COS30018 Intelligent Systems – Project summary report

Ryan Chessum 102564760

Project B – Stock Price Prediction System

Table of contents

Introduction

Fintech101 is a time series prediction model for predicting the stock price of a given stock and utilises machine learning. The system downloads financial data of a stock from yahoo finance to split into training/test data for use on a model. The system has a function that can be used to create different machine learning models for use. The loaded data can be used on the models for making predictions on the future price of the stock. There are also functions to plot data, use models to create multistep predictions and make ensemble predictions. To make the system I used a few libraries such as pandas for the datasets, yahoo finance to get financial data and TensorFlow for the machine learning models. The system is just a prototype so the predictions generated may not be accurate but are a proof of concept for a stock price prediction system.

Overall System Architecture

Loading data

The system can load data for use in training a model to make predictions. The parameters of the function allow for the stock ticker, start, and end date and price type (feature column) to be specified.

```
def load_data(ticker, start_date, end_date, saving=True, n_steps=60, scale=True, shuffle=True, lookup_step=1, split_by_date=True,
              test_size=0.2, feature_columns=['adjclose', 'close', 'volume', 'open', 'high', 'low'], t='adjclose'):
```

The function downloads the data needed from yahoo finance and processes it for use. It will create a pandas data frame containing the data. Then it will scale the data so it can be used on a model. Finally, it will split the data into testing and training data sets.

Create model

The next function in the program is a function for making different models.

```
def create_model(sequence_length, n_features, units=256, cell=LSTM, n_layers=2, dropout=0.3,
                 loss="mean_absolute_error", optimizer="rmsprop", bidirectional=False):

    #Variables:
    ###################
    # sequence_length: no of days for lookback
    # n_features:no of feature columns
    # units:number of nodes in each layer
    # cell: cell type
    # n_layers: number of layers
    # dropout: drop out frequency
    # loss: loss metric
    # optimizer: what optimizer we want to use
    # bidirectional: bidirectional y/n
    ####################################

    #create a sequential model
    model = Sequential()

    #Create amount of layers equal to n_layers
    for i in range(n_layers):
        if i == 0:
            # first layer
            if bidirectional:
                #set the input layer if first layer
                model.add(Bidirectional(cell(units, return_sequences=True), batch_input_shape=(None, sequence_length, n_features)))
            else:
                #if not bidirectional
                model.add(cell(units, return_sequences=True, batch_input_shape=(None, sequence_length, n_features)))
        elif i == n_layers - 1:
            # last layer
            if bidirectional:
                #no return sequences on last layer
                model.add(Bidirectional(cell(units, return_sequences=False)))
            else:
                model.add(cell(units, return_sequences=False))
        else:
            if bidirectional: #if bidirectional
                model.add(Bidirectional(cell(units, return_sequences=True)))
            else:#if not bidirectional
                model.add(cell(units, return_sequences=True))
        # add dropout after each layer
        model.add(Dropout(dropout))
    #final layer, specify linear activation function
    model.add(Dense(1, activation="linear"))
    #compire the model
    model.compile(loss=loss, metrics=["mean_absolute_error"], optimizer=optimizer)
    return model
```

The parameters of the function allow for the cell type, number of layers and number of nodes to be specified. This function makes experimenting with different types of neural networks very quick and easy to do.

Multistep predictions

The system also has a function for making multistep predictions. This means that predictions can be made multiple days into the future. The function uses 6 different models to predict the adjclose, close, volume, open, high and low for the next day. The predictions are then added to the last sequence of financial data as if they were a real day. Then another prediction is made for the next day, using the predictions made. The function will make a prediction for the number of days specified. The more days into the future predicted, the more inaccurate the predictions will become.

```
def multistep_prediction(adjclose, close, volume, op, high, low, data, n_steps, k = 1, scale=True, target='adjclose'):
```

Ensemble predictions

Finally, ensemble predictions can also be made using the ensemble function. Because different models have different biases, some models may tend to predict too high or low. So, the average of many kinds of models may be more accurate. The function simply takes a list of predictions and calculates the average.

```
def ensemble_prediction(predictions):
    #calculate the average of all predictions
    result = 0

    #add predictions
    for p in predictions:
        result += p

    #divide by number of predictions to find average
    result = result / len(predictions)

    return result
```

Implemented data processing techniques

Downloading Data

The program uses the yahoo finance library to download stock price data from the yahoo finance website.

```
#check the data type of ticker
# if ticker is a string use yahoo to get the stock data between the start and end date
if(isinstance(ticker, str)):
    data_frame = si.get_data(ticker, start_date, end_date)
elif(isinstance(ticker, pd.DataFrame)): #else if data passed in is already a pandas dataframe data type, s
    data_frame = ticker
else:
    raise TypeError("ticker can be either a str or a `pd.DataFrame` instances")
```

Once the data is downloaded it is ready to be processed.

Scaling data

The first bit of processing that happens to the data is that it is scaled down to a value between 0 and 1. This is so that I can be used on a machine learning model as the input data for the models needs to be within 0 and 1.

```
#if scale variable is true, scale down data for use (like in v0.01)
if scale:
    #dictionary for storing scalers
    column_scaler = {}
    # scale the data (prices) from 0 to 1
    #loop to cycle through the columns
    for column in feature_columns:
        scaler = preprocessing.MinMaxScaler() #Assign an new scaler
        #scale the data in the dataframe
        data_frame[column] = scaler.fit_transform(np.expand_dims(data_frame[column].values, axis=1))
        #store the scaler in the ditionary
        column_scaler[column] = scaler

    # add the MinMaxScaler instances to the result returned
    results["column_scaler"] = column_scaler
```

The program uses the sklearn library's pre-processing minmaxscaler to do this. The scaler is used on the data and then saved so it can be used to invert the output data back to the proper scale later.

Dropping NaNs

The next bit of processing that happens to the data is that NaNS are dropped. NaNs are blank spaces in our data which could cause problems for our machine learning models. So it is best to remove them.

```
# drop NaNs
data_frame.dropna(inplace=True)
#inplace true means that the the dropping is done on the same object instead of making and returning a new dataframe with the values dropped
```

Fortunately, the pandas dataframe has a function which lets us drop NaNs in just one line of code.

Splitting the data

The final bit of processing that happens to the data is that it is split into different sets for training and testing the model. The last sequence of days is also found.

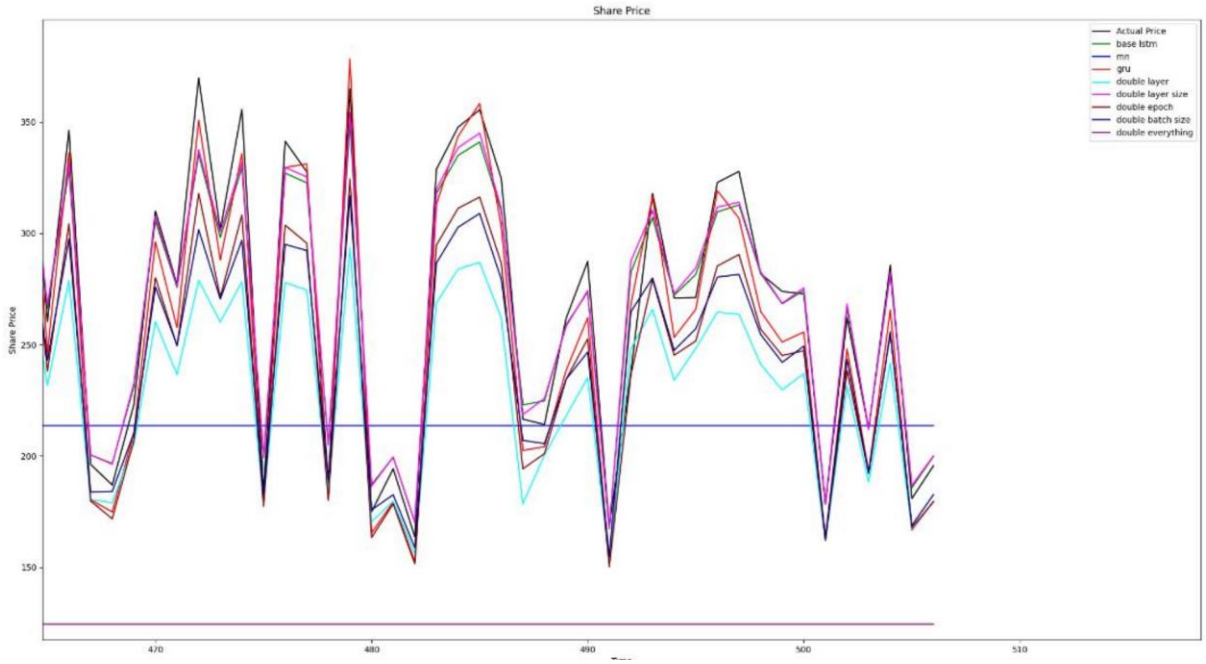
```
sequence_data = []
sequences = deque(maxlen=n_steps)

for entry, target in zip(data_frame[feature_columns + ["date"]].values, data_frame['future'].values):
    sequences.append(entry)
    if len(sequences) == n_steps:
        sequence_data.append([np.array(sequences), target])

# get the last sequence by appending the last `n_step` sequence with `lookup_step` sequence
# for instance, if n_steps=50 and lookup_step=10, last_sequence should be of 60 (that is 50+10) length
# this last_sequence will be used to predict future stock prices that are not available in the dataset
last_sequence = list([s[:len(feature_columns)] for s in sequences]) + list(last_sequence)
last_sequence = np.array(last_sequence).astype(np.float32)
# add to result
results['last_sequence'] = last_sequence

# construct the X's and y's
X, y = [], []
for seq, target in sequence_data:
    X.append(seq)
    y.append(target)

# convert to numpy arrays
X = np.array(X) #last 60 days
y = np.array(y) #future day

#split up the data into train and test data based on the parameters
if split_by_date:
    # split the dataset into training & testing sets by date (not randomly splitting)
    train_samples = int((1 - test_size) * len(X)) #int value for tranng the data based on the test size and length of X
    results["X_train"] = X[:train_samples] # training data
    results["y_train"] = y[:train_samples] # prediction training data
    results["X_test"]  = X[train_samples:] # testing data
    results["y_test"]  = y[train_samples:] # prediction testing data
    if shuffle:
        # shuffle the datasets for training (if shuffle parameter is set)
        shuffle_in_unison(results["X_train"], results["y_train"])
        shuffle_in_unison(results["X_test"], results["y_test"])
else:
    # split the dataset randomly
    results["X_train"], results["X_test"], results["y_train"], results["y_test"] = train_test_split(X, y,
                                                                                                     test_size=test_size, shuffle=shuffle)

# get the list of test set dates
dates = results["X_test"][:, -1, -1]
# retrieve test features from the original dataframe
results["test_df"] = results["data_frame"].loc[dates]
# remove duplicated dates in the testing dataframe
results["test_df"] = results["test_df"][~results["test_df"].index.duplicated(keep='first')]
# remove dates from the training/testing sets & convert to float32
results["X_train"] = results["X_train"][:, :, :len(feature_columns)].astype(np.float32)
results["X_test"] = results["X_test"][:, :, :len(feature_columns)].astype(np.float32)
```

The data is divided up using slice notation. The size of the samples is based on the specified training size and amount of data collected. The X train and test values are the data of previous training days while y train and test are the days to be predicted. The data may be split randomly or by date.

Experimented Machine Learning Techniques

Different models

Thanks to the model creation function I was easily able to experiment with different types of models. Some of the types of models I used were LSTM, GRU simple RNN and ARIMA.



I tried using multiple different models and model sizes. I found that more layers tended to make the model predict lower but increasing the size of the layers brought the models predictions closer to the actual price. I was also surprised at how well the GRU model did. It seemed to get very close to the actual price most of the times compared to the other models.

I also made a function to create ARIMA models. ARIMA models are a simpler machine learning algorithm which bases its prediction off of the rolling average.

```
def create_arima_model(data, target="close"):

    #get training data
    training_data = data["data_frame"][target].values
    #get raw values
    td = [x for x in training_data]

    #create arima model
    model = ARIMA(td, order=(1, 1, 0))
    #fit the model
    fitted_model = model.fit()

    #return the model
    return fitted_model
```

Multistep prediction

Another technique I experimented with was multistep prediction. This means that instead of predicting the stock price for 1 day into the future it would predict the stock prices for the next k days in the future.

In my research I found that there were 2 methods for doing this. The first was to add the predicted price to the last sequence of days as if it was a real day, then use that new set of days to make a prediction. The other method is to create a new data set and make the output (y_train and y_test) a set of days instead of a single day. The method I implemented was the first.

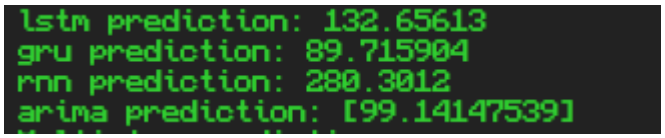
Ensemble prediction

The other method I experimented with was ensemble predictions. Looking at the experiment data from before you can see that different models have different biases. Some predict lower while others predict higher than the actual price. So, it can be reasoned that the average would end up being closer to the actual stock price. For this I normally use a LSTM, GRU, Simple RNN and ARIMA model to make the prediction with this method.

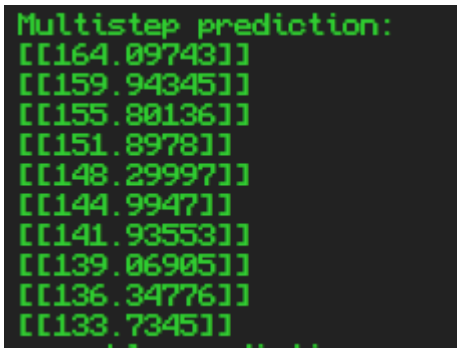
Example of how the system works

If someone was interested in purchasing or selling some META stock, they may want to look at a prediction to double check it was a good idea. (don't do this the model is very inaccurate) they would be able to run the program a get a few predictions on the price of meta.

First the program would generate some predictions for the next day with several different models.



Next it would give a multistep prediction for the next 10 days.



Finally, the program would generate an ensemble prediction.

Critical analysis of the implementation

Can’t save models

One of the current flaws in the system is there is no way to save or load models. So, every time the program is used, the models need to be re-trained. The functionality of saving models would mean that the models would be able to train over multiple sessions which would greatly increase their accuracy as well as let the same models be tested every day when used to forecast stock prices.

Multistep prediction implementation

The current method for multistep prediction is very inefficient. As stated, before the current method uses predictions as if they were real data to predict multiple days into the future. The more days into the future predicted with this method, the more inaccurate the predictions become. This method also requires 6 different models for the close, adjclose, open, high, low and volume. The other method would only require 1 model and would be more accurate.

Only uses numerical data to predict stock prices

Another flaw with the system is that the only data used by the system to make predictions is numerical data. The models used in the system only notice trends in the data make predictions. This information is useful for predicting stock prices but someone who was trying to predict the price of the stock would also consider things like historical elements of the company, what the companies plans are, public perception of the company etc. To make the system more effective, other data such as social media posts analysed with a sentiment analysis model could be used or the company’s quarterly earnings then combined in the ensemble prediction.

Summary

Though there are aspects of the project that could be improved, the system itself is a good proof of concept for time a time series forecasting. It can make simple predictions based on previous stock data. If I were to improve the system in the future, I would make the system take into account more data factors such as the company's performance and public perception. As well as train the models for much longer to improve its accuracy.