Aritifical Intelligence Assignment 2

SAHIL PUNDORA

216092223

INTRODUCTION

Could we possibly predict future movements of, say, AAPL Stocks, by analyzing the last 10 days of closing price and volume of the stock? To check, if there exists some, at least better than random, relationship here that a recurrent neural network could discover.

In [1]:
```
# Read the dataset
import pandas as pd
df = pd.read_csv('/home/sahil94/Desktop/LSTM/AAPL.csv', names=['date', 'low', 'high', 'open', 'close', 'volume'])

#Converting Date Format to Julian Date
df['date_'] =  pd.to_datetime(df['date'])
df['DATE'] = df['date_'].dt.strftime('%y%j')

#We will just be using the closing price and volume to predict
df.set_index("DATE", inplace=True)
main_df = df[['close', 'volume']]

# if there are gaps in data, use previously known values
main_df.fillna(method="ffill", inplace=True)
main_df.dropna(inplace=True)
print(main_df.head())
```

```
          close       volume
DATE
80347  0.513393  117258400.0
80350  0.486607   43971200.0
80351  0.450893   26432000.0
80352  0.462054   21610400.0
80353  0.475446   18362400.0

/anaconda/envs/py36/lib/python3.6/site-packages/pandas/core/frame.py:3790: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy)
  downcast=downcast, **kwargs)
/anaconda/envs/py36/lib/python3.6/site-packages/ipykernel/__main__.py:15: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy)
```

TARGET

If price goes up in 2 days, then it's a buy. If it goes down in 2 days, not buy/sell.

In [2]:
```
SEQ_LEN = 10  # how long of a preceeding sequence to collect for RNN
FUTURE_PERIOD_PREDICT = 2  # how far into the future are we trying to predict?
RATIO_TO_PREDICT = "AAPL_Stocks"
```

In [3]:
```
# This function will take values from 2 columns.
# If the "future" column is higher, great, it's a 1 (buy). Otherwise it's a 0 (sell).
# To do this, first, we need a future column!

def classify(current, future):
    if float(future) > float(current):
        return 1
    else:
        return 0
```

In [4]:
```
# .shift will just shift the columns for us, a negative shift will shift them "up."
# So shifting up 2 will give us the price 2 days in the future, and we're just assigning this to a new column.
# Now that we've got the future values, we can use them to make a target using the function we made above.

main_df['future'] = main_df['close'].shift(-FUTURE_PERIOD_PREDICT)
main_df['target'] = list(map(classify, main_df['close'], main_df['future']))
print(main_df.head(20))
```

```
          close       volume    future  target
DATE
80347  0.513393  117258400.0  0.450893       0
80350  0.486607   43971200.0  0.462054       0
80351  0.450893   26432000.0  0.475446       1
80352  0.462054   21610400.0  0.504464       1
80353  0.475446   18362400.0  0.529018       1
80354  0.504464   12157600.0  0.551339       1
80357  0.529018    9340800.0  0.580357       1
80358  0.551339   11737600.0  0.633929       1
80359  0.580357   12000800.0  0.642857       1
80361  0.633929   13893600.0  0.627232       0
80364  0.642857   23290400.0  0.609375       0
80365  0.627232   17220000.0  0.616071       0
80366  0.609375    8937600.0  0.602679       0
81002  0.616071    5415200.0  0.575893       0
81005  0.602679    8932000.0  0.551339       0
81006  0.575893   11289600.0  0.540179       0
81007  0.551339   13921600.0  0.569196       1
81008  0.540179    9956800.0  0.564732       1
81009  0.569196    5376000.0  0.544643       0
81012  0.564732    5924800.0  0.546875       0

/anaconda/envs/py36/lib/python3.6/site-packages/ipykernel/__main__.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy)
/anaconda/envs/py36/lib/python3.6/site-packages/ipykernel/__main__.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy (http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy)
```

In [5]:
```
#Since our data is sequential, we want to slice our validation while it's still in order.
#So I'd take the last 5% of the data as validation
times = sorted(main_df.index.values)  # get the days
last_5pct = sorted(main_df.index.values)[-int(0.05*len(times))]  # get the last 5% of the days
# make the validation data where the index is in the last 5%
validation_main_df = main_df[(main_df.index >= last_5pct)]
main_df = main_df[(main_df.index < last_5pct)]  # now the main_df is all the data up to the last 5%
```

In [6]:
```
from sklearn import preprocessing
from collections import deque
import numpy as np
import random


#Normalizing and Balancing the data
def preprocess_df(df):
    # don't need this anymore.
    df = df.drop("future", 1)

    for col in df.columns:  # go through all of the columns
        if col != "target":  # normalize all ... except for the target itself!
            df[col] = df[col].pct_change()  # pct change "normalizes" the different currencies
            df.dropna(inplace=True)  # remove the nas created by pct_change
            df[col] = preprocessing.scale(df[col].values)  # scale between 0 and 1.

    df.dropna(inplace=True)


    sequential_data = []  # this is a list that will CONTAIN the sequences
    prev_days = deque(maxlen=SEQ_LEN)  # These will be our actual sequences.
    #They are made with deque, which keeps the maximum length by popping out older values as new ones come in

    for i in df.values:  # iterate over the values
        prev_days.append([n for n in i[:-1]])  # store all but the target
        if len(prev_days) == SEQ_LEN:  # make sure we have 60 sequences!
            sequential_data.append([np.array(prev_days), i[-1]])

    random.shuffle(sequential_data)  # shuffle for good measure.

    buys = []  # list that will store our buy sequences and targets
    sells = []  # list that will store our sell sequences and targets

    for seq, target in sequential_data:  # iterate over the sequential data
        if target == 0:  # if it's a "not buy"
            sells.append([seq, target])  # append to sells list
        elif target == 1:  # otherwise if the target is a 1...
            buys.append([seq, target])  # it's a buy!

    random.shuffle(buys)  # shuffle the buys
    random.shuffle(sells)  # shuffle the sells!
```

```
    lower = min(len(buys), len(sells))

    buys = buys[:lower]
    sells = sells[:lower]

    sequential_data = buys+sells  # add them together
    random.shuffle(sequential_data)  # another shuffle

    X = []
    y = []

    for seq, target in sequential_data:  # going over our new sequential data
        X.append(seq)  # X is the sequences
        y.append(target)  # y is the targets/labels (buys vs sell/notbuy)

    return np.array(X), y  # return X and y...and make X a numpy array!
```

In [7]:

```
train_x, train_y = preprocess_df(main_df)
validation_x, validation_y = preprocess_df(validation_main_df)

print(f"Train data: {len(train_x)} validation: {len(validation_x)}")
print(f"Dont buys: {train_y.count(0)}, Buys: {train_y.count(1)}")
print(f"VALIDATION Dont buys: {validation_y.count(0)}, Buys: {validation_y.count(1)}")
```

```
Train data: 9086 validation: 420
Dont buys: 4543, Buys: 4543
VALIDATION Dont buys: 210, Buys: 210
```

In [20]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM, BatchNormalization
from keras.backend import clear_session

# SIMPLE LSTM MODEL
clear_session()
model = Sequential()
model.add(LSTM(128, input_shape=(train_x.shape[1:])))
model.add(Dropout(0.2))
#normalizes activation outputs
#same reason you want to normalize your input data.
model.add(BatchNormalization())

model.add(Dense(32, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(2, activation='softmax'))

opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)

# Compile model
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=opt,
    metrics=['accuracy']
)

# Train model
model.fit(train_x, train_y, batch_size=64, epochs=20, validation_data=(validation_x, validation_y))

# Score model
score = model.evaluate(validation_x, validation_y, verbose=3)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 9086 samples, validate on 420 samples
Epoch 1/20
9086/9086 [==============================] - 3s 314us/step - loss: 0.6757 - acc: 0.5571 - val_loss: 1.1148 - val_acc: 0.5214
Epoch 2/20
9086/9086 [==============================] - 2s 273us/step - loss: 0.6755 - acc: 0.5575 - val_loss: 1.2782 - val_acc: 0.5167
Epoch 3/20
9086/9086 [==============================] - 2s 212us/step - loss: 0.6742 - acc: 0.5541 - val_loss: 1.0888 - val_acc: 0.5357
```

```
Epoch 4/20
9086/9086 [==============================] - 2s 221us/step - loss: 0.6750 - acc: 0.5524 - val_loss: 1.1597 - val_acc: 0.5190
Epoch 5/20
9086/9086 [==============================] - 3s 291us/step - loss: 0.6736 - acc: 0.5604 - val_loss: 1.3226 - val_acc: 0.5262
Epoch 6/20
9086/9086 [==============================] - 2s 247us/step - loss: 0.6726 - acc: 0.5621 - val_loss: 1.4503 - val_acc: 0.5357
Epoch 7/20
9086/9086 [==============================] - 2s 219us/step - loss: 0.6704 - acc: 0.5614 - val_loss: 1.2853 - val_acc: 0.5333
Epoch 8/20
9086/9086 [==============================] - 2s 258us/step - loss: 0.6726 - acc: 0.5614 - val_loss: 1.1707 - val_acc: 0.5143
Epoch 9/20
9086/9086 [==============================] - 2s 223us/step - loss: 0.6693 - acc: 0.5700 - val_loss: 1.5069 - val_acc: 0.5405
Epoch 10/20
9086/9086 [==============================] - 2s 214us/step - loss: 0.6675 - acc: 0.5719 - val_loss: 1.4370 - val_acc: 0.5071
Epoch 11/20
9086/9086 [==============================] - 2s 238us/step - loss: 0.6678 - acc: 0.5666 - val_loss: 1.3180 - val_acc: 0.5405
Epoch 12/20
9086/9086 [==============================] - 2s 241us/step - loss: 0.6696 - acc: 0.5654 - val_loss: 1.6552 - val_acc: 0.5429
Epoch 13/20
9086/9086 [==============================] - 2s 218us/step - loss: 0.6678 - acc: 0.5719 - val_loss: 1.3911 - val_acc: 0.5405
Epoch 14/20
9086/9086 [==============================] - 2s 245us/step - loss: 0.6670 - acc: 0.5698 - val_loss: 1.5641 - val_acc: 0.5500
Epoch 15/20
9086/9086 [==============================] - 2s 266us/step - loss: 0.6635 - acc: 0.5768 - val_loss: 1.6601 - val_acc: 0.5238
Epoch 16/20
9086/9086 [==============================] - 2s 225us/step - loss: 0.6623 - acc: 0.5800 - val_loss: 1.5610 - val_acc: 0.5357
Epoch 17/20
9086/9086 [==============================] - 2s 220us/step - loss: 0.6632 - acc: 0.5725 - val_loss: 1.4525 - val_acc: 0.5310
Epoch 18/20
9086/9086 [==============================] - 2s 271us/step - loss: 0.6616 - acc: 0.5756 - val_loss: 1.4520 - val_acc: 0.5500
Epoch 19/20
9086/9086 [==============================] - 2s 221us/step - loss: 0.6615 - acc: 0.5744 - val_loss: 1.5113 - val_acc: 0.5381
Epoch 20/20
9086/9086 [==============================] - 2s 224us/step - loss: 0.6607 - acc: 0.5813 - val_loss: 1.4311 - val_acc: 0.5524
Test loss: 1.431086052031744
Test accuracy: 0.5523809523809524
```

In [21]:
```
# LSTM STACKED MODEL
clear_session()
model = Sequential()
model.add(LSTM(128, input_shape=(train_x.shape[1:]), return_sequences=True))
model.add(Dropout(0.2))
model.add(BatchNormalization())

model.add(LSTM(128, return_sequences=True))
model.add(Dropout(0.1))
model.add(BatchNormalization())

model.add(LSTM(128))
model.add(Dropout(0.2))
model.add(BatchNormalization())

model.add(Dense(32, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(2, activation='softmax'))

opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)

# Compile model
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=opt,
    metrics=['accuracy']
)

# Train model
model.fit(train_x, train_y, batch_size=64, epochs=20, validation_data=(validation_x, validation_y))

# Score model
score = model.evaluate(validation_x, validation_y, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 9086 samples, validate on 420 samples
Epoch 1/20
9086/9086 [==============================] - 3s 342us/step - loss: 0.6606 - acc: 0.5754 - val_loss: 1.7524 - val_acc: 0.5595
Epoch 2/20
9086/9086 [==============================] - 2s 228us/step - loss: 0.6580 - acc: 0.5799 - val_loss: 1.5661 - val_acc: 0.5429
Epoch 3/20
```

```
9086/9086 [==============================] - 2s 208us/step - loss: 0.6586 - acc: 0.5862 - val_loss: 1.5842 - val_acc: 0.5405
Epoch 4/20
9086/9086 [==============================] - 2s 217us/step - loss: 0.6559 - acc: 0.5819 - val_loss: 1.9530 - val_acc: 0.5333
Epoch 5/20
9086/9086 [==============================] - 2s 275us/step - loss: 0.6551 - acc: 0.5786 - val_loss: 1.6143 - val_acc: 0.5524
Epoch 6/20
9086/9086 [==============================] - 2s 209us/step - loss: 0.6521 - acc: 0.5869 - val_loss: 1.8023 - val_acc: 0.5452
Epoch 7/20
9086/9086 [==============================] - 2s 211us/step - loss: 0.6519 - acc: 0.5845 - val_loss: 1.9247 - val_acc: 0.5643
Epoch 8/20
9086/9086 [==============================] - 3s 291us/step - loss: 0.6519 - acc: 0.5899 - val_loss: 1.8847 - val_acc: 0.5405
Epoch 9/20
9086/9086 [==============================] - 2s 226us/step - loss: 0.6498 - acc: 0.5835 - val_loss: 2.2825 - val_acc: 0.5524
Epoch 10/20
9086/9086 [==============================] - 2s 219us/step - loss: 0.6471 - acc: 0.5938 - val_loss: 2.0779 - val_acc: 0.5524
Epoch 11/20
9086/9086 [==============================] - 3s 278us/step - loss: 0.6460 - acc: 0.5943 - val_loss: 2.1986 - val_acc: 0.5357
Epoch 12/20
9086/9086 [==============================] - 2s 223us/step - loss: 0.6479 - acc: 0.5950 - val_loss: 2.0863 - val_acc: 0.5619
Epoch 13/20
9086/9086 [==============================] - 2s 225us/step - loss: 0.6414 - acc: 0.6037 - val_loss: 2.5094 - val_acc: 0.5524
Epoch 14/20
9086/9086 [==============================] - 3s 275us/step - loss: 0.6399 - acc: 0.5989 - val_loss: 2.1243 - val_acc: 0.5238
Epoch 15/20
9086/9086 [==============================] - 2s 221us/step - loss: 0.6406 - acc: 0.6002 - val_loss: 2.3122 - val_acc: 0.5643
Epoch 16/20
9086/9086 [==============================] - 2s 214us/step - loss: 0.6378 - acc: 0.6032 - val_loss: 2.1432 - val_acc: 0.5500
Epoch 17/20
9086/9086 [==============================] - 2s 239us/step - loss: 0.6383 - acc: 0.6053 - val_loss: 2.2681 - val_acc: 0.5500
Epoch 18/20
9086/9086 [==============================] - 2s 252us/step - loss: 0.6344 - acc: 0.6021 - val_loss: 2.4338 - val_acc: 0.5500
Epoch 19/20
9086/9086 [==============================] - 2s 217us/step - loss: 0.6321 - acc: 0.6063 - val_loss: 2.3302 - val_acc: 0.5381
Epoch 20/20
9086/9086 [==============================] - 2s 235us/step - loss: 0.6297 - acc: 0.6005 - val_loss: 2.5938 - val_acc: 0.5381
Test loss: 2.593750592995258
Test accuracy: 0.5380952380952381
```

Conclusion

We could see that both Simple and Stacked model perform a little bit better than chance with 55.23% and 53.8% respectively. Simple LSTM in this case performs better than stacked.

Both the models are insufficient for practical use, and are only for educational purposes.

This is mainly because-

Finance, particulary stocks, are terrible to predict.

Historical results are not indicative of future results.

There are so many external (latent) factors affecting the stock prices that cannot be taken into account.

References- Sentdex -cryptocurrency-predicting RNN Model - Deep Learning w/ Python, TensorFlow and Keras https://www.youtube.com/watch?v=yWkpRdpOiPY (https://www.youtube.com/watch?v=yWkpRdpOiPY)

THE END

In [ ]:

```

```