Disclaimer: Coursera, StackOverFlow, YouTube and GenAI, NewsAPI.org, FinBERT via HuggingFace and Optuna were used during the coding process.

Purpose of this code:

The key purpose of this code is to investigate the extent to which the Efficient Market Hypothesis (EMH) exists. In broad terms, EMH suggests that we (the general public) cannot predict the stock markets, as all publicly available information is simultaneously factored into the markets. There are three main types of EMH: weak, semi-strong and strong form. Due to the inaccessibility of 'insider' information, I could only investigate the weak and semi-strong forms of EMH.

To do this, I made use of neural networks (LSTMs and CNNs specifically) to learn patterns from the stock price data and sentiment data. To obtain sentiment scores, NewsAPI.org was used to scrape headlines and these were input into the FinBERT NLP model from HuggingFace. This, paired with the average stock price (average of open and closing price for an interval), formed the features my model was trained on.

Delving into the code:

What was done:	Why it was done:	Any major errors whilst implementing
Use of yfinance stock library	- Access of up to 6 months of stock data - Easily implementable - Free of cost	
Use of LSTMs	- LSTMS are RNNs that are capable of learning patterns from previous data (stock price data and sentiment scores in this case), in order to make future predictions. - I made use of single layer LSTMs and stacked LSTMs and evaluated them using the validation mean average error (discussed further on) - It is important to note that we can stack many layers of LSTMs together, but realistically, the effectiveness stagnates	

	after 1-2 layers. Whilst we could test this out by creating a model with more than 2 layers, I didn't do this as it'd waste time.	
Use of CNNs	- Whilst CNNs aren't conventionally used for stock price prediction, they can be good at picking up complex or short terms patterns.	
Use of val_mae as a metric	- Val_Mae stands for the Validation Mean Absolute Error - A model can be deemed 'good' if it performs well on the validation dataset (the data that the model wasn't trained on/ is unseen). - In this investigation, I used the classic 80/20 split for training and validation data. - I used the in-built 'val_dataset' function in Keras for ease and to avoid data leakage. - The lower the val_mae value, the better the model is thought to generalise.	
Use of random search	- In theory, grid search is one of the best methods to choose hyperparameters because it evaluates all possible combinations of hyperparameters in the range you provide - However, random search is known to be the more efficient choice, since it only tests a set number of combinations (predetermined by you), saving time	
Switching from random search to Optuna	- Whilst random search is the more efficient way of finding the optimum	

	hyperparameters, it does so in a random way, without learning from previous combinations (i.e. from whether they generate a relatively lower or higher val_mae value) - Optuna follows the same concept that random search does with respect to trying a predetermined number of combinations, but it makes use of Bayesian optimisation (mathematics) to learn from previous combinations and improve the choice of new combinations	
Use of MinMax scalar	- The magnitude of features can lead to a disproportionate effect on the output (e.g. input price may affect the output more than sentiment score, since it is likely larger in magnitude than sentiment score) - MinMax scalar standardises the values of price between 0 and 1 to ensure that each feature has an equal effect on the prediction.	- Upon running my code, I realised that when I was rescaling (inverse transforming) my predictions, I was using the features scalar to do so. This was causing an error because it was expecting two inputs (price and a sentiment score), however, since the only output is price, only one input was being fed into the scalar. Consequently, I had to amend the scalar I was using and had to create a separate prices scalar (only expects one input which is the price) to fix this issue.
Use of random walk as a baseline	- A baseline model is needed to compare the effectiveness of more complex models, such as stacked LSTMs, against a basic model, like a random walk.	- Upon initial implementation, the random walk model was providing a better directional accuracy than the other models. However, this was because I was predicting

		our current value to be the last value, or the price at time n+1 = the price at time n. - This is unrealistic because noise is a common occurrence and impacts the next value at a given timepoint. - Thus, I amended my random walk model to include ‘white noise’, which means that a price at time n+1 is the price at time n with some noise included. This fixed the problem by being more representative of the real world where one price isn’t necessarily dependent/ the same as the last.
Use of windowed dataset	- We define a ‘window size’, which is the last x number of datapoints up to the datapoint at time n. - Our model tries to learn a pattern from the datapoints and then attempts to make a prediction for the datapoint at time n+1. - This method is used to aid our model in adjusting weights of parameters. - In some sense, this is a way to get many small sets of data from a large dataset, helping our model to generalise.	
Use of early stopping/checkpoint	- If we run too many epochs (e.g. 100), we risk overfitting (in some cases) and thus choosing the wrong model. - EarlyStopping ensures that if the val_mae doesn’t change or is higher for a	

	certain number of epochs (10 in my case), then we can stop running further epochs and evaluate based on this val_mae. - If the val_mae is lower, the epochs continue to run (as there is a chance of improving our val_mae value even further).	
Use of two features	- In order to test for the semi-strong form of EMH, the sentiment (in the form of sentiment scores) from news articles must be factored in as a feature, alongside the average price of stocks. - Using these two features, I attempted to see if we could make predictions of future prices based on our models learning from the price and the news available at that time. - Due to the scarcity of news articles, there were some prices that'd share the same sentiment score, which would update when any new news was available.	
Use of NewsAPI + FinBERT	- To access the news articles, I implemented an API (NewsAPI.org). This gave me free access to the past months' news articles, as well as 100 calls to the API (which was enough for my proof-of-concept model). Namely, I extracted the news headline and date/time this was published. - In order to access headlines relevant to my ticker, I had to create a list	

	of keywords manually relevant to that industry so we could find relevant news articles. - Once obtained, I inputted each article headline into the FinBERT NLP model to obtain a sentiment score.	
Use of directional accuracy	- Whilst I was making predictions for future prices, this isn't necessarily the most important outcome, since when an individual invests, they don't bother about the price as much as they do about whether the stock price will increase or decrease. - The simplest way to encompass this was through calculating a directional accuracy value.	

Results:

I made use of some stocks from different industries, mainly in technology. The directional accuracy results are as follows:

Stock	Industry	Run 1 (%)	Run 2 (%)	Run 3 (%)	Mean (%)
Apple	Technology	50.9804	37.2549	48.0392	45.42
Microsoft	Technology	53.92	40.20	N/A	47.06
Lloyds	Finance	52.10	45.92	48.24	48.76

*N/A because Google Colab crashed when running.

<u>Discussion and Improvements:</u>

At first sight, these results don't look overly promising. Randomness, at least for this context, is deemed to be at the 50% directional accuracy level. Any percentage above this is considered to be better than random, whilst anything below this is considered to be worse than random. Many of the results in the table above seem to be below the 50% mark, and thus worse than random by our definition. However, there are also some percentages that are above the 50% mark, suggesting that there may be some predictive signal present.

The key takeaway from this is that we can't draw any conclusions about EMH without much more thorough testing, which wasn't possible at the scale I created this POC on.

The two main areas which restricted testing were a lack of computational power and API constraints. The following delves into these in more depth:

1) Random Search (number of combinations)

Due to the lack of computational power (i.e. testing on a laptop as opposed to AWS or Azure servers), I had to limit the combinations I tested in order to keep testing times reasonable and prevent crashes.

This meant that I could only test for 30 combinations of hyperparameters – a relatively small number when trying to find the optimum combination. The chances are that the most optimum set of hyperparameters wasn't covered by those 30 combinations.

However, had I tested any more combinations, the training time would be much longer, as each set of hyperparameters had to be applied across three models, repeated three times, and run for multiple stock tickers. Consequently, the results gained using these hyperparameters weren't accurate enough to support firm conclusions.

Thus, in the future, I'd want to test for many more hyperparamters to ensure that my random search method is getting the best or near the best set of hyperparamters.

2) NewsAPI - The keywords used to search and the 'Free Tier' limitations:

NewsAPI was used to fetch news articles and their metadata based on information, such as sector keywords and the stock symbol (e.g. AAPL for Apple, etc.). For the sector keywords, I made use of a small, generic list relevant to that industry. However, the limited scope of this list likely affected the retrieval of relevant articles, directly affecting the quality of the sentiment score.

Thus, when testing for many stocks, it'd be easier to use an LLM, like OpenAI (GPT) for instance, to be able to input the sector and stock ticker, and receive an output of the keywords. This'd be more efficient than having multiple individual lists per sector, which takes up a lot of space in the code and isn't necessarily as comprehensive as a list that OpenAI could provide. Additionally, it'd easily provide a large list of sector-relevant words, solving both problems.

Furthermore, the ‘Free’ tier of NewsAPI is limited to the past month of news articles and their metadata. This severely limits the number of articles present (and thus availability of sentiment scores), which influenced the time frame used when collecting stock information. It meant that I had to limit my stock data to a month and I had to use much smaller intervals to ensure I had sufficient data points. As a result, I had very granular data, which isn’t ideal for pattern detection of LSTMs, which try to learn from longer term patterns. This is yet another reason why the results I obtained cannot form a conclusion on the EMH and it’d take more resources from NewsAPI to improve my experiment.

3) Using more stocks and reproducibility of data:

This is similar to the first point in that we’d like to repeat this experiment over at least 50-100 stocks from varying industries to see if we can observe a generalisable pattern, or whether EMH holds stronger in certain industries than others. Additionally, we’d want to perform repeat experiments multiple times per stock to check for reproducibility for individual stocks. Once again, this requires more computational resources to execute and wasn’t possible at my POC scale.

By implementing the above, I believe this POC could be much more valuable and could produce some more substantial results than this initial POC.