Documentation of the Project "Stock_AI"

Contents

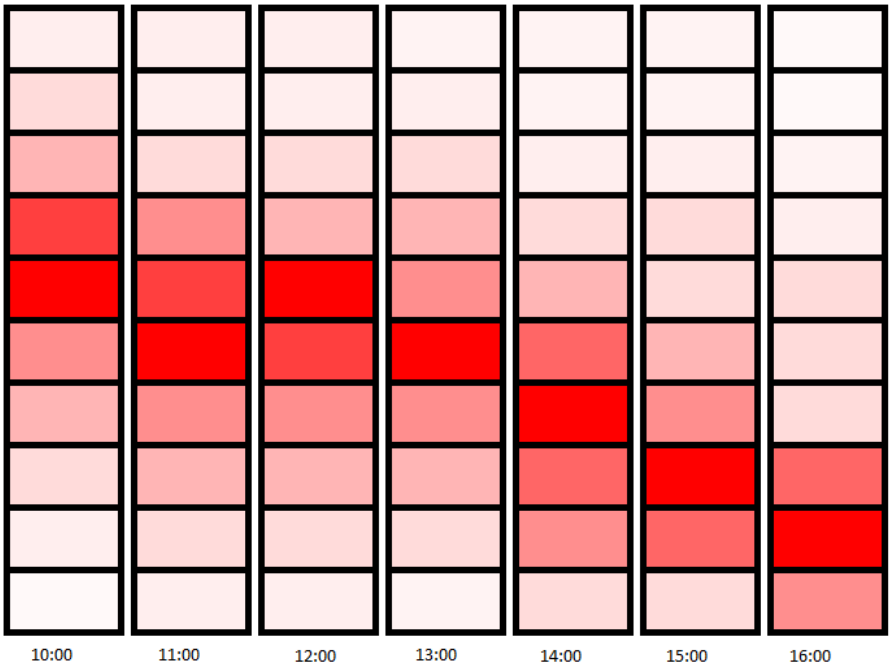
The Goal is the following:

You train an ai to correctly predict where the price of a given stock will be in the next few hours.
In this case, you will do that for stocks, that gapped up more than 20% over night.

The output should be an image, which displays the probabilities for different price-ranges for each whole hour till the end of the day.

So, let's say the at the beginning of the day the price is 10$.
An output could look like:

	10:00	11:00	...	...	20:00
>13$:	4%	...			...
13-11:	11%	...			...
11-10.5:	20%	...			...
10.5-10:	24%	...			...
10-9.50:	16%	...			...
9.50-9:	10%	...			...
9-7$:	8%	...			...
<7$:	7%	...	...	...	



This is what it can potentially look like.

More specifically:

Input: name_of_stock, date with optional startingtime, finishtime, n_borders
Output: graph with hourly prediction, e.g., "Desired_result.png"

The project contains three main parts:

- getandsorteod, gets data with specifique criteria, sorts it for traintf, uses eodhd as the api
- traintf, trains a machine learning model, which trains on the whole data
- stockai, takes the trained model and makes a prediction with given inputs

The whole project is created in two main folders.

stock_ai_scripts, which contains the scripts and helper-files that run the project.
stock_ai_data, which contains the data for training and for the model.
The path names to these two folders, can be adjusted, in "path_names.txt" in the script folder.
You must modify them accordingly in order to run the programs.

The project uses the eodhd-, polygon-API. However, you can adjust the code so that you can use your API provider of choice.

getandsorteod

Data must be downloaded, filled up & normed.

Should it contain Fundamental data? -> Not now, can add it later if needed.

How many datapoints should it contain? -> A lot, you can use fewer later if you want (e.g., 7 Trading days)

This part consists out of 4 tasks:

1. Get the list_of_events
2. Get the raw_data
3. Clean up raw_data
4. Bring it into the correct form to train the model

1. Get list_of_events

First a list of events gets created.

An event happens when a stock meets the criteria.

The criteria are the following that the stock has gapped up overnight more than 20% (Gap > 20%).

The list_of_events contain for each event: stock_name, event_date

So, for example, if Apple would have gapped up overnight by 23% on the 2nd of July 2015, then I want my list to include: "AAPL", "2015-7-2"

To create this list, you first need a list of all stock names that exist (ListOfTickers.txt).

Then you download the daily data for each ticker (dailydata.pkl).

Timeframe: 2004-1-1 -- 2022-11-09

Then you can filter out the events and create list_of_events.pkl

2. Get raw_data

Using list_of_events you can now download the datapoints for each event.

The timerange is -5 to 5 days. Where day 0 is the event day.

That means, that we want price data of the 5 previous days before the event_day, 5 days after it happened, and data from the event_day itself. So, 11 days in total.

1 Minute time intervalls get downloaded and because trading hours (including Pre/Post Market) are from 4:00 - 20:00, that means a maximum of 10560 datapoints per event (16 hours * 11 days * 60 = 10560).

One datapoint includes:

'timestamp', 'gmtoffset', 'datetime', 'open', 'high', 'low', 'close', 'volume'

A challenge that arises is that when making an api-request, you need to specify the exact unix_timestamp from when to when you want to get the data.

The problem is that we only want to include actual tradingdays in our 11-day timespan.

That means that we must factor in weekends, holidays and other occurences where no trading was done. In order to do that, I created a list named "non_trading_days.pkl" in which every non trading day from 2004-01-01 -- 2022-12-26 is contained.

Note that often the stock market closed earlier (01:00 PM). This is not reflected in the list.

Another complication is the conversion from a string (e.g., "2022-01-01") to the unix_time_stamp required by the api.

In addition, the yearly shift from Eastern standard time to Daylight savings time also must be considered.

To keep the code readable, I created "helper.py", in which these helper-functions are written.

Now you can download the raw data if you have bought a valid api key.

I used EODHD as my api-source, but often theses api requests work in a similar way, so if you have a different api you dont need to change the code very much.

I recommend downloading in batches, as to not overload the system and the api.

I split it into 41 batches, but you can easily change it by giving different arguments to the function.

Now you have raw_data.pkl

3. Clean data

Now that we have the raw_data, let’s look at it and clean it up.

First, the data has a lot of missing datapoints.

You would expect 10560 entries for each event, but as it turns out, that is never the case.

Reasons for that can be that a trade hasn't happened in every minute of the day.

This makes sense, because almost all the tickers are Small Caps, which typically dont have a lot of volume and liquidity.

Second, the quality of the API is not great, often data that should exist, doesn't.

From the 31037 recorded events, only 9747 events have any datapoints in them.

And only 4375 events had more than 3000 datapoints in them.

This is quite disappointing because it already severly limits the capabilities of the model.

Having only ~4k examples is not enough to train a complex neural network.

To clean up, first we take each batch of data which we downloaded and delete all events with less than 3000 datapoints.

Next, we cut off all the datapoints after the eventday and then finally unite them into one "presorted_0.pkl".

Why did we download data after the eventdate in the first place?

Well in the future, if we want to add to our project, we can make further predictions and dont must download the data again.

For example, we can study the price-action in the following days of the Gap (event_day). For now, this isnt our concern.

Now each event only has a maximum of 6*16*60 = 5760 datapoints.

However most events have datapoints missing, which we now need to fill up.

For example:

When there is data for 8:35 o'clock but not for 8:36, this datapoint will be filled up with the same OHLC values, but with a volume of 0.

Also, for some events the gap is below 20%. In these instances, there occured some errors either in the data request or in the processing afterwards.

We just filter out these edge cases and are left with clean data of 4060 events.

We can plot each event with "plot_cleaned_data" and see the charts that the ai model will be trained with for ourselves.

4. Data to np

Now we need to bring the data into the final form.

We normalize our values, so that only relative changes in stockprice matter.

Also, we scale the volume values down logarithmically to avoid big volume spikes.

Then we add the correct outputvalues that the model should predict for given input values.

As said earlier, we try to predict the stockvalue for each whole hour till the end of a regular trading day (16:00).

This means, depending on startingtime, we have different percentages that the stock moved.

We also reduce the task down to a classification problem. That means we create 10 price ranges for each hour, to which the model makes a prediction.

Each price range has the same amount (406) of events in it (that ensures, that the probabilities for a new stock are even). Then for each event we can calculate, in which price range it falls.

For 16:00 o'clock the borders look like this:

[-23.09, - 16.50, -12.01, -8.64, -5.27, -1.70, 2.24, 7.81, 20.21]

0 1 2 3 4 5 6 7 8 9

So, if a stock has moved up 4% from starting time, it falls in the price range 7 and therefore gets this value as the correct outputvalue.

Note that you can change the startingtime, so if you want to train the model starting from e.g., 11:00 o'clock you can simply change it.

In theory it should predict the early hours better because there is less time for the stock to move.

However, you also would need to train the model, now with these different input-/ouputvalues and borders.

To finish up, we cut the inputdata after the startingtime because any data after it of course wouldn’t be available in a real world scenario where we don’t already know all future price action.

Now we have final_data_09_30.pkl, with which we can train our model.

traintf

The model can be trained using the train_tf.py file. The packages needed are tensorflow, numpy, pyplot.

Before the model gets trained, the data needs a few final adjustments. That is because we have a few hyperparameters and we want to change the data accordingly.

One is to specify the number of days we include. We can have a maximum of 5 days before eventday. If we want less, we need to cut it with the parameter n_days_chosen.

The other parameter is the time intervall. Our data is in 1min time intervalls. If we want longer, we can specify it with n_time_frame.

The last parameter is n_data_points. It specifies what one datapoint contains, [O H L C V Time]. So, if it is 1, we train only with the open values. We can also look at the inputdata, to see if everything is working correctly with the function "showdata(starting_time)".

Now the model is ready to be trained. I tested it with different hyperparameters and different types of layers.

The preexisting script train_tf.py contains the configuration, which resulted in the best and most consistent results.

However, these results are not too great.

The average valuation_accuracy is around 15-18%. That means that the model does indeed work a bit (the random case would be 100% / 10 = 10%) (Dividing by 10, because we have 10 price ranges that we want to classify).

However, this is still a pretty low result. Plus, the variation between epochs and training runs are quite big. The main reason for the poor performance is probably the lack of high-quality data and the stock price action with its inherit randomness. There needs to be way more datapoints and events to train an AI which is complex enough to predict these types of scenarios.

After training a plot appears, that shows the development of the training and valuation accuracy over the number of epochs. If the graphs diverge from each other, that means that overfitting is happening.

There is also the option to save the model, which can be selected with traintf(save=True).

The script also makes a copy of itself and saves this copy in the stock_ai_data folder. This is so that good configurations dont get lost while experimenting around.

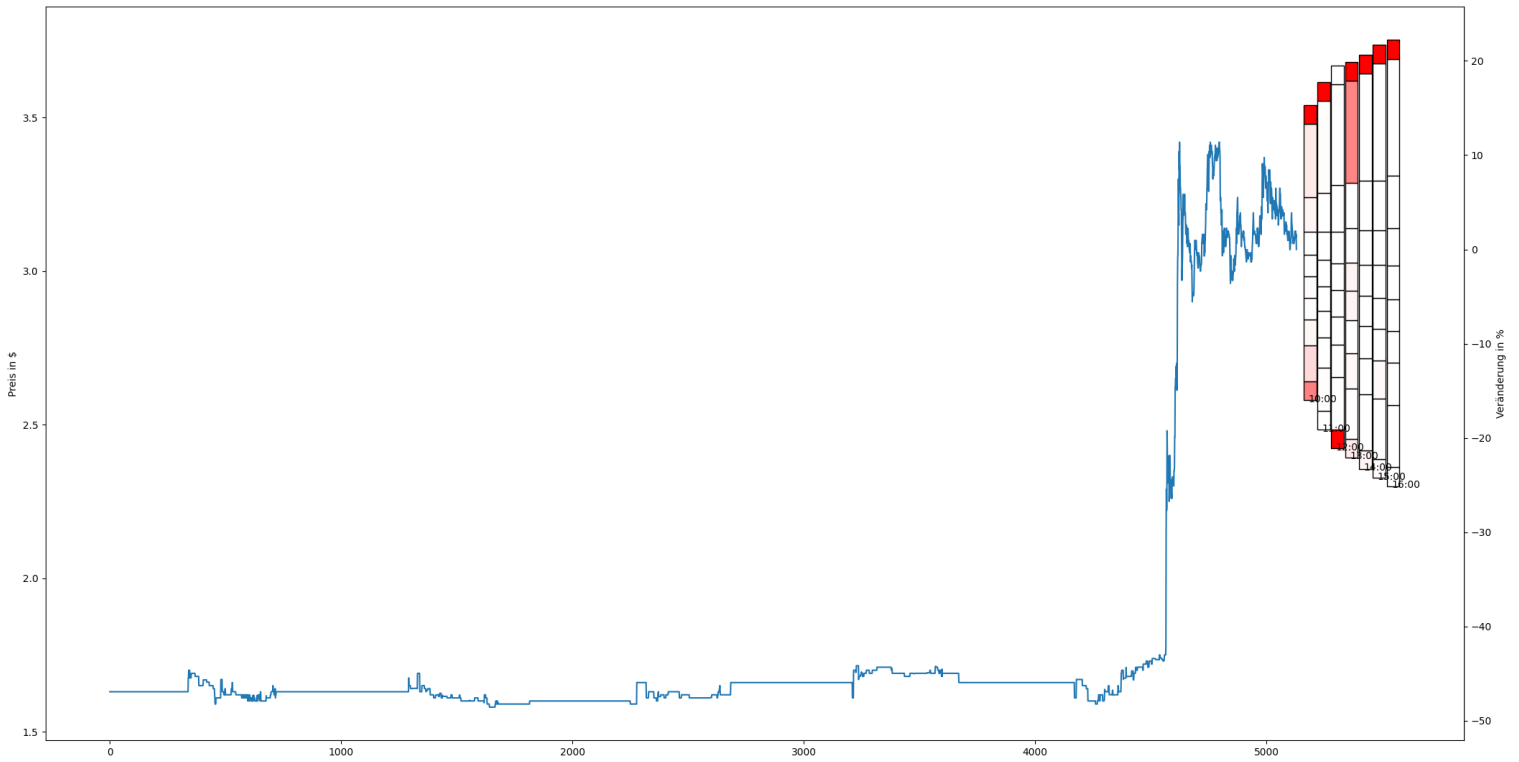
stockai

This is our final program that we want to execute if we want to predict price action in the future. The main function stockai(tickername, date) plots out a diagram, that shows the likelihood for the different price ranges.

This can work with real time data, so if a stock has gapped up more than 20%, then you can prompt this function in real time, e.g., at market opening.

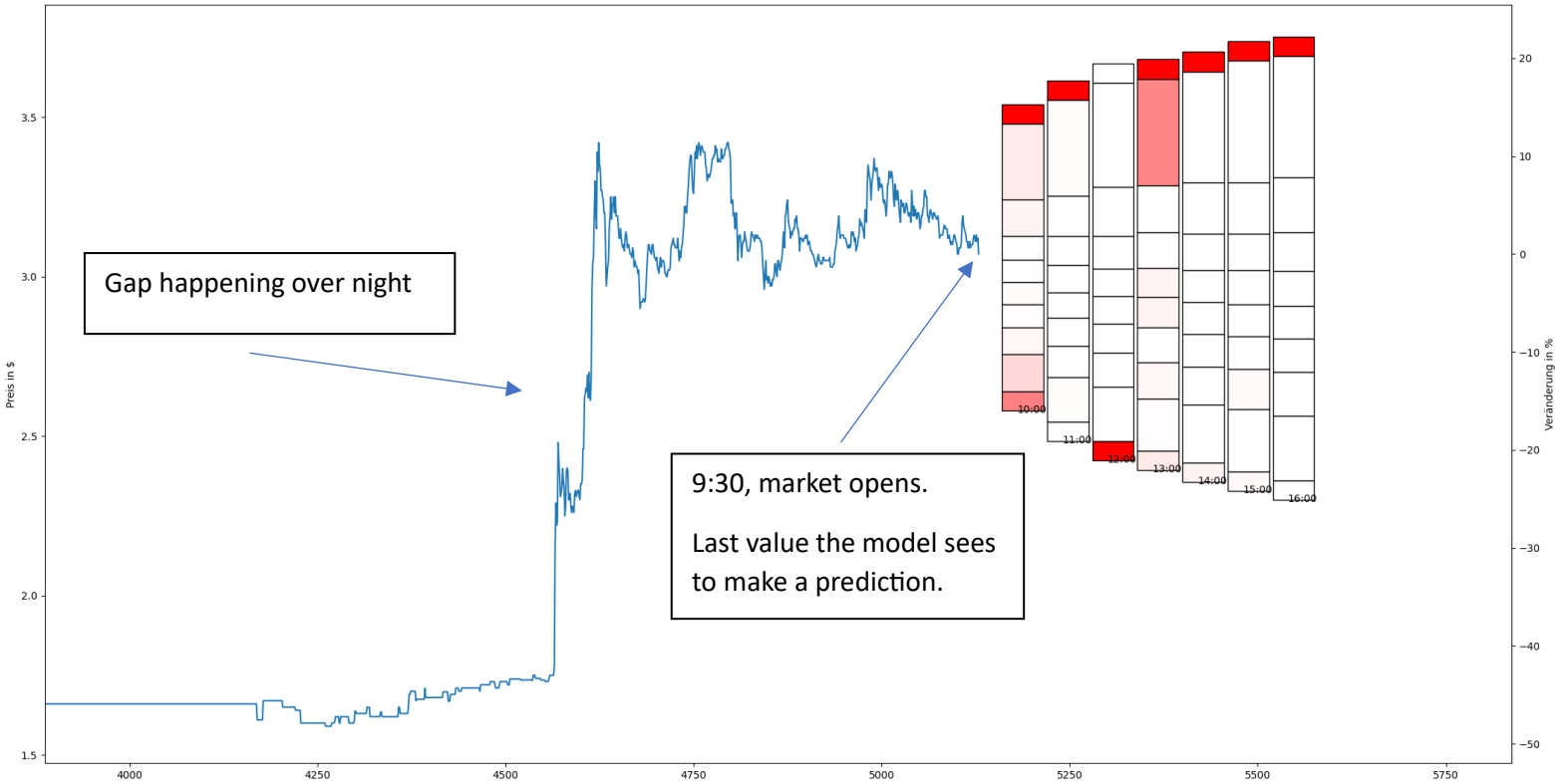
However, you need to include your own API Token in the source code, which you can get with the Polygon_api. If you want free real time data, you need to adjust the code and work with e.g., the yfinance-api instead of the polygon-api.

The code first downloads the data, then cleans and prepares it for the models we have trained earlier with traintf(). With these models and the data, it then makes a prediction and shows it in a plot. You can also back test what the model would have predicted for an event in the past. The factual pct change in this case gets calculated and showed along with the predicted value.



Actual Output for the Stock “SIEN” on the 9th of June 2023.

The redder the price range, the more likely the model is that the price will be in this range for this hour.



Zoomed in on the actual Output for the Stock “SIEN” on the 9th of June 2023.