An Introduction into Machine Learning for Algorithmic Trading

Brandon Christensen

CPTS 315

Course Project Report

Introduction

Background

The stock market is a game with a high learning curve and huge potentials for gains or losses. According to **Investopedia**, the stock market is "the collection of markets and exchanges where regular activities of buying, selling, and issuance of shares of publicly-held companies take place".

When I was first introduced, it was simplified down to, buy a stock and hope it goes up. Eventually, further down the rabbit hole I learned about **trends and technical indicators**. These are pattern-based signals which indicate future price movements. For example, using statistical trends gathered from **trading activity, price, and volume** to indicate a stock moving up or down in the next given time period. At first glance, these indicators sound extremely promising. Technical analysis has also been around since the 1900's, however, the results have been mixed at best. That is why it is important to remember that the signals should be used as reinforcement, not proof of a trend. If they were always right, then everyone would be rich on the stock market.

However, I want to challenge that idea. Maybe there is more to these indicators. What if humans just aren't very good at making logical and emotionless decisions? Humans are also only capable of handling a miniscule subset of the data available on the stock market at a time. But what if there was an automated process that could use these technical indicators to **predict the market?**

Machine Learning **(ML)** fits this criteria perfectly. ML is a method of data analysis that automates analytical model building. By using supervised learning algorithms, we can use past data to classify features, and then hopefully predict future price trends with them.

With ML, we can use technical indicators as features to produce a model, and then we can make use of the large amounts of data from the stock market to produce labels and test the model. A working system would make money automatically and optimally, without any human intervention. I believe that ML is powerful enough to accomplish this.

Questions

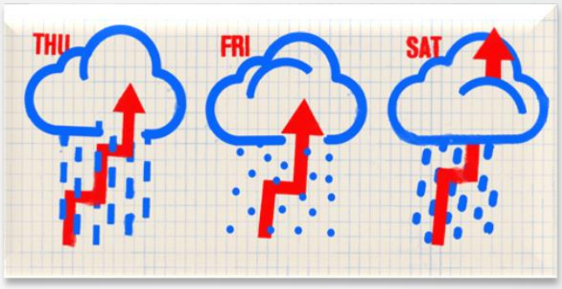
I now had a task, but I still didn't quite know what the project would entail. So, I began with a simple list of questions that I was interested in answering by the end of the project:

? Is ML a **good strategy** for trading stock, and is it even applicable

? Is removing the **human** element in trading beneficial

? What python **APIs** can help with this project

Goals



Along with my questions, I also had personal goals that I wanted to accomplish. My number one priority being to make some **MONEY!**

However, that might be a bit too ambitious for this first iteration. In fact, from the research I have done online, there seems to be a consensus that ML at this point is an ineffective strategy to use when trading on the market. This does not mean it is impossible, but I realize that this won't be an easy one and done task. So, I set myself more realistic goals to achieve by the end of this project:

- ❑ Go live and make profitable trades over 50% of the time (very ambitious long-term goal)
- ❑ Create a working framework that uses ML and can back test on trade history data
  - ❑ Connect to TD Ameritrade and use their API
- ❑ Deepen my understanding of ML and its inner mechanics
  - ❑ Understand what technical indicators are and how they are calculated
- ❑ Understand how to use the APIs I find for this project

Challenges

Before I started coding, I needed to research a couple of different topics. I first wanted to know how ML works so that I could properly apply it to this problem. Then I needed to learn about different technical indicators and if they were implementable with the data I had. Finally, I had to read into the documentation of the APIs that I ended up using.

After that, I then started the coding process. I first broke it down into manageable steps and completed the project like an iterative process. A brief overview of my steps were:

1. Use the TD Ameritrade API to get stock history
   - Convert to and from a CSV file
2. Choose technical indicators to use as features, and translate them into code
   - Find online and convert them into code
3. Choose classifiers to use and fit them to my features
   - Using scikit-learn API
4. Back test my strategy and find its effectiveness on real data
   - Using scikit-learn, pandas, and backtesting.py APIs

Results

In the end, all my returns ended up in the **negative**, but my least red return was using a Decision Tree classifier, using the run once strategy, and having many features (-64.05 % return). I will go into further in depth on how I got my results and what trading strategy I implemented.

Data Mining Task

Input Data

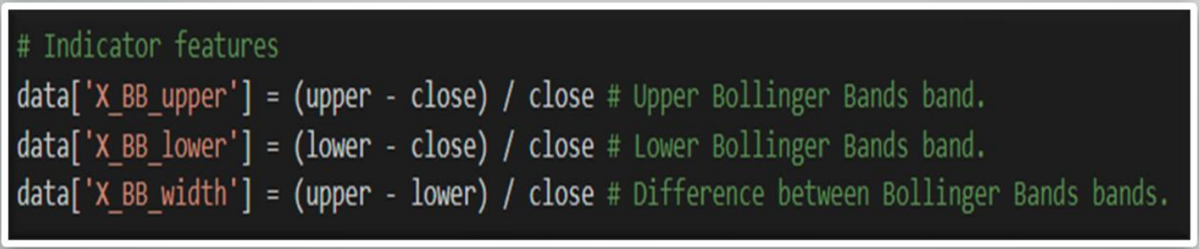
The input data is retrieved directly from the TD Ameritrade API. I collect 20 years of ‘APPL’ trade history with daily candle frequency. I explain how I retrieved this later in the Evaluation Methodology section.

Output data

After getting the price history and creating a CSV file, I now needed to make features from this data. All the features that I could create must come from the data, so I am limited by the data I collect **(Date, Open, High, Low, Close, and Volume)**. This goes to show why data is so crucial, and why the TD Ameritrade API is so powerful. There are many more end points offered in their API, but this is a good start.

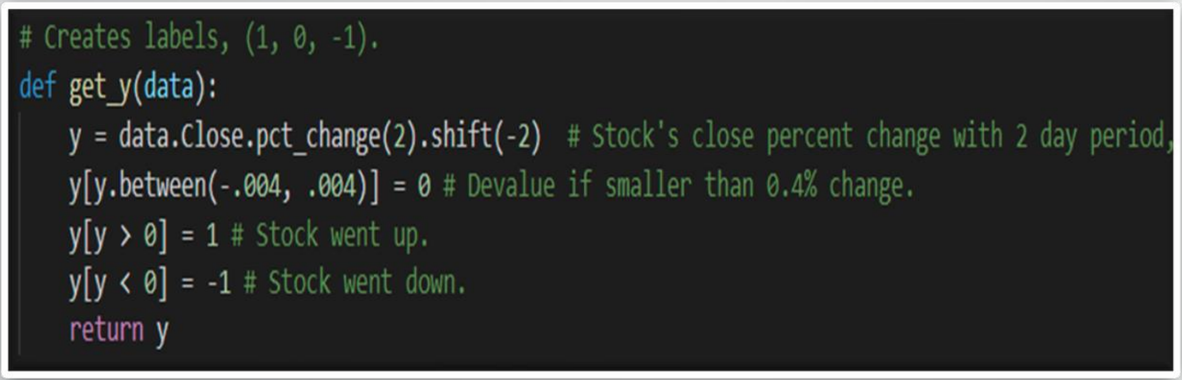
I used pandas to read in the CSV file. I originally only create one feature, the **Bollinger Bands Indicator** to test my classifier. This indicator uses the Simple Moving Average and a Typical Price to calculate an upper and lower boundary for a stock. All of this can be calculated using just the data I collected.

After creating the Bollinger Bands Indicator, I created a new column in my data and I added these features to it, where the header is marked with an ‘X’. I also removed all the rows with NaN values, to avoid any issues further on.



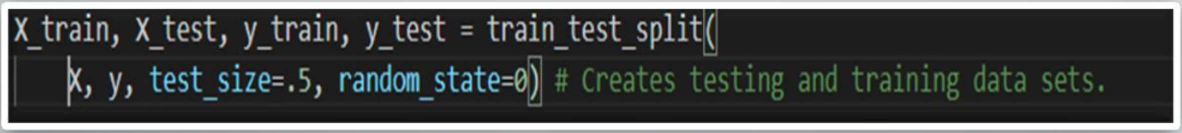
```
# Indicator features
data['X_BB_upper'] = (upper - close) / close # Upper Bollinger Bands band.
data['X_BB_lower'] = (lower - close) / close # Lower Bollinger Bands band.
data['X_BB_width'] = (upper - lower) / close # Difference between Bollinger Bands bands.
```

After creating my features, I then created my labels for the model. The labels are binary, where '1' meant the stock's closing price would have risen in 2 days, while a '-1' would mean that it closed lower. A '0' would mean the stock had no significant change in price (< +- 0.4%). The pandas API makes this easier to do with the pct_change() and shift() methods, which calculate the stocks closing price percent change after a given period.



```
# Creates labels, (1, 0, -1).
def get_y(data):
    y = data.Close.pct_change(2).shift(-2)  # Stock's close percent change with 2 day period,
    y[y.between(-.004, .004)] = 0 # Devalue if smaller than 0.4% change.
    y[y > 0] = 1 # Stock went up.
    y[y < 0] = -1 # Stock went down.
    return y
```

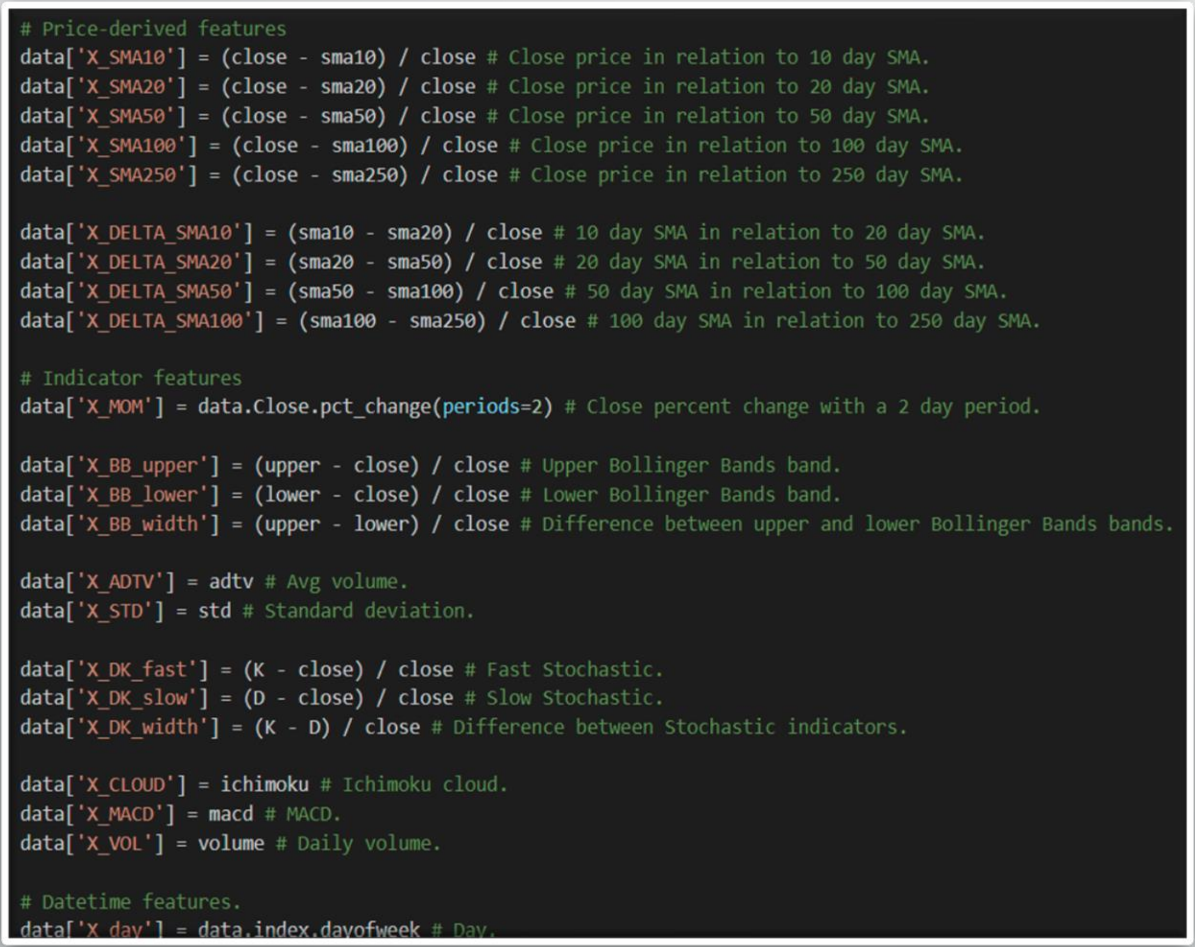
I now have created the features and model labels for my ML classifier to use. The scikit-learn API has a useful feature to create testing and training data called train_test_split(). With this, I can pass the model and labels, along with a random state and test size ratio, and it automatically returns training and testing sets. At this point, the data is all set up for the classifiers to use.



```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=.5, random_state=0) # Creates testing and training data sets.
```

Questions

I began with only one feature, but that arises the question, how many features should I implement? Is having more features necessarily better, or does it just lead to overfitting and extra noise? So, for experimentation, I created a copy of the program I have now. I then implemented as many technical indicators as I could to the copy. Then I created features with all of them. This includes the 20-day Average Daily Trading Volume, the Standard Deviation of Closing prices, the Stochastic Oscillator, the Ichimoku Cloud, the Moving Average Convergence Divergence, the Daily Volume, and the Simple Moving Averages for 10, 20, 50, 100, and 250 days. I was aiming for quantity over quality, so there wasn't any deeper reason I chose any of these indicators.



```
# Price-derived features
data['X_SMA10'] = (close - sma10) / close # Close price in relation to 10 day SMA.
data['X_SMA20'] = (close - sma20) / close # Close price in relation to 20 day SMA.
data['X_SMA50'] = (close - sma50) / close # Close price in relation to 50 day SMA.
data['X_SMA100'] = (close - sma100) / close # Close price in relation to 100 day SMA.
data['X_SMA250'] = (close - sma250) / close # Close price in relation to 250 day SMA.

data['X_DELTA_SMA10'] = (sma10 - sma20) / close # 10 day SMA in relation to 20 day SMA.
data['X_DELTA_SMA20'] = (sma20 - sma50) / close # 20 day SMA in relation to 50 day SMA.
data['X_DELTA_SMA50'] = (sma50 - sma100) / close # 50 day SMA in relation to 100 day SMA.
data['X_DELTA_SMA100'] = (sma100 - sma250) / close # 100 day SMA in relation to 250 day SMA.

# Indicator features
data['X_MOM'] = data.Close.pct_change(periods=2) # Close percent change with a 2 day period.

data['X_BB_upper'] = (upper - close) / close # Upper Bollinger Bands band.
data['X_BB_lower'] = (lower - close) / close # Lower Bollinger Bands band.
data['X_BB_width'] = (upper - lower) / close # Difference between upper and lower Bollinger Bands bands.

data['X_ADTV'] = adtv # Avg volume.
data['X_STD'] = std # Standard deviation.

data['X_DK_fast'] = (K - close) / close # Fast Stochastic.
data['X_DK_slow'] = (D - close) / close # Slow Stochastic.
data['X_DK_width'] = (K - D) / close # Difference between Stochastic indicators.

data['X_CLOUD'] = ichimoku # Ichimoku cloud.
data['X_MACD'] = macd # MACD.
data['X_VOL'] = volume # Daily volume.

# Datetime features.
data['X_day'] = data.index.dayofweek # Day.
```

This created even more questions. I also wanted to know if there is a way to find which combination of features is best, and automate it? Are some indicators only good for certain periods of time, and if so, then is 20 years too much data? If more features isn't better, then maybe parameters are more important?

It might take years of tinkering to understand how all these concepts interweave and affect each other. The task quickly becomes much more complicated, but that is hopefully where ML can help.

Challenges

My original challenge was where to get started. I looked for inspiration by searching up YouTube tutorials that explained how the TD Ameritrade API worked. Luckily, others have also gone down the same rabbit hole I have, and the recommend videos listed directed me to the backtesting.py API (the algorithm is working!). This is where it all started to come together.

At this point I just needed to flush out my ideas involving how can I use this API to answer my questions and figuring out how it works. This meant reading the API's documentation and recreating their examples. I went line by line to understand what each did so that I could better use their API for my own testing.

I also realized that setting up the features seems simple at first but knowing what parameters to use is actually very hard. The first barrier is knowing what the features do, and that involves understanding some complicated statical theories (which I still don't understand fully).

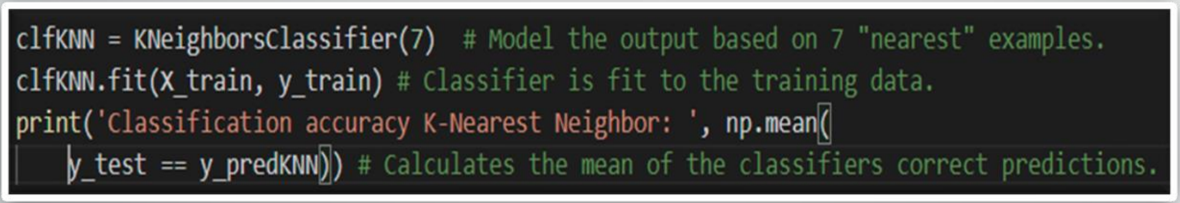
I ended up using a lot of arbitrary values, and this is something that I need to optimize further. For example, when setting up my labels I found the closing percent change for a 2-day period, I devalue any changes that are within a .4% change, and I chose random features to add to my multiple features copy. These were inferred from the examples but are arbitrarily chosen beyond that. I need to spend more time researching how these effect my data and optimize them further.

Technical Approach

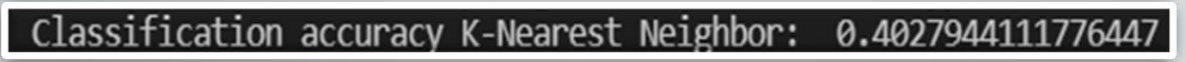
In this section, I will go into further detail on how I trained my classifiers, how I designed my trading strategy to work with Machine Learning, and how I designed a trading strategy that does not use Machine Learning for comparison.

First, I wanted to test my classifier's accuracy after fitting it with the training data. I used a K-neighbors classifier, and I set the parameter to be the 7 nearest examples. This is made extremely easy with the scikit-learn API, as it has many classifiers already created for you to use. It also has a function to fit your classifier to the training data called fit(), and it has a method to get predictions with the trained classifier called predict().

After doing that, to get the accuracy, I used a NumPy array to get the mean of the correct predictions by comparing the testing data and the classifier predictions.



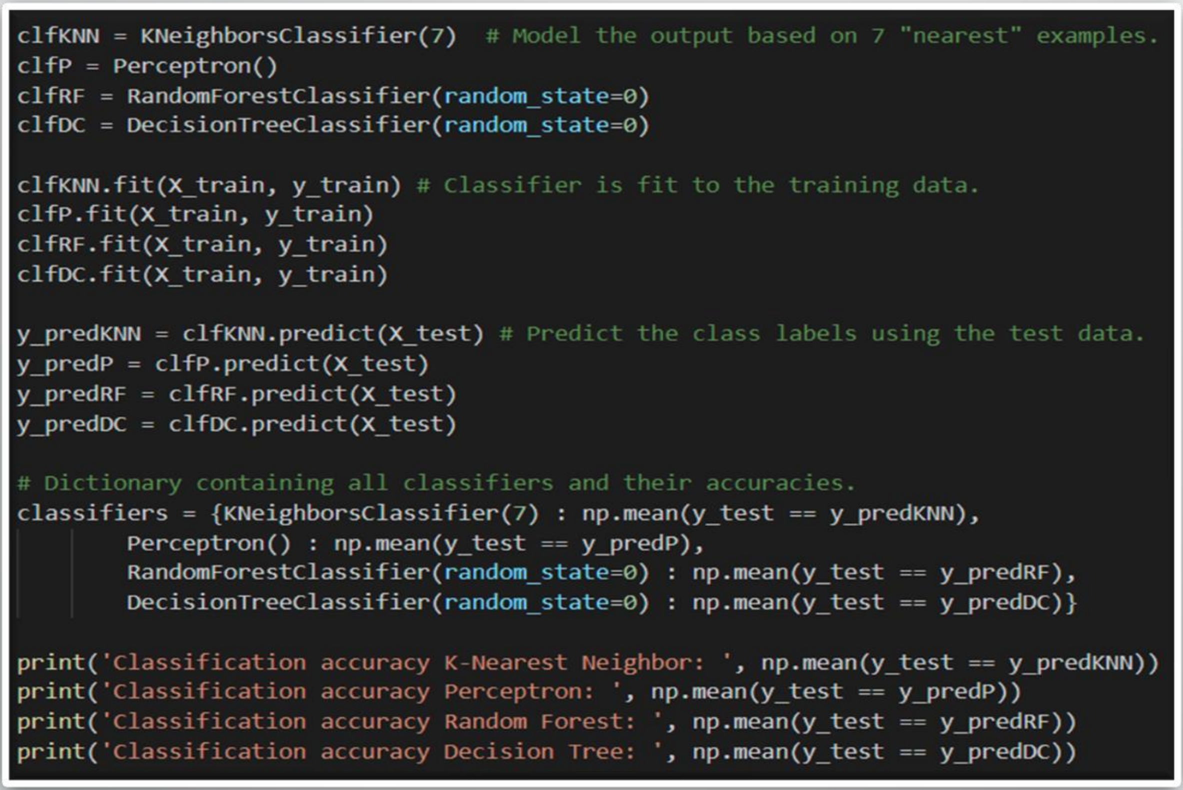
```
clfKNN = KNeighborsClassifier(7)  # Model the output based on 7 "nearest" examples.
clfKNN.fit(X_train, y_train) # Classifier is fit to the training data.
print('Classification accuracy K-Nearest Neighbor: ', np.mean(
    y_test == y_predKNN)) # Calculates the mean of the classifiers correct predictions.
```



```
Classification accuracy K-Nearest Neighbor:  0.4027944111776447
```

The accuracy is very poor, but the test works. The poor accuracy ends up creating weird data at times, but I will come back to that in the Results and Discussion section.

After seeing the poor accuracy, I had a new question. Are there better classifiers for my problem, and if so, which one is the best? So, I added a perceptron, random forests, and decision tree classifier to the tests. Like the features, I was going for quantity here. I also add their results to a dictionary for later use.



```
clfKNN = KNeighborsClassifier(7)  # Model the output based on 7 "nearest" examples.
clfP = Perceptron()
clfRF = RandomForestClassifier(random_state=0)
clfDC = DecisionTreeClassifier(random_state=0)

clfKNN.fit(X_train, y_train) # Classifier is fit to the training data.
clfP.fit(X_train, y_train)
clfRF.fit(X_train, y_train)
clfDC.fit(X_train, y_train)

y_predKNN = clfKNN.predict(X_test) # Predict the class labels using the test data.
y_predP = clfP.predict(X_test)
y_predRF = clfRF.predict(X_test)
y_predDC = clfDC.predict(X_test)

# Dictionary containing all classifiers and their accuracies.
classifiers = {KNeighborsClassifier(7) : np.mean(y_test == y_predKNN),
        Perceptron() : np.mean(y_test == y_predP),
        RandomForestClassifier(random_state=0) : np.mean(y_test == y_predRF),
        DecisionTreeClassifier(random_state=0) : np.mean(y_test == y_predDC)}

print('Classification accuracy K-Nearest Neighbor: ', np.mean(y_test == y_predKNN))
print('Classification accuracy Perceptron: ', np.mean(y_test == y_predP))
print('Classification accuracy Random Forest: ', np.mean(y_test == y_predRF))
print('Classification accuracy Decision Tree: ', np.mean(y_test == y_predDC))
```

I found that the random forest classifier had the best accuracy at 41% when using only the Bollinger Bands feature, and it was also the best at 53% in the multiple features test.

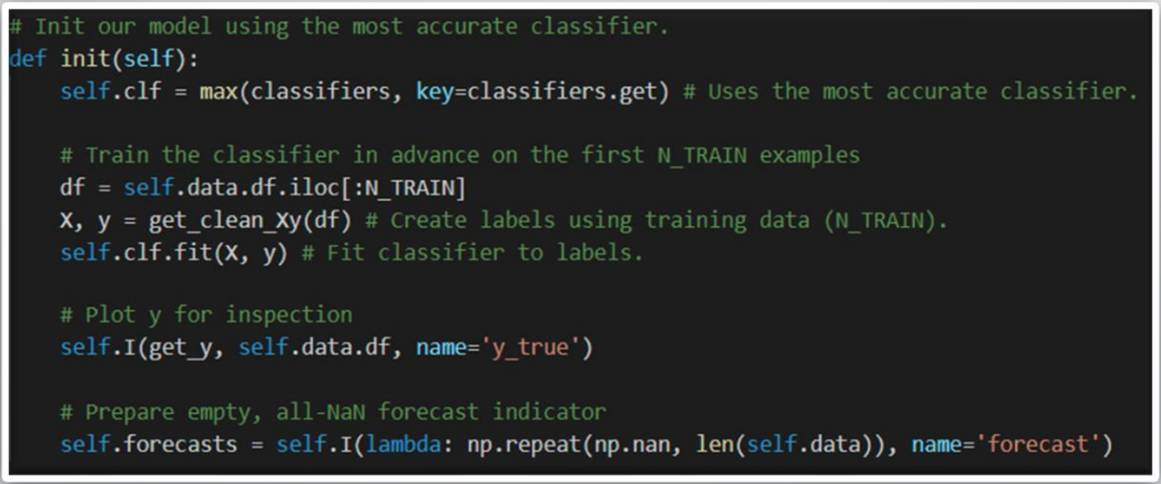
Now that my classifiers are setup, I can finally start creating my ML strategy. I wanted to test out two different strategies for comparison, one where I only train the classifier once at the beginning, another where I use a walk forward strategy. This is like using leave-one-out cross-validation, except I re-train my classifier every 20 iterations on the data as it comes in. Here is another spot where I can optimize by changing the number of iterations.

I'll only be explaining how I implemented the first strategy, as the walk forward strategy inherits from it, and the only difference is that the walk forward strategy calls the fit() method every 20 iterations.

The strategy is setup in two parts, init() and next().

In the init() method, I select which classifier to use using the dictionary I made earlier, and I select the most accurate classifier. I assumed that this would give the best results.

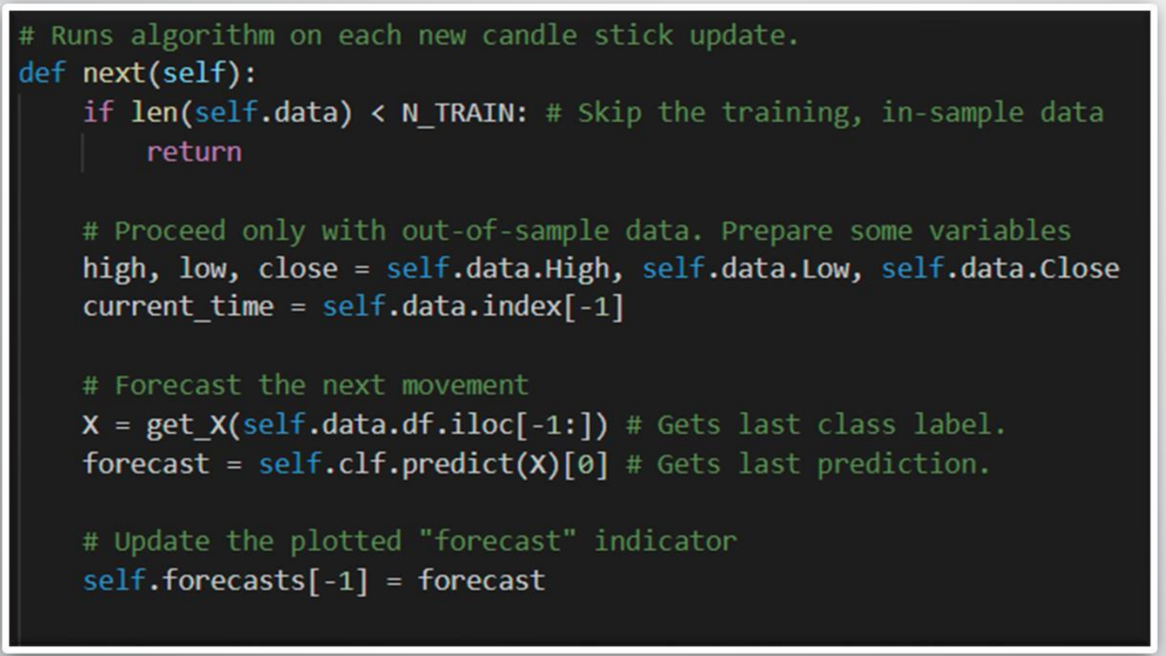
I then fit the classifier on the beginning 20% of test data. Again, another place for optimization. I also setup a forecasts array, which backtest.py will use later to plot in its visualizer (which I show in the Evaluation Methodology section).



```
# Init our model using the most accurate classifier.
def init(self):
    self.clf = max(classifiers, key=classifiers.get) # Uses the most accurate classifier.

    # Train the classifier in advance on the first N_TRAIN examples
    df = self.data.df.iloc[:N_TRAIN]
    X, y = get_clean_Xy(df) # Create labels using training data (N_TRAIN).
    self.clf.fit(X, y) # Fit classifier to labels.

    # Plot y for inspection
    self.I(get_y, self.data.df, name='y_true')

    # Prepare empty, all-NaN forecast indicator
    self.forecasts = self.I(lambda: np.repeat(np.nan, len(self.data)), name='forecast')
```

In the next() method, I simulate getting a new candle (a day of data). This gets skipped until we are past the training data used to fit the classifier (beginning 20% of data).

I then get some variables (the stocks current high, low, close, and date) to use for later. I also get the stock's labels and pass them to the classifier to get its prediction. I then add the prediction to forecast to plot for latter.

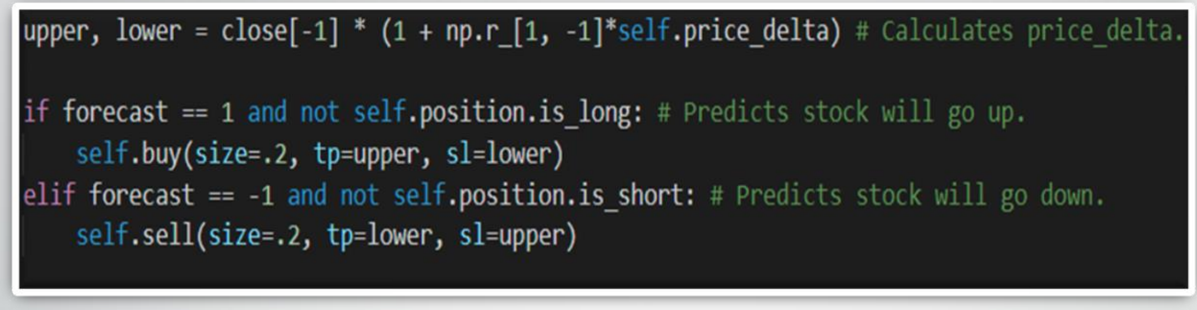


```python
# Runs algorithm on each new candle stick update.
def next(self):
    if len(self.data) < N_TRAIN: # Skip the training, in-sample data
        return

    # Proceed only with out-of-sample data. Prepare some variables
    high, low, close = self.data.High, self.data.Low, self.data.Close
    current_time = self.data.index[-1]

    # Forecast the next movement
    X = get_X(self.data.df.iloc[-1:]) # Gets last class label.
    forecast = self.clf.predict(X)[0] # Gets last prediction.

    # Update the plotted "forecast" indicator
    self.forecasts[-1] = forecast
```

Trading Strategy

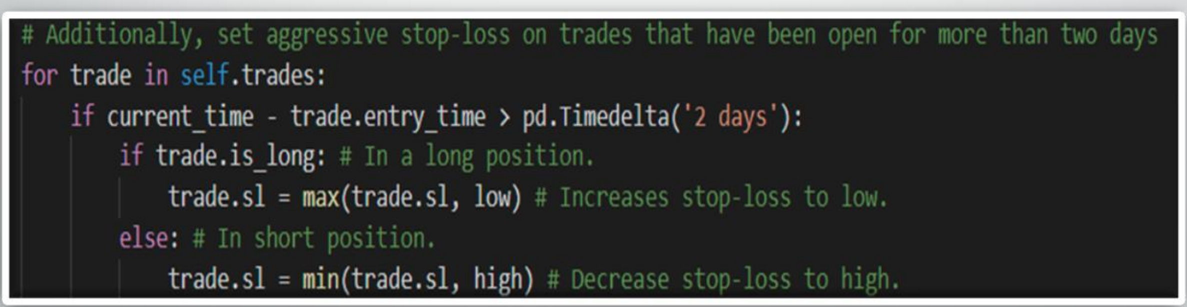
Using an if-else block, I decide to place a buy order or a sell order depending on the prediction. More complexity can be added here for more complex strategies. For example, I wrote a gap reversal strategy which I will describe at the end of this section. If I had more time, I would have liked to have tried to implement that here as well.

To add more safety to my trades, I add a stop-loss and target price with each order. These are included in backtesting.py and are extra triggers to sell a position. Again, more optimization could be added here, but I defaulted to using +-0.4% of the closing price, which was used in the examples.



```
upper, lower = close[-1] * (1 + np.r_[1, -1]*self.price_delta) # Calculates price_delta.

if forecast == 1 and not self.position.is_long: # Predicts stock will go up.
    self.buy(size=.2, tp=upper, sl=lower)
elif forecast == -1 and not self.position.is_short: # Predicts stock will go down.
    self.sell(size=.2, tp=lower, sl=upper)
```

Additionally, I added a another if-else block that makes the stop-loss more aggressive if the trade is still alive after 2-days (same as example period). This is a simple example of how I can still add more features to my ML program.



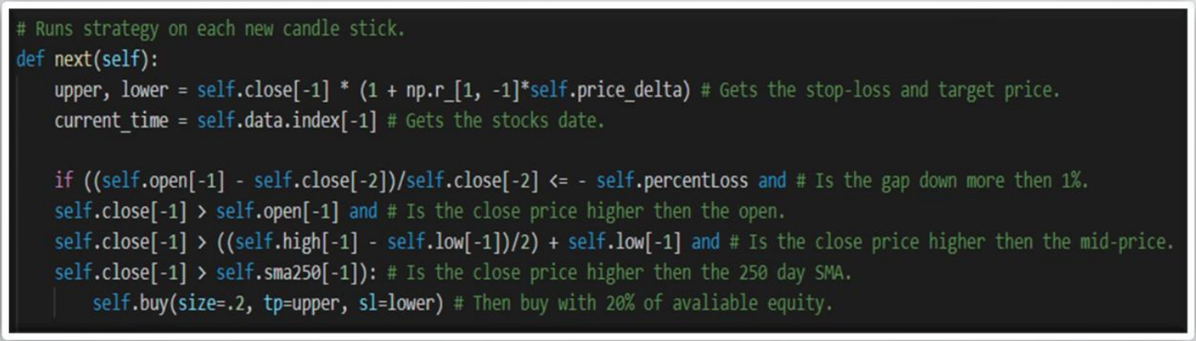
```
# Additionally, set aggressive stop-loss on trades that have been open for more than two days
for trade in self.trades:
    if current_time - trade.entry_time > pd.Timedelta('2 days'):
        if trade.is_long: # In a long position.
            trade.sl = max(trade.sl, low) # Increases stop-loss to low.
        else: # In short position.
            trade.sl = min(trade.sl, high) # Decrease stop-loss to high.
```

The best results I got using this strategy with random forest as the classifier was a -77.59% return (using only B-Bands and the walk forward strategy). Again, this is obviously terrible, however the program is working and returning usable statistics.

Here I thought was a good place to try to answer one of my questions; is ML a **good strategy** for trading stock, and is it even applicable?

I was able to apply a Machine Learning strategy and test it, however it performed very poorly. There could be many reasons for this, but before I try to figure that out, I wanted to see if a simple algorithmic trading strategy would do better. An algo trading strategy is automated like the ML strategy, except it doesn't use a classifier, and so it doesn't change weights or learn.

I decided I would try to create my own gap reversal strategy to test this. It also uses init() and next() in the same way as the ML strategy. It works by looking for days when a stock price opens lower then expected and places a buy order expecting for it to come back up to where it closed the previous day. There are a couple of other safety features implemented here as well.



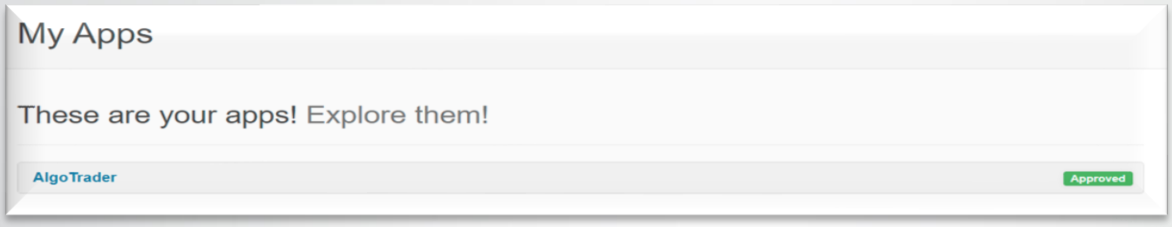
```python
# Runs strategy on each new candle stick.
def next(self):
    upper, lower = self.close[-1] * (1 + np.r_[1, -1]*self.price_delta) # Gets the stop-loss and target price.
    current_time = self.data.index[-1] # Gets the stocks date.

    if ((self.open[-1] - self.close[-2])/self.close[-2] <= - self.percentLoss and # Is the gap down more then 1%.
    self.close[-1] > self.open[-1] and # Is the close price higher then the open.
    self.close[-1] > ((self.high[-1] - self.low[-1])/2) + self.low[-1] and # Is the close price higher then the mid-price.
    self.close[-1] > self.sma250[-1]): # Is the close price higher then the 250 day SMA.
        self.buy(size=.2, tp=upper, sl=lower) # Then buy with 20% of avaliable equity.
```

Running my gap reversal strategy with optimal parameters returned a -4.08% return, which is much better than my ML algorithm . This was much easier to implement and to understand. I think that ML has the potential to be a very successful strategy, but maybe this is a sign that more complex doesn't always mean better.

Evaluation Methodology

Data Source

The source of all my data was TD Ameritrade. To get the data, I first needed to make an account and create an app through the TD Ameritrade developer website.

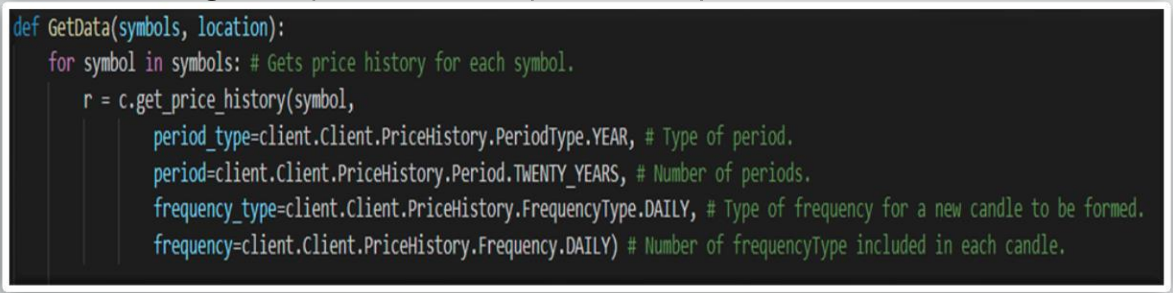


My Apps

These are your apps! Explore them!

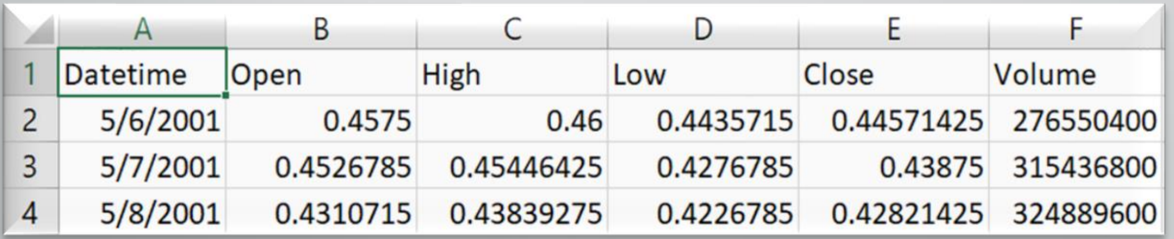
AlgoTrader — Approved

Then, I used the tda-api API to create a token which authenticated my program as me. I needed to install an extra tool, Chrome Driver, in order to access the TD Ameritrade developer website through my program. Once verified, I simply had to make a 'GET request' through the API asking for a stock's price history, which returned the history as a dictionary. I selected 'APPL', and I got 20 years of history with 1-day candles.



```
def GetData(symbols, location):
    for symbol in symbols: # Gets price history for each symbol.
        r = c.get_price_history(symbol,
                period_type=client.Client.PriceHistory.PeriodType.YEAR, # Type of period.
                period=client.Client.PriceHistory.Period.TWENTY_YEARS, # Number of periods.
                frequency_type=client.Client.PriceHistory.FrequencyType.DAILY, # Type of frequency for a new candle to be formed.
                frequency=client.Client.PriceHistory.Frequency.DAILY) # Number of frequencyType included in each candle.
```

I then converted this dictionary into a CSV file and saved it into my directory. Here is an example of what 3 days of data looks like after it has been saved as a CSV file:



	A	B	C	D	E	F
1	Datetime	Open	High	Low	Close	Volume
2	5/6/2001	0.4575	0.46	0.4435715	0.44571425	276550400
3	5/7/2001	0.4526785	0.45446425	0.4276785	0.43875	315436800
4	5/8/2001	0.4310715	0.43839275	0.4226785	0.42821425	324889600

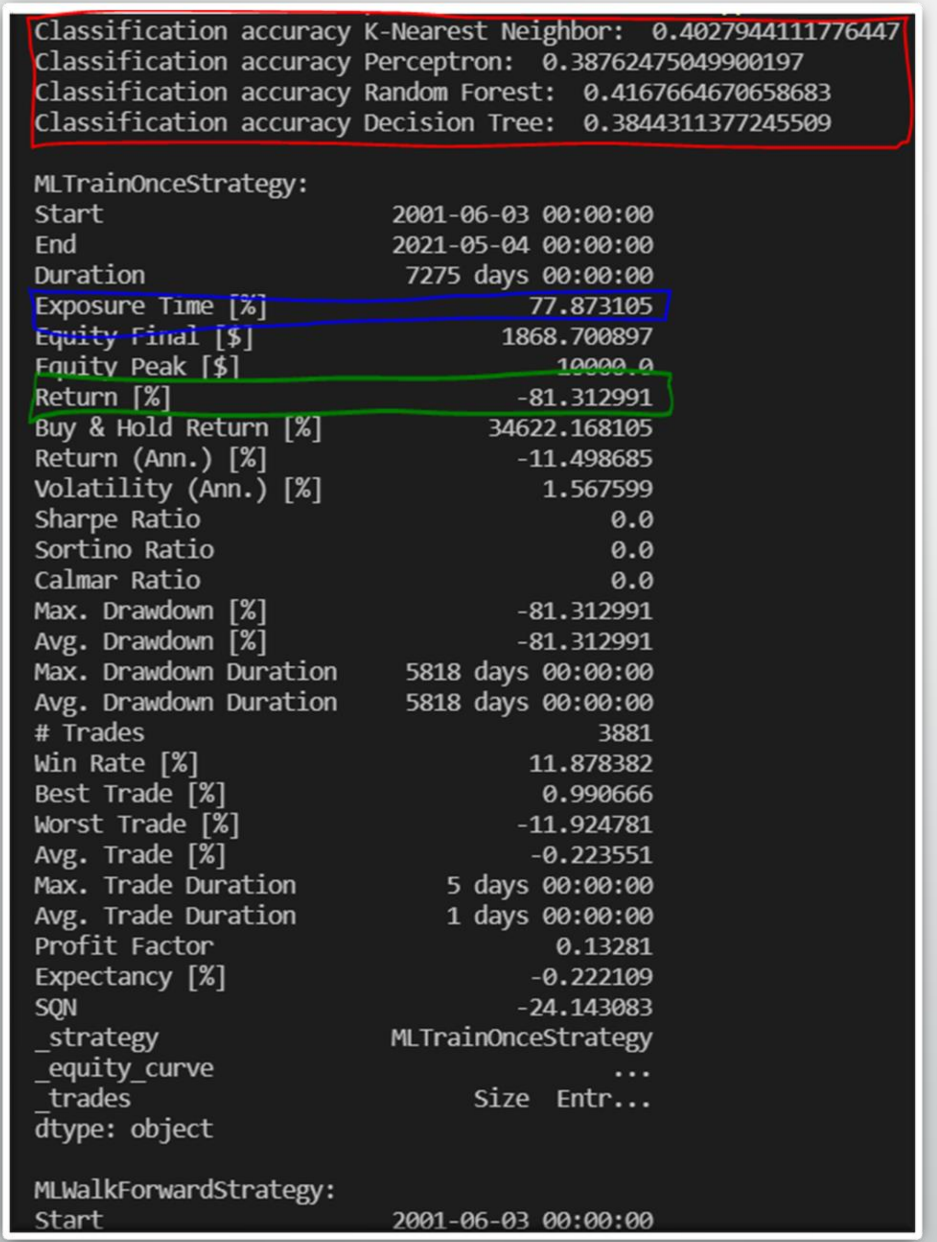
Metrics

I got all my statistics from the backtesting.py and scikit-learn APIs. The criteria I used to rate my classifications were their accuracies. Having a 50% accuracy would is the same as random guessing, and something in the 90% would be a good goal to aim for. Almost all my accuracies were sub 50%, so my classifiers were doing worse then guessing.

I used the Return % to evaluate how my strategies would have done in the real world. This is how much my final equity changed after using my strategy on historical price data. Since the end goal is to make a profit, this is a great way to analyze how well a strategy did. Unfortunately, all my strategies returned a negative %, in otherwards I lost money.

In the next couple of pages, I show some examples of the statistics that are available through backtesting.py. Most of it I didn't use, not because they are not useful, but because I just didn't have the time to research them all. For example, I would have liked to look at the exposure time more. This is important to minimize, because more exposure time on the market means more risk. This could mean doing more optimization with the stop-loss and target prices or adding even more security features.

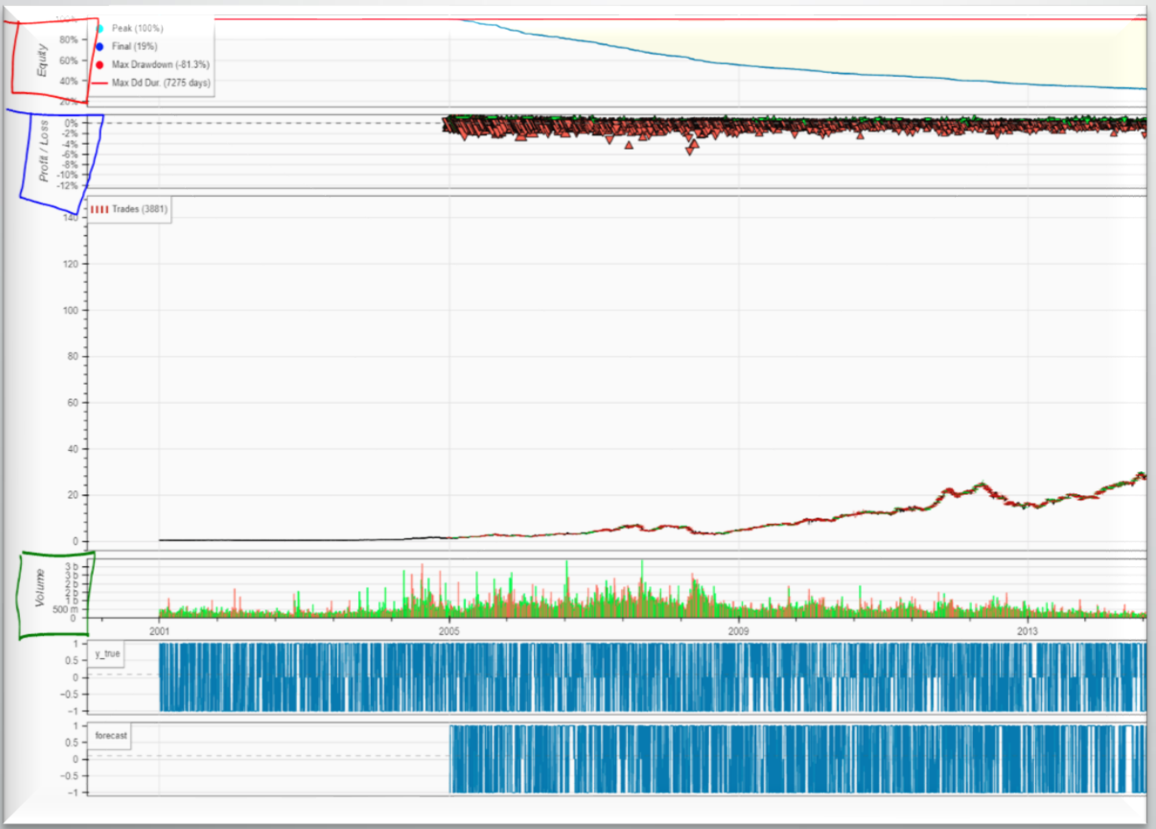
Here is an example output from my program testing the Random Forest classifier using the ML train once strategy. In the red box I display the classification accuracies. In the green box I display the return %. In the blue box, I show the exposure time %.



```
Classification accuracy K-Nearest Neighbor:  0.4027944111776447
Classification accuracy Perceptron:  0.38762475049900197
Classification accuracy Random Forest:  0.4167664670658683
Classification accuracy Decision Tree:  0.3844311377245509

MLTrainOnceStrategy:
Start                     2001-06-03 00:00:00
End                       2021-05-04 00:00:00
Duration                   7275 days 00:00:00
Exposure Time [%]                   77.873105
Equity Final [$]                  1868.700897
Equity Peak [$]                       10000.0
Return [%]                         -81.312991
Buy & Hold Return [%]            34622.168105
Return (Ann.) [%]                  -11.498685
Volatility (Ann.) [%]                1.567599
Sharpe Ratio                              0.0
Sortino Ratio                             0.0
Calmar Ratio                              0.0
Max. Drawdown [%]                  -81.312991
Avg. Drawdown [%]                  -81.312991
Max. Drawdown Duration     5818 days 00:00:00
Avg. Drawdown Duration     5818 days 00:00:00
# Trades                                 3881
Win Rate [%]                        11.878382
Best Trade [%]                       0.990666
Worst Trade [%]                    -11.924781
Avg. Trade [%]                      -0.223551
Max. Trade Duration           5 days 00:00:00
Avg. Trade Duration           1 days 00:00:00
Profit Factor                         0.13281
Expectancy [%]                      -0.222109
SQN                                -24.143083
_strategy                 MLTrainOnceStrategy
_equity_curve                             ...
_trades                          Size  Entr...
dtype: object

MLWalkForwardStrategy:
Start                     2001-06-03 00:00:00
```

Backtesting.py also allows you to plot the data and see it visually. This is what I was referring to when I created the forecast in the next() call. This is another feature that I wish I had more time to play around with. Backtesting.py also has parameter heatmapping and a couple other helpful plotting tools. As you can see, there is a lot of information that could be gained from this, however I would need more time to learn to use it all.

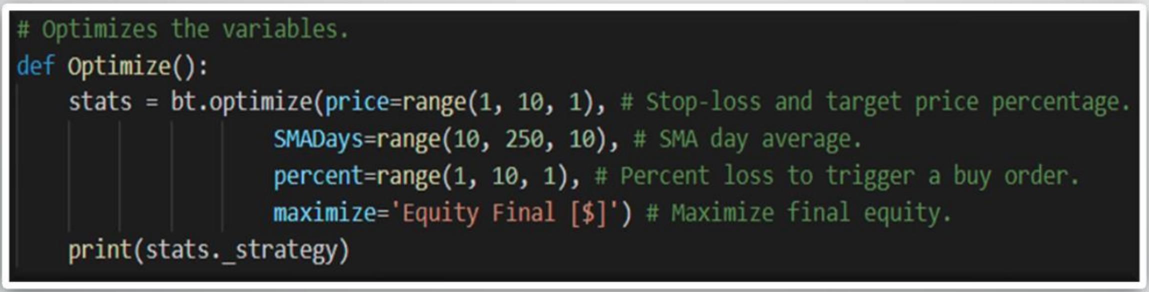
The red box shows my total equity vs time. The blue box shows each individual trade I would have made and if it was profitable or not by the color of the arrow. The green box shows the daily volume. The very bottom two graphs show the test predictions, and the classifier predictions.



I also wanted to briefly mention that backtesting.py has a method called optimize(). I tried it out on my gap reversal strategy, and I wish I could have figured out how to use it with my Machine Learning strategy too.

It allows you to set a range for your parameters, and then it tries to maximize your equity (or anything else you pass it).

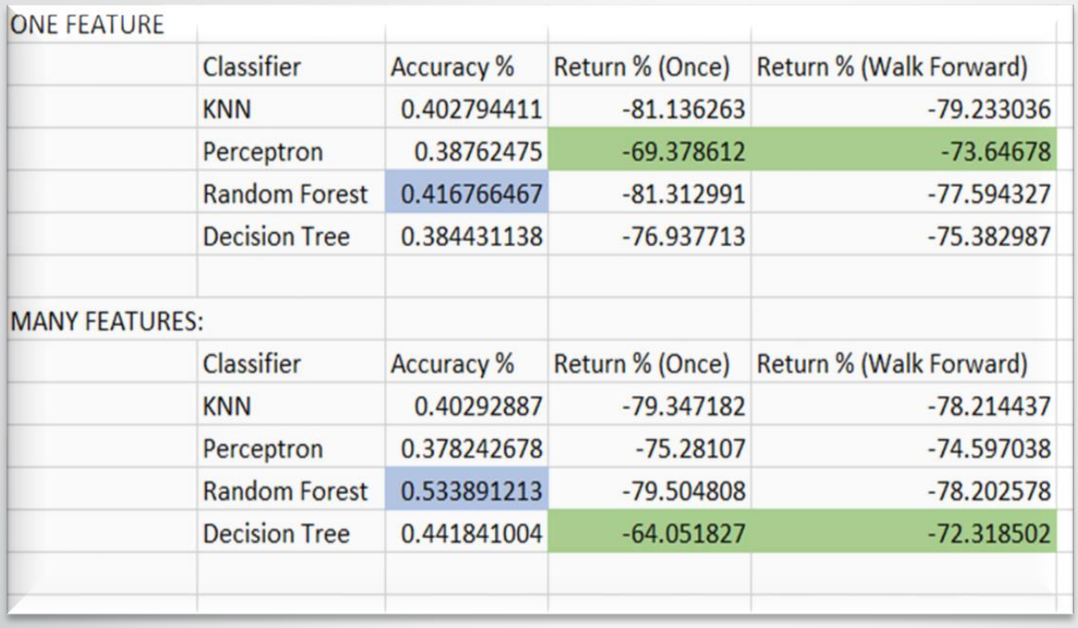
This might have been one of the reasons why my gap reversal strategy did so much better than my Machine Learning strategy. Using this would solve all my problems where I didn't know what parameters to use. I believe that using this tool more would have given me much better results.



```python
# Optimizes the variables.
def Optimize():
    stats = bt.optimize(price=range(1, 10, 1), # Stop-loss and target price percentage.
                        SMADays=range(10, 250, 10), # SMA day average.
                        percent=range(1, 10, 1), # Percent loss to trigger a buy order.
                        maximize='Equity Final [$]') # Maximize final equity.
    print(stats._strategy)
```

Results and Discussion

Here are my results, which I inputted into a excel file to make it easier to view :

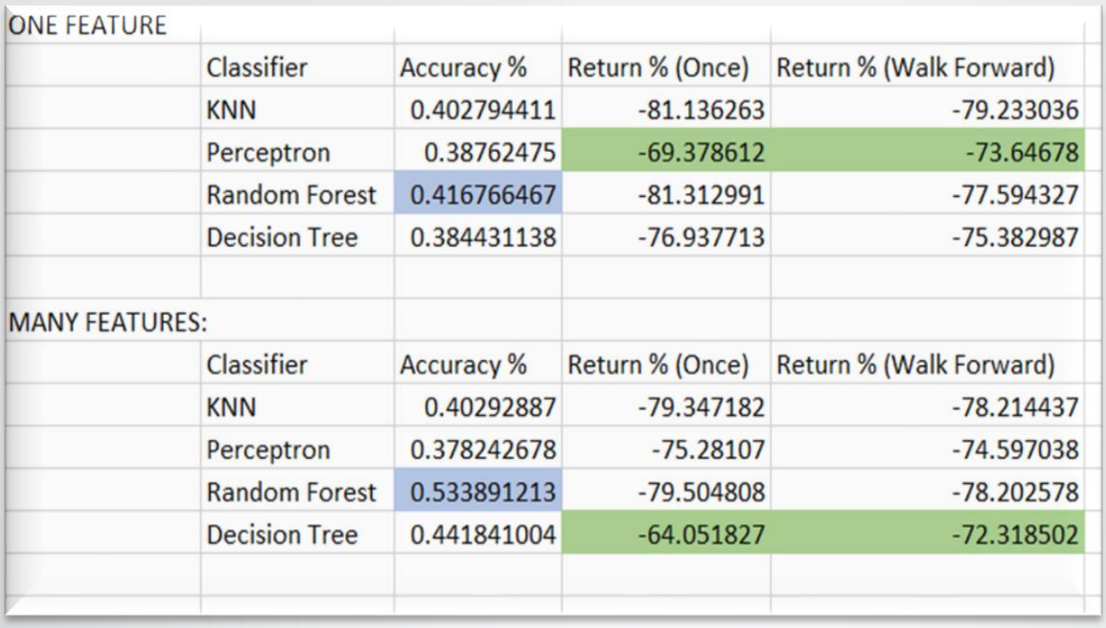


ONE FEATURE				
	Classifier	Accuracy %	Return % (Once)	Return % (Walk Forward)
	KNN	0.402794411	-81.136263	-79.233036
	Perceptron	0.38762475	-69.378612	-73.64678
	Random Forest	0.416766467	-81.312991	-77.594327
	Decision Tree	0.384431138	-76.937713	-75.382987
MANY FEATURES:				
	Classifier	Accuracy %	Return % (Once)	Return % (Walk Forward)
	KNN	0.40292887	-79.347182	-78.214437
	Perceptron	0.378242678	-75.28107	-74.597038
	Random Forest	0.533891213	-79.504808	-78.202578
	Decision Tree	0.441841004	-64.051827	-72.318502

Observations

In blue I highlighted the classifier with the best accuracy score. In green, I highlighted the best return %.

- The accuracy being sub 50% means that the classifiers were doing worse then randomly guessing. This shows that I need to put a lot of work into improving them. This would involve researching why one classifier would work better then another and optimizing the features and parameters. I also believe that this is what leads to a lot of the weird data that I observe.
- Having more features did seem to improve the accuracy. Overall, most of the Return %'s were less negative when more features were added. However, the KNN and the perceptron both didn't improve. This is probably because I did not setup the parameters correctly, or because they are not good classifiers for this type of data. This could also be the poor accuracy as well.



ONE FEATURE				
	Classifier	Accuracy %	Return % (Once)	Return % (Walk Forward)
	KNN	0.402794411	-81.136263	-79.233036
	Perceptron	0.38762475	-69.378612	-73.64678
	Random Forest	0.416766467	-81.312991	-77.594327
	Decision Tree	0.384431138	-76.937713	-75.382987
MANY FEATURES:				
	Classifier	Accuracy %	Return % (Once)	Return % (Walk Forward)
	KNN	0.40292887	-79.347182	-78.214437
	Perceptron	0.378242678	-75.28107	-74.597038
	Random Forest	0.533891213	-79.504808	-78.202578
	Decision Tree	0.441841004	-64.051827	-72.318502

- The accuracy did not indicate which classifier would give the best Return %. The perceptron did better in the B-Bands only test, and the decision tree did better in the many features test. This is interesting, as the assumption would be that a better accuracy would mean better predictions. This is probably because the accuracies are so low across the board that the classifiers were basically just guessing randomly. This is something I need to test.
- In most cases, the walk forward strategy performed better then the only once strategy. This is what was expected, as the walk forward optimizes the strategy more times. However, there are still outliers. The perceptron in the B-Bands only test, and decision tree in the many features test. Interestingly, these are the two classifiers that did the best overall, yet they both performed worse with more optimization. I again think that this is due to the low accuracy, as the rest of the classifiers behaved as expected. This could also be because of overfitting, but I don't think that was the problem.

Lessons Learned

- Is ML a **good strategy** for trading stock, and is it even applicable?

  I would say for now, it seems like more work than it is worth if my goal was to have something that works and that makes money. However, I think that this is a great way to learn more about Machine Learning, and on how to use data in a real-world application. There is still a lot to learn by working on this project, and I plan to continue trying to improve it. However, I would look elsewhere if I wanted something applicable to use on the stock market.

- Is removing the **human** element in trading beneficial?

  Following my previous question, I would argue that it isn't. Having something automated and without human emotion seems like it should do well, however I think the stock market and humans are too intertwined for something automated to fully be able to predict what the market is going to do. For example, I think that the classifiers change in effectiveness because humans trade in the market differently overtime. That is why I need to do more testing on the amount of data I use, as 20 years might be too long of a period.

- What python **APIs** can help with this project?

  I was able to find three very helpful API's. The tda-api API is a wrapper around the TD Ameritrade API which made it much easier to get the trading histories of stocks. The Backtesting.py API is a framework that makes it easier to visualize and infer trading strategies using historical data. Finally, the scikit-learn API was very helpful for testing different types of ML classifiers, like Random Forest, Decision Tree, Perceptron, and K – Nearest Neighbor. Pandas and NumPy were also useful in finding averages and creating arrays.

Conclusion

I learned a lot from this project. I now have a much better understanding of the inner mechanics of Machine Learning. I now know how important the setup process is. This means having good parameters, good data, and good features. I also now have firsthand experience with the impact of poor accuracy, how badly that scales to big data, and how it effects your results. I also have a much better understanding of the stock market because of this project. Researching how technical indicators are calculated has made me much more comfortable with the terminology and the statistical reasoning behind them.

I am happy with the results that I have gotten. This is a good start to my program, and it is in a working state. This will be a good example for me to use in the future as a time where I went from an idea to creating a working application. I can also see the future steps I need to take in my project, which I touched on throughout the paper. I plan to continue to improve this program as it is a great way for me to continue learning about Machine Learning.

Bibliography

- Mitchell, Cory. "Using Bollinger Bands to Gauge Trends." Investopedia, Investopedia, 16 Nov. 2020, www.investopedia.com/trading/using-bollinger-bands-to-gauge-trends/.
- Hayes, Adam. "Exponential Moving Average (EMA)." Investopedia, Investopedia, 29 Apr. 2021, www.investopedia.com/terms/e/ema.asp.
- Hayes, Adam. "Stochastic Oscillator." Investopedia, Investopedia, 30 Apr. 2021, www.investopedia.com/terms/s/stochasticoscillator.asp.
- Fernando, Jason. "Moving Average Convergence Divergence (MACD) Definition." Investopedia, Investopedia, 29 Apr. 2021, www.investopedia.com/terms/m/macd.asp#:~:text=Moving%20average%20convergence%20divergence%20(MACD)%20is%20a%20trend%2Dfollowing,from%20the%2012%2Dperiod%20EMA.
- Mitchell, Cory. "Ichimoku Cloud Definition and Uses." Investopedia, Investopedia, 30 Apr. 2021, www.investopedia.com/terms/i/ichimoku-cloud.asp#:~:text=The%20Ichimoku%20Cloud%20is%20a,plotting%20them%20on%20the%20chart.
- Alexgolec. "Alexgolec/Tda-Api." GitHub, github.com/alexgolec/tda-api.
- "Backtesting.py." Backtest Trading Strategies in Python, kernc.github.io/backtesting.py/.
- "Api: An Unofficial TD Ameritrade Client." Tda, tda-api.readthedocs.io/en/latest/index.html.
- "Trading Strategy: Gap Reversal." Free Trading Strategies Which Can Be Automated: the Gap Reversal Strategy., www.whselfinvest.com/en-lu/trading-platform/free-trading-strategies/tradingsystem/58-gap-reversal-free#:~:text=The%20Gap%20Reversal%20strategy%20is%20based%20on%20a%20precise%20chart%20pattern.&text=The%20strategy%20is%20applied%20on,before%20they%20can%20be%20accepted.
- "Learn." Scikit, scikit-learn.org/stable/index.html.
- "Technical Analysis." Wikipedia, Wikimedia Foundation, 27 Apr. 2021, en.wikipedia.org/wiki/Technical_analysis.
- "WebDriver for Chrome - Downloads." ChromeDriver, chromedriver.chromium.org/downloads.
- Harrison, Onel. "Machine Learning Basics with the K-Nearest Neighbors Algorithm." Medium, Towards Data Science, 14 July 2019, towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761#:~:text=KNN%20works%20by%20finding%20the,in%20the%20case%20of%20regression).
- Donges, Niklas. "A Complete Guide to the Random Forest Algorithm." Built In, builtin.com/data-science/random-forest-algorithm.
- Yadav, Prince. "Decision Tree in Machine Learning." Medium, Towards Data Science, 23 Sept. 2019, towardsdatascience.com/decision-tree-in-machine-learning-e380942a4c96#:~:text=Decision%20Trees%20are%20a%20non,values%20are%20called%20classification%20trees.