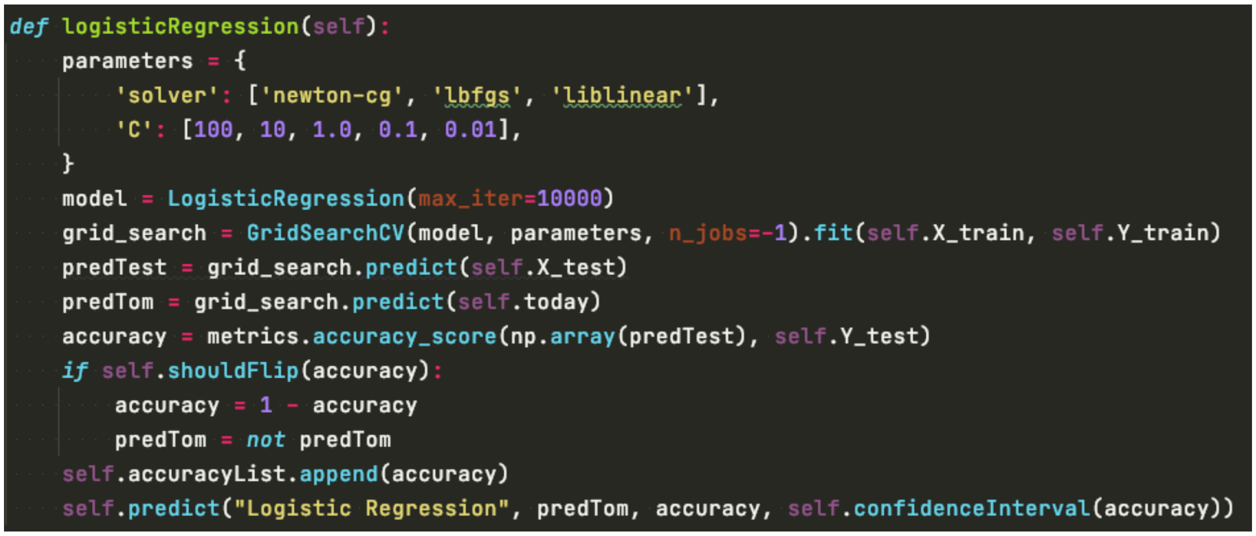
The next step is to separate the data for training vs. testing:



```python
def __init__(self):
    self.X, self.Y = self.cleanDF()
    self.Y_test = np.array(self.Y.iloc[-floor(len(self.X) * .05):]).reshape(-1, 1)
    self.Y_train = self.Y.iloc[:-floor(len(self.X) * .05)]
    self.X_test = self.X[-floor(len(self.X) * .05):]
    self.X_train = self.X[:-floor(len(self.X) * .05)]
    self.accuracyList = []
```

Leaving some data to test is important in order to validate the accuracy of the model.

Training the model

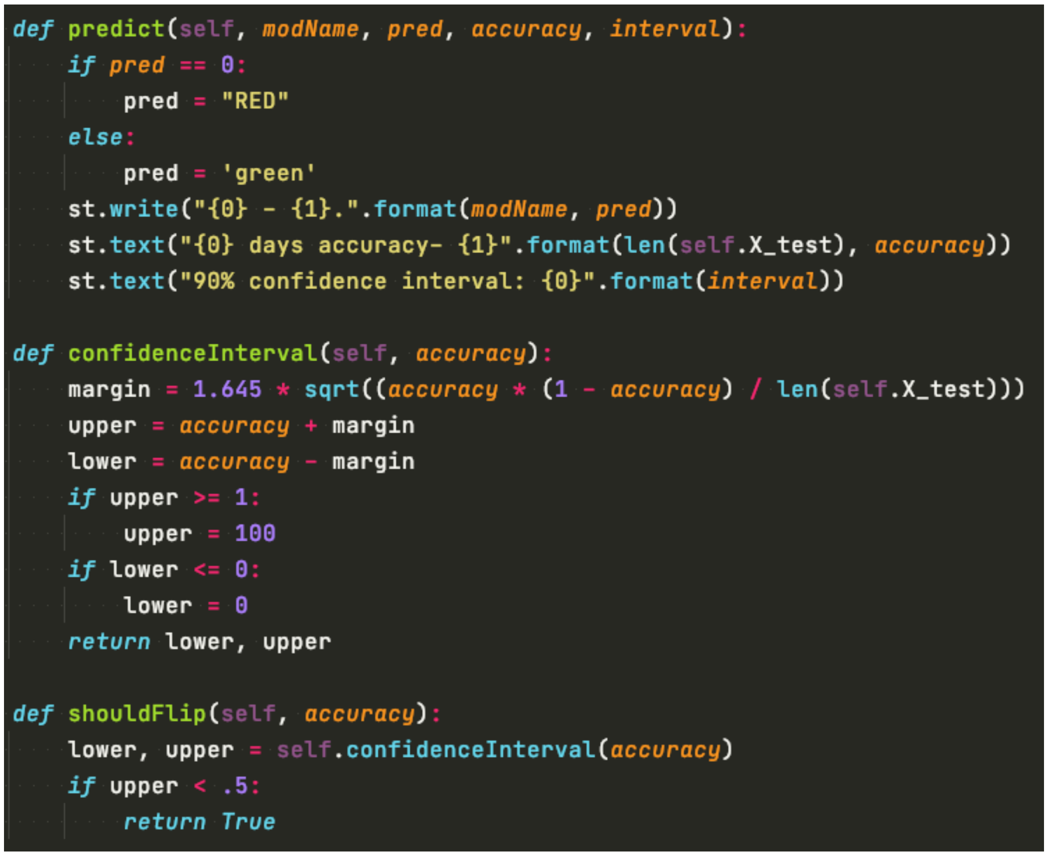


```python
def logisticRegression(self):
    parameters = {
        'solver': ['newton-cg', 'lbfgs', 'liblinear'],
        'C': [100, 10, 1.0, 0.1, 0.01],
    }
    model = LogisticRegression(max_iter=10000)
    grid_search = GridSearchCV(model, parameters, n_jobs=-1).fit(self.X_train, self.Y_train)
    predTest = grid_search.predict(self.X_test)
    predTom = grid_search.predict(self.today)
    accuracy = metrics.accuracy_score(np.array(predTest), self.Y_test)
    if self.shouldFlip(accuracy):
        accuracy = 1 - accuracy
        predTom = not predTom
    self.accuracyList.append(accuracy)
    self.predict("Logistic Regression", predTom, accuracy, self.confidenceInterval(accuracy))
```

While there are 9 unique models, the template for each is the same. First you determine the variety of parameters the model can possibly take. Hyperparameter tuning necessitates trying every possible combination of tuning parameters in order to find the best model to fit. Each parameter wanted to be tested is within the parameters{} dictionary, another abstract data type in python.

The 9 classification models I have used are:
Logistic Regression
Stochastic Gradient Descent Regression
RandomForestClassifier
AdaBoostClassifier
GradientBoostingClassifier
GaussianProcess Classifier
Multi-Layer Perceptron Classifier.

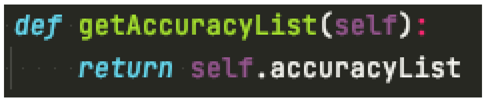
Displaying Results



```python
def predict(self, modName, pred, accuracy, interval):
    if pred == 0:
        pred = "RED"
    else:
        pred = 'green'
    st.write("{0} - {1}.".format(modName, pred))
    st.text("{0} days accuracy- {1}".format(len(self.X_test), accuracy))
    st.text("90% confidence interval: {0}".format(interval))

def confidenceInterval(self, accuracy):
    margin = 1.645 * sqrt((accuracy * (1 - accuracy) / len(self.X_test)))
    upper = accuracy + margin
    lower = accuracy - margin
    if upper >= 1:
        upper = 100
    if lower <= 0:
        lower = 0
    return lower, upper

def shouldFlip(self, accuracy):
    lower, upper = self.confidenceInterval(accuracy)
    if upper < .5:
        return True
```

These three methods calculate the confidence interval, deduce whether or not the model's accuracy can be flipped so that the model's accuracy is greater than 50%, and outputs the data using streamlit.

The accuracy of each model is stored in self.accuracyList and can be retrieved with the getter method:

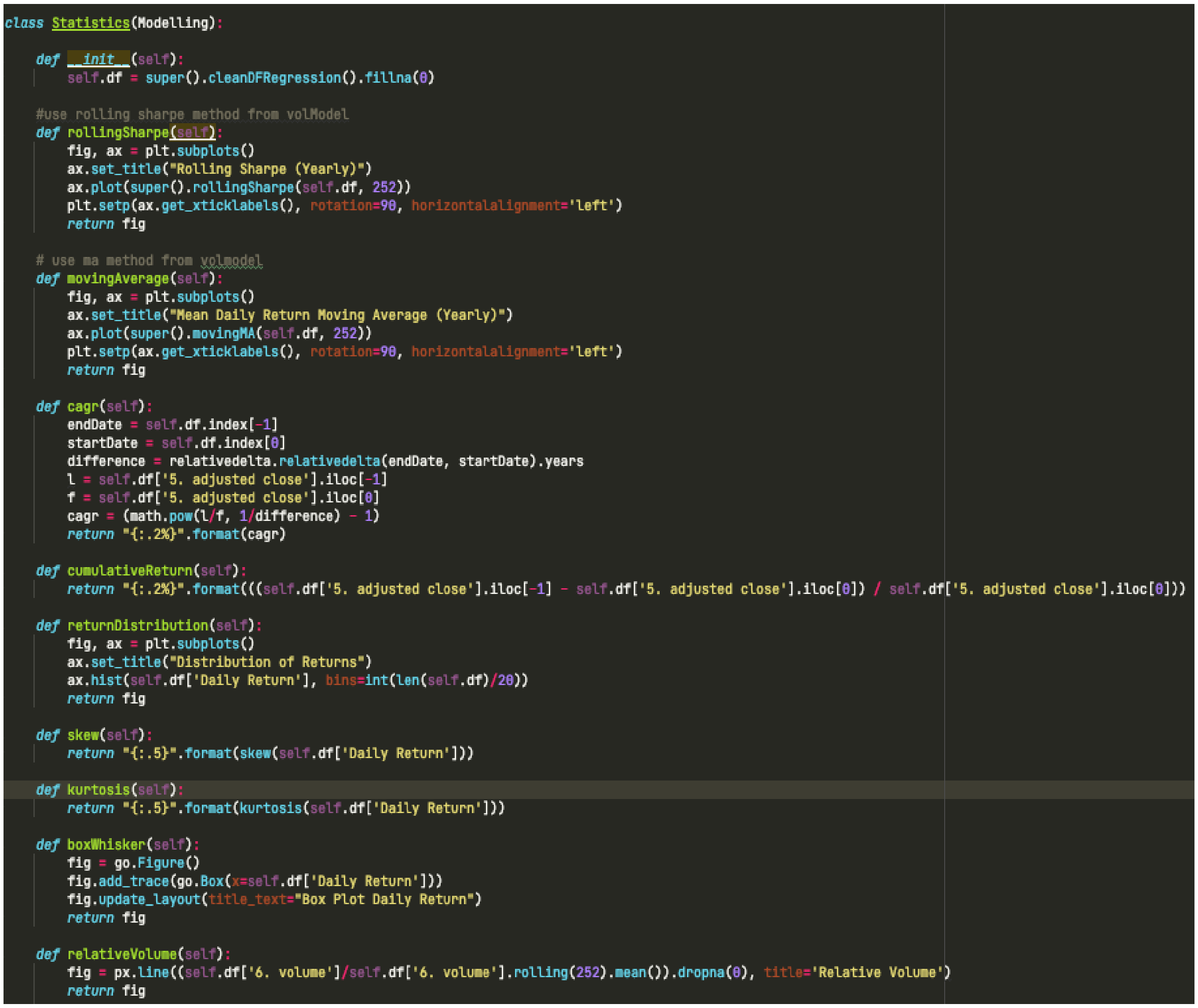


```python
def getAccuracyList(self):
    return self.accuracyList
```

This is used later in the Monte Carlo simulations.

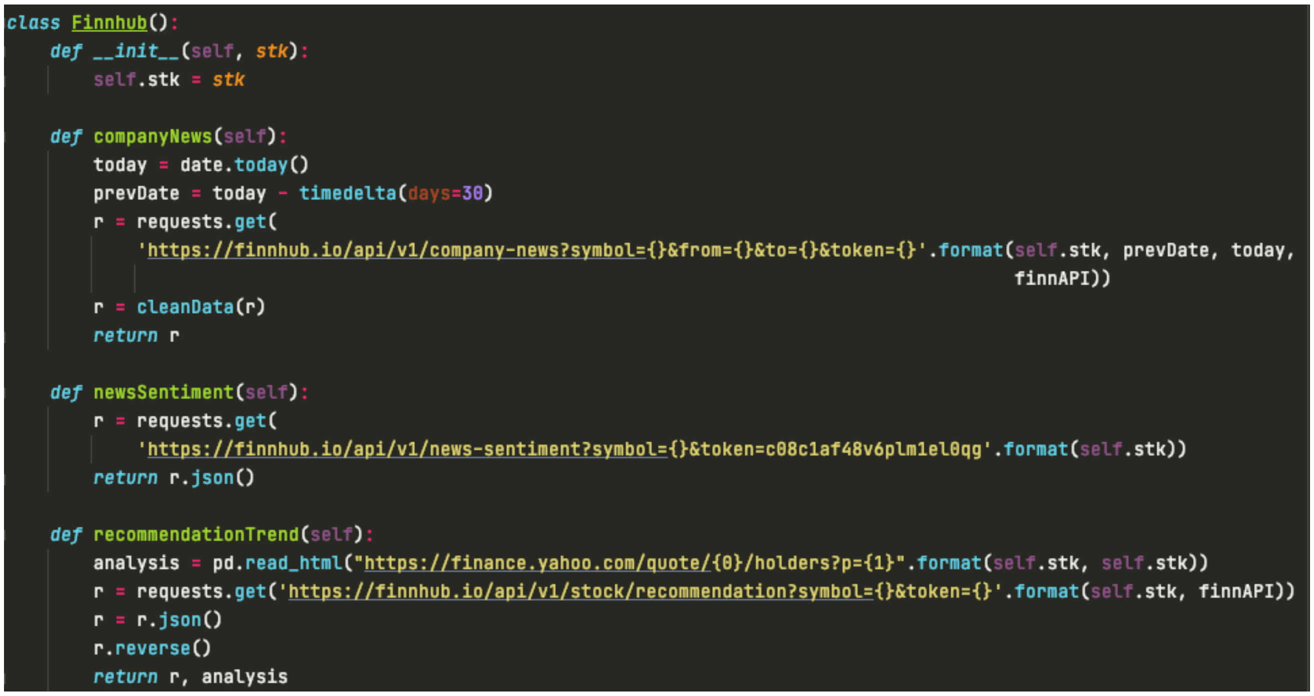
Class Statistics - Graphing

Class statistics inherits Class Modelling's methods and overrides the parent class functions using super(). The purpose of this class is to visualize statistical data using matplotlib and plotly. This process involves creating a subplot, inputting data from the DataFrame, then returning the plot object. Visualization is a crucial requirement for the client as visualizing data allows for analyzing trends and understanding momentum.



```python
class Statistics(Modelling):

    def __init__(self):
        self.df = super().cleanDFRegression().fillna(0)

    #use rolling sharpe method from volModel
    def rollingSharpe(self):
        fig, ax = plt.subplots()
        ax.set_title("Rolling Sharpe (Yearly)")
        ax.plot(super().rollingSharpe(self.df, 252))
        plt.setp(ax.get_xticklabels(), rotation=90, horizontalalignment='left')
        return fig

    # use ma method from volmodel
    def movingAverage(self):
        fig, ax = plt.subplots()
        ax.set_title("Mean Daily Return Moving Average (Yearly)")
        ax.plot(super().movingMA(self.df, 252))
        plt.setp(ax.get_xticklabels(), rotation=90, horizontalalignment='left')
        return fig

    def cagr(self):
        endDate = self.df.index[-1]
        startDate = self.df.index[0]
        difference = relativedelta.relativedelta(endDate, startDate).years
        l = self.df['5. adjusted close'].iloc[-1]
        f = self.df['5. adjusted close'].iloc[0]
        cagr = (math.pow(l/f, 1/difference) - 1)
        return "{:.2%}".format(cagr)

    def cumulativeReturn(self):
        return "{:.2%}".format(((self.df['5. adjusted close'].iloc[-1] - self.df['5. adjusted close'].iloc[0]) / self.df['5. adjusted close'].iloc[0]))

    def returnDistribution(self):
        fig, ax = plt.subplots()
        ax.set_title("Distribution of Returns")
        ax.hist(self.df['Daily Return'], bins=int(len(self.df)/20))
        return fig

    def skew(self):
        return "{:.5}".format(skew(self.df['Daily Return']))

    def kurtosis(self):
        return "{:.5}".format(kurtosis(self.df['Daily Return']))

    def boxWhisker(self):
        fig = go.Figure()
        fig.add_trace(go.Box(x=self.df['Daily Return']))
        fig.update_layout(title_text="Box Plot Daily Return")
        return fig

    def relativeVolume(self):
        fig = px.line((self.df['6. volume']/self.df['6. volume'].rolling(252).mean()).dropna(0), title='Relative Volume')
        return fig
```

Class Finnhub and Finhub Market - HTTP Requests

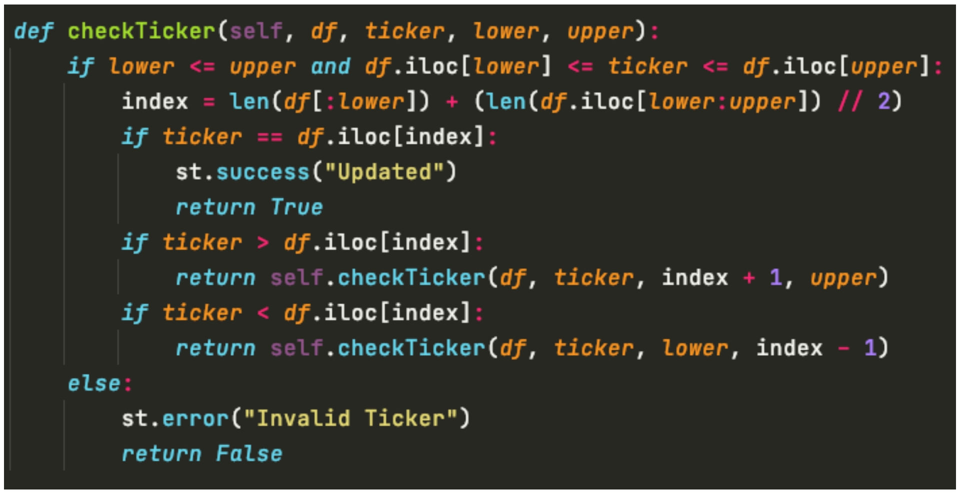


```python
class Finnhub():
    def __init__(self, stk):
        self.stk = stk

    def companyNews(self):
        today = date.today()
        prevDate = today - timedelta(days=30)
        r = requests.get(
            'https://finnhub.io/api/v1/company-news?symbol={}&from={}&to={}&token={}'.format(self.stk, prevDate, today,
                                                                                              finnAPI))
        r = cleanData(r)
        return r

    def newsSentiment(self):
        r = requests.get(
            'https://finnhub.io/api/v1/news-sentiment?symbol={}&token=c08c1af48v6plm1el0qg'.format(self.stk))
        return r.json()

    def recommendationTrend(self):
        analysis = pd.read_html("https://finance.yahoo.com/quote/{0}/holders?p={1}".format(self.stk, self.stk))
        r = requests.get('https://finnhub.io/api/v1/stock/recommendation?symbol={}&token={}'.format(self.stk, finnAPI))
        r = r.json()
        r.reverse()
        return r, analysis
```

Both these classes are responsible for scraping websites and returning json files. This class requires the python library requests, which sends HTTP requests for data. Finnhub() is responsible for retrieving financial news from Finnhub.io (using an API) as well as scraping Yahoo Finance for Equity Holdings information.

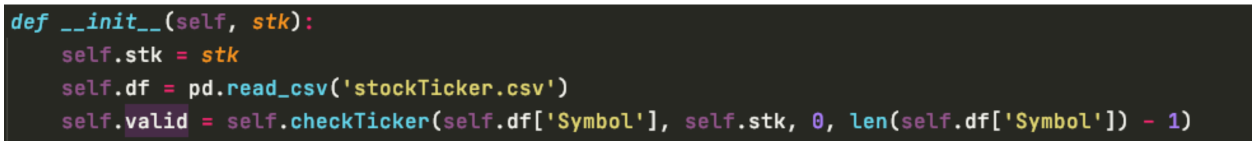
Class Equity - File Reading, Searching Algorithm

Class equity has two crucial roles: Initialize the time series data and check for a valid ticker.



```python
def checkTicker(self, df, ticker, lower, upper):
    if lower <= upper and df.iloc[lower] <= ticker <= df.iloc[upper]:
        index = len(df[:lower]) + (len(df.iloc[lower:upper]) // 2)
        if ticker == df.iloc[index]:
            st.success("Updated")
            return True
        if ticker > df.iloc[index]:
            return self.checkTicker(df, ticker, index + 1, upper)
        if ticker < df.iloc[index]:
            return self.checkTicker(df, ticker, lower, index - 1)
    else:
        st.error("Invalid Ticker")
        return False
```

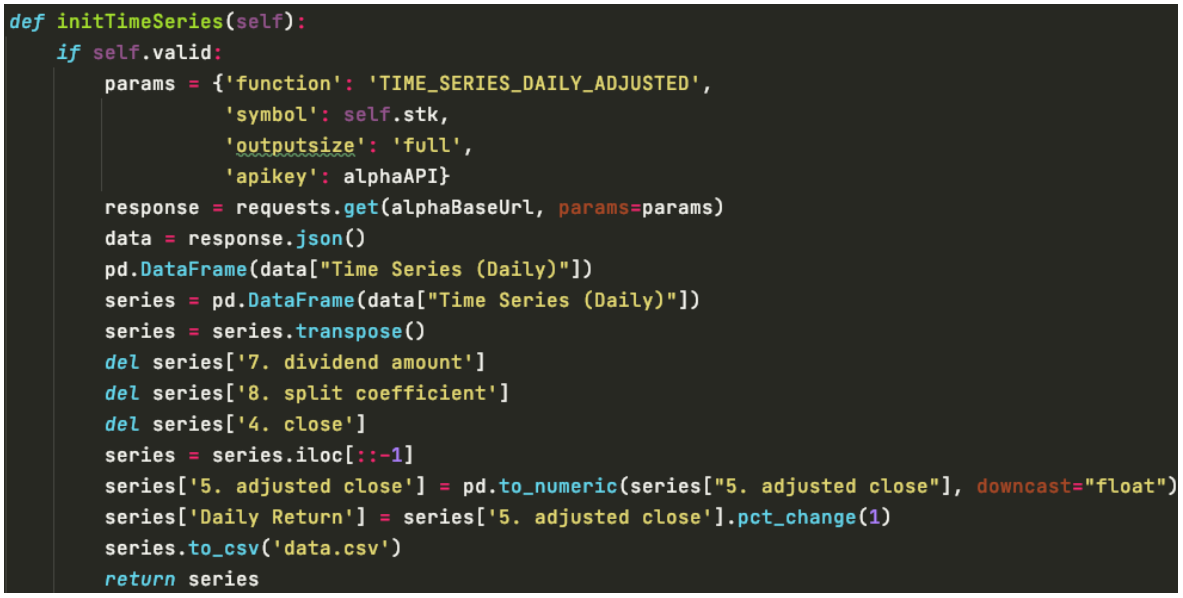
checkTicker() is a recursive binary search algorithm. This algorithm is accessed in the __init__ constructor:



```python
def __init__(self, stk):
    self.stk = stk
    self.df = pd.read_csv('stockTicker.csv')
    self.valid = self.checkTicker(self.df['Symbol'], self.stk, 0, len(self.df['Symbol']) - 1)
```

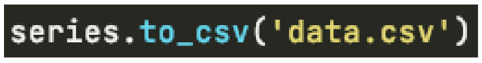
Due to it's recursive nature, this algorithm parses through nearly 8000+ stock tickers with a time complexity O(log n).

The next step is to request historical data using initTimeSeries():



```python
def initTimeSeries(self):
    if self.valid:
        params = {'function': 'TIME_SERIES_DAILY_ADJUSTED',
                  'symbol': self.stk,
                  'outputsize': 'full',
                  'apikey': alphaAPI}
        response = requests.get(alphaBaseUrl, params=params)
        data = response.json()
        pd.DataFrame(data["Time Series (Daily)"])
        series = pd.DataFrame(data["Time Series (Daily)"])
        series = series.transpose()
        del series['7. dividend amount']
        del series['8. split coefficient']
        del series['4. close']
        series = series.iloc[::-1]
        series['5. adjusted close'] = pd.to_numeric(series["5. adjusted close"], downcast="float")
        series['Daily Return'] = series['5. adjusted close'].pct_change(1)
        series.to_csv('data.csv')
        return series
```

This method uses the python library requests and sends and HTTP request to "https://www.alphavantage.co/" with custom parameters in the dictionary params{}. This requires an api key, and will return daily historical data for a stock since inception. Next, the data (returns as a json file but converted into a pandas DataFrame) is manipulated and cleaned so that only the necessary data remains. Finally, the final data is written into 'data.csv' using

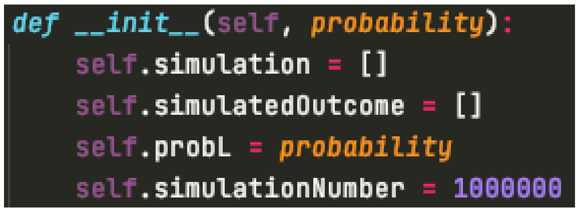


```python
series.to_csv('data.csv')
```

.

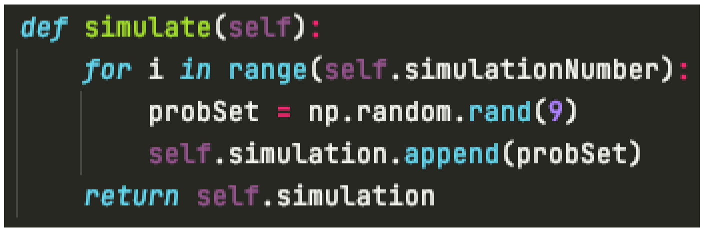
Monte Carlo Simulation

Monte Carlo simulations are effective at simulating probability distributions. The class MonteCarlo requires a list of 9 probabilities, representing the model accuracies.



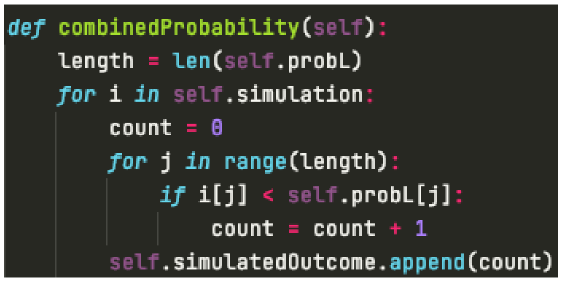
```python
def __init__(self, probability):
    self.simulation = []
    self.simulatedOutcome = []
    self.probL = probability
    self.simulationNumber = 1000000
```

This program will use 1 million simulations. As the law of large numbers states, as the number of simulations increases, the mean value of the simulation gets closer to the real mean.



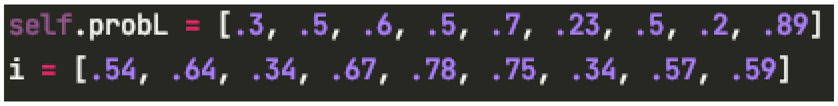
```python
def simulate(self):
    for i in range(self.simulationNumber):
        probSet = np.random.rand(9)
        self.simulation.append(probSet)
    return self.simulation
```

simulate() generates 9 random numbers between 0 and 1 self.simulationNumbers amount of times. Next, the list of 9 numbers are appended into the list self.simulation. By the end of this simulation the two-dimensional list self.simulation contains all the random numbers.



```python
def combinedProbability(self):
    length = len(self.probL)
    for i in self.simulation:
        count = 0
        for j in range(length):
            if i[j] < self.probL[j]:
                count = count + 1
        self.simulatedOutcome.append(count)
```

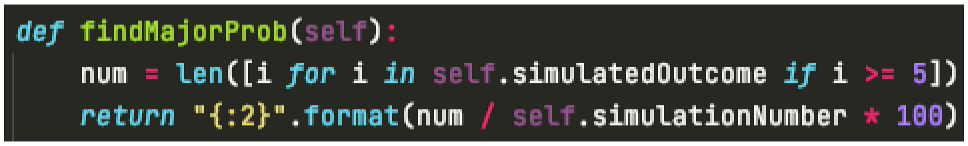
The time complexity of this algorithm is O(self.simulation * 9). This is faster than O(n^2) but way slower than O(N). The nested for loop compares the list of 9 random numbers with 9 model accuracies. For example:



```python
self.probL = [.3, .5, .6, .5, .7, .23, .5, .2, .89]
i = [.54, .64, .34, .67, .78, .75, .34, .57, .59]
```

The algorithm then compares each index for both lists in order to calculate if a model made the “correct prediction”.

After the count has been listed, the probability is calculated and returned.

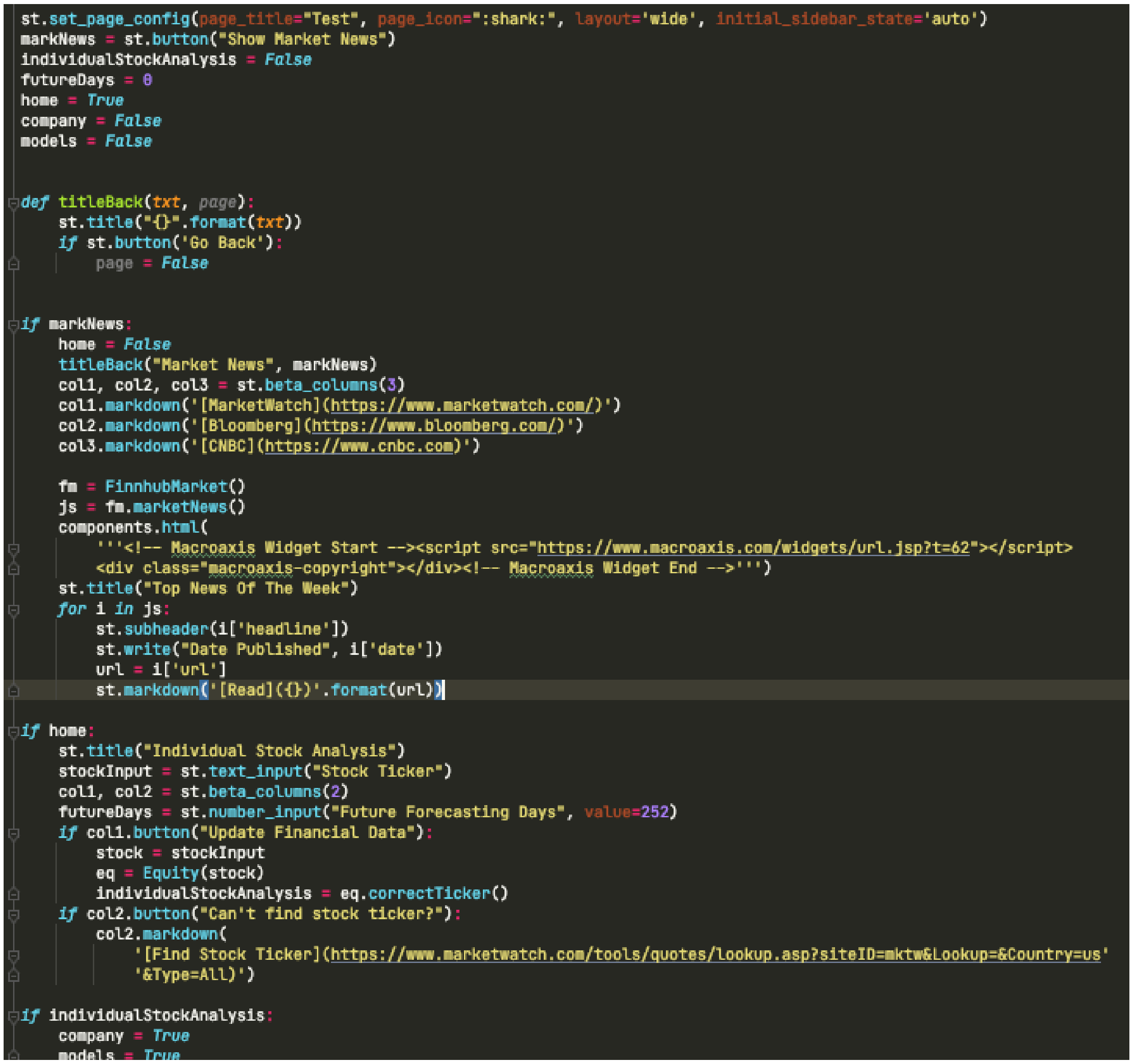


```python
def findMajorProb(self):
    num = len([i for i in self.simulatedOutcome if i >= 5])
    return "{:2}".format(num / self.simulationNumber * 100)
```

App.py - Graphical User Interface

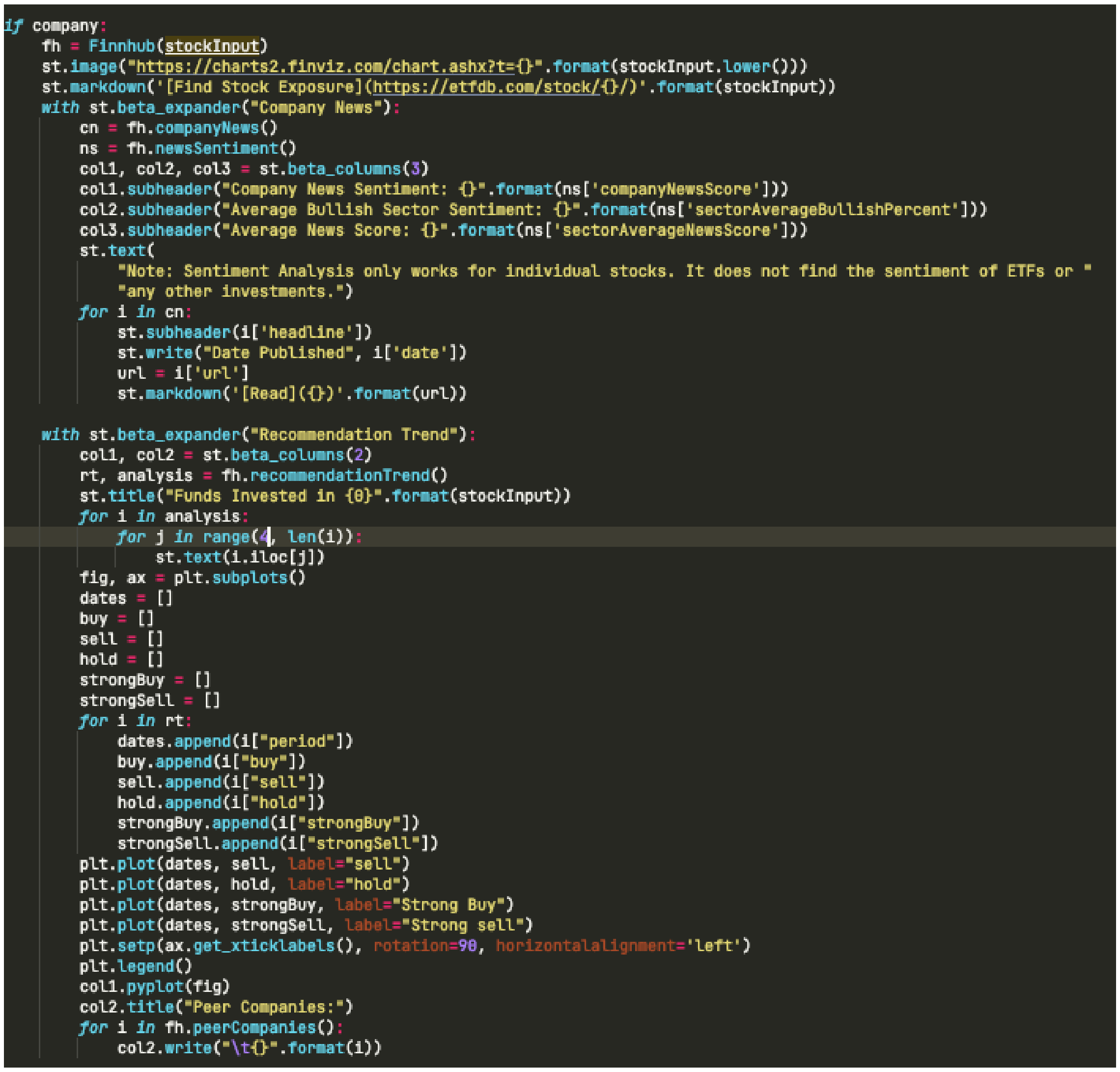
App.py is responsible for visualizing the web app and displaying all the data and information. I decided to use streamlit, a python library which provides a framework for displaying data analysis and machine learning projects. App.py operates using a series of boolean operators. Depending on the state, different parts of the project are displayed.

markNews is responsible for displaying the holistic Market news on a separate page. Additionally, home is the default initialization page seen when first using the program.

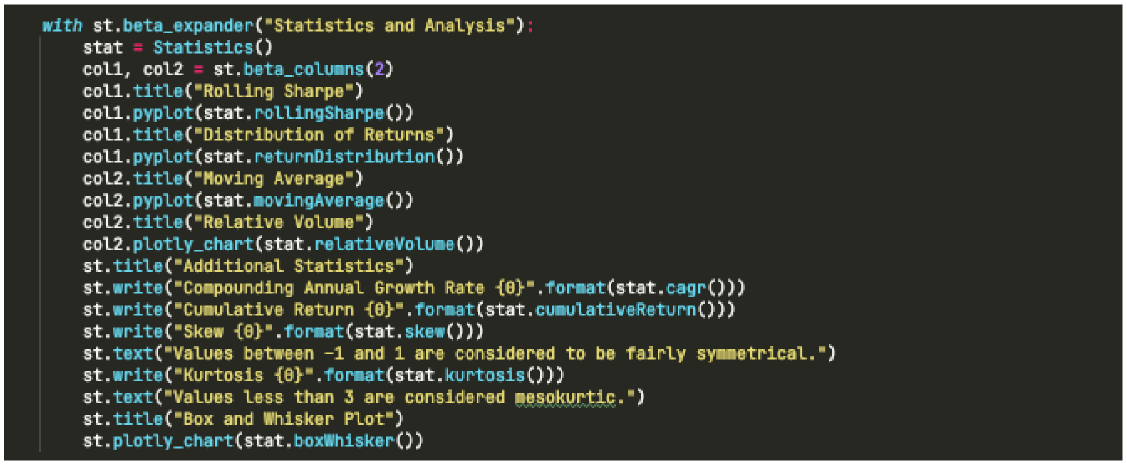


```python
st.set_page_config(page_title="Test", page_icon=":shark:", layout='wide', initial_sidebar_state='auto')
markNews = st.button("Show Market News")
individualStockAnalysis = False
futureDays = 0
home = True
company = False
models = False


def titleBack(txt, page):
    st.title("{}".format(txt))
    if st.button('Go Back'):
        page = False


if markNews:
    home = False
    titleBack("Market News", markNews)
    col1, col2, col3 = st.beta_columns(3)
    col1.markdown('[MarketWatch](https://www.marketwatch.com/)')
    col2.markdown('[Bloomberg](https://www.bloomberg.com/)')
    col3.markdown('[CNBC](https://www.cnbc.com)')

    fm = FinnhubMarket()
    js = fm.marketNews()
    components.html(
        '''<!-- Macroaxis Widget Start --><script src="https://www.macroaxis.com/widgets/url.jsp?t=62"></script>
        <div class="macroaxis-copyright"></div><!-- Macroaxis Widget End -->''')
    st.title("Top News Of The Week")
    for i in js:
        st.subheader(i['headline'])
        st.write("Date Published", i['date'])
        url = i['url']
        st.markdown('[Read]({})'.format(url))

if home:
    st.title("Individual Stock Analysis")
    stockInput = st.text_input("Stock Ticker")
    col1, col2 = st.beta_columns(2)
    futureDays = st.number_input("Future Forecasting Days", value=252)
    if col1.button("Update Financial Data"):
        stock = stockInput
        eq = Equity(stock)
        individualStockAnalysis = eq.correctTicker()
    if col2.button("Can't find stock ticker?"):
        col2.markdown(
            '[Find Stock Ticker](https://www.marketwatch.com/tools/quotes/lookup.asp?siteID=mktw&Lookup=&Country=us'
            '&Type=All)')

if individualStockAnalysis:
    company = True
    models = True
```

Once it is confirmed that the ticker is valid, the company displays the dropdown menus and the respective data.

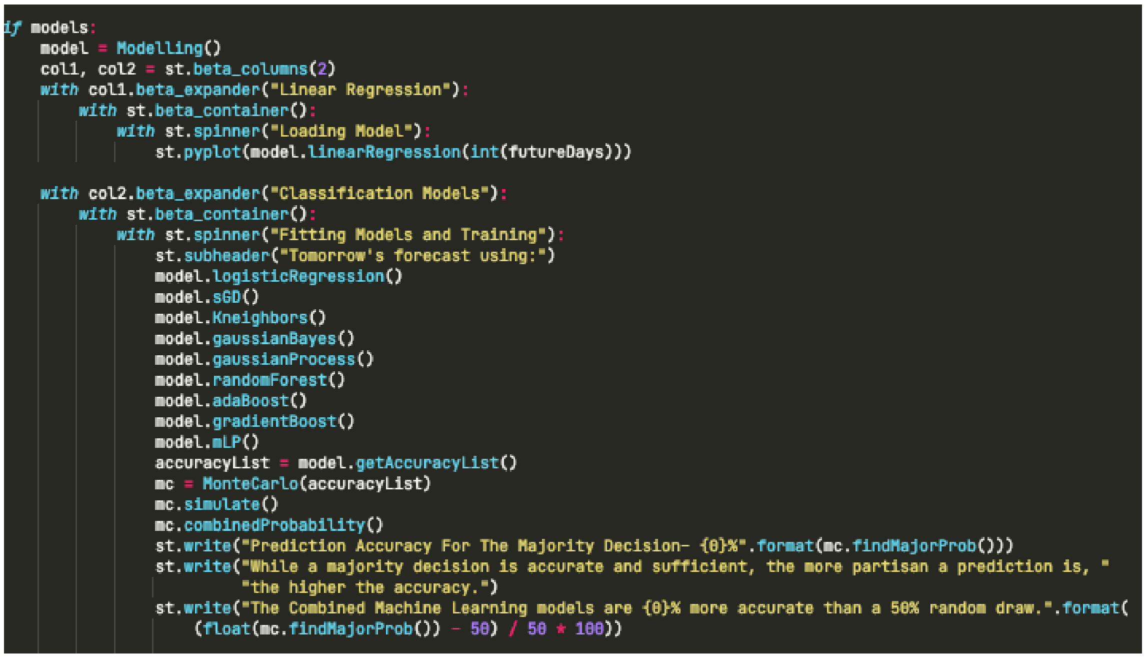


```python
if company:
    fh = Finnhub(stockInput)
    st.image("https://charts2.finviz.com/chart.ashx?t={}".format(stockInput.lower()))
    st.markdown('[Find Stock Exposure](https://etfdb.com/stock/{}/)'.format(stockInput))
    with st.beta_expander("Company News"):
        cn = fh.companyNews()
        ns = fh.newsSentiment()
        col1, col2, col3 = st.beta_columns(3)
        col1.subheader("Company News Sentiment: {}".format(ns['companyNewsScore']))
        col2.subheader("Average Bullish Sector Sentiment: {}".format(ns['sectorAverageBullishPercent']))
        col3.subheader("Average News Score: {}".format(ns['sectorAverageNewsScore']))
        st.text(
            "Note: Sentiment Analysis only works for individual stocks. It does not find the sentiment of ETFs or "
            "any other investments.")
        for i in cn:
            st.subheader(i['headline'])
            st.write("Date Published", i['date'])
            url = i['url']
            st.markdown('[Read]({})'.format(url))

    with st.beta_expander("Recommendation Trend"):
        col1, col2 = st.beta_columns(2)
        rt, analysis = fh.recommendationTrend()
        st.title("Funds Invested in {0}".format(stockInput))
        for i in analysis:
            for j in range(4, len(i)):
                st.text(i.iloc[j])
        fig, ax = plt.subplots()
        dates = []
        buy = []
        sell = []
        hold = []
        strongBuy = []
        strongSell = []
        for i in rt:
            dates.append(i["period"])
            buy.append(i["buy"])
            sell.append(i["sell"])
            hold.append(i["hold"])
            strongBuy.append(i["strongBuy"])
            strongSell.append(i["strongSell"])
        plt.plot(dates, sell, label="sell")
        plt.plot(dates, hold, label="hold")
        plt.plot(dates, strongBuy, label="Strong Buy")
        plt.plot(dates, strongSell, label="Strong sell")
        plt.setp(ax.get_xticklabels(), rotation=90, horizontalalignment='left')
        plt.legend()
        col1.pyplot(fig)
        col2.title("Peer Companies:")
        for i in fh.peerCompanies():
            col2.write("\t{}".format(i))
```



```python
    with st.beta_expander("Statistics and Analysis"):
        stat = Statistics()
        col1, col2 = st.beta_columns(2)
        col1.title("Rolling Sharpe")
        col1.pyplot(stat.rollingSharpe())
        col1.title("Distribution of Returns")
        col1.pyplot(stat.returnDistribution())
        col2.title("Moving Average")
        col2.pyplot(stat.movingAverage())
        col2.title("Relative Volume")
        col2.plotly_chart(stat.relativeVolume())
        st.title("Additional Statistics")
        st.write("Compounding Annual Growth Rate {0}".format(stat.cagr()))
        st.write("Cumulative Return {0}".format(stat.cumulativeReturn()))
        st.write("Skew {0}".format(stat.skew()))
        st.text("Values between -1 and 1 are considered to be fairly symmetrical.")
        st.write("Kurtosis {0}".format(stat.kurtosis()))
        st.text("Values less than 3 are considered mesokurtic.")
        st.title("Box and Whisker Plot")
        st.plotly_chart(stat.boxWhisker())
```

Finally, models is responsible for displaying all the machine learning models, as well as the prediction accuracy for the majority decision (MonteCarlo)



```
if models:
    model = Modelling()
    col1, col2 = st.beta_columns(2)
    with col1.beta_expander("Linear Regression"):
        with st.beta_container():
            with st.spinner("Loading Model"):
                st.pyplot(model.linearRegression(int(futureDays)))

    with col2.beta_expander("Classification Models"):
        with st.beta_container():
            with st.spinner("Fitting Models and Training"):
                st.subheader("Tomorrow's forecast using:")
                model.logisticRegression()
                model.sGD()
                model.Kneighbors()
                model.gaussianBayes()
                model.gaussianProcess()
                model.randomForest()
                model.adaBoost()
                model.gradientBoost()
                model.mLP()
                accuracyList = model.getAccuracyList()
                mc = MonteCarlo(accuracyList)
                mc.simulate()
                mc.combinedProbability()
                st.write("Prediction Accuracy For The Majority Decision- {0}%".format(mc.findMajorProb()))
                st.write("While a majority decision is accurate and sufficient, the more partisan a prediction is, "
                         "the higher the accuracy.")
                st.write("The Combined Machine Learning models are {0}% more accurate than a 50% random draw.".format(
                    (float(mc.findMajorProb()) - 50) / 50 * 100))
```