BiLSTM Model for predicting trend

In [ ]:
```python
#imports
import numpy as np
from math import sqrt
from numpy import concatenate

import sklearn
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error
from sklearn.utils import shuffle
import joblib

from matplotlib import pyplot
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure

import pandas as pd
from pandas import read_csv
from pandas import DataFrame
from pandas import concat


import keras.utils
from keras.models import Sequential
from keras.layers import Dropout
from keras.layers import Activation
from keras.layers import Bidirectional
from keras.layers import Dense
from keras.layers import LSTM
from keras.optimizers import Adam

import tensorflow as tf
import seaborn as sn
import seed
import os
tf.get_logger().setLevel('ERROR')
```

In [ ]:
```python
"""
method to create lagged features

data - data
to_keep - number of lagged_features
to_remove - number of days to remove

"""
def create_lagged_features(data, to_keep=1, to_remove=1):
    variables = 1 if type(data) is list else data.shape[1]
    df = DataFrame(data)
    columns, names = list(), list()

    for i in range(to_keep, 0, -1):
        columns.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(variables)]

    for i in range(0, to_remove):
        columns.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(variables)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(variables)]

    #put it all together
    final = concat(columns, axis=1)
    final.columns = names

    #drop rows with NaN values
    final.dropna(inplace=True)

    return final
```

In [ ]:
```python
"""
function to calculate rsi

data - data
period - RSI period

"""
def rsi(data, period: int = 14):

    delta = data["Close"].diff()

    up, down = delta.copy(), delta.copy()
    up[up < 0] = 0
    down[down > 0] = 0

    gain = up.ewm(com=(period - 1), min_periods=period).mean()
    loss = down.abs().ewm(com=(period - 1), min_periods=period).mean()

    RS = gain / loss
    return 100 - (100 / (1 + RS))
```

In [ ]:
```python
#lag granularity - days or hours
lag_granularity = "days"
#lag value
lag = 1
# type of analyser - TextBlob or vader
analyser = "vader"
# analyser = "TextBlob"
#dataset grouped type - day or hour
dataset_grouped_by = "day"
```

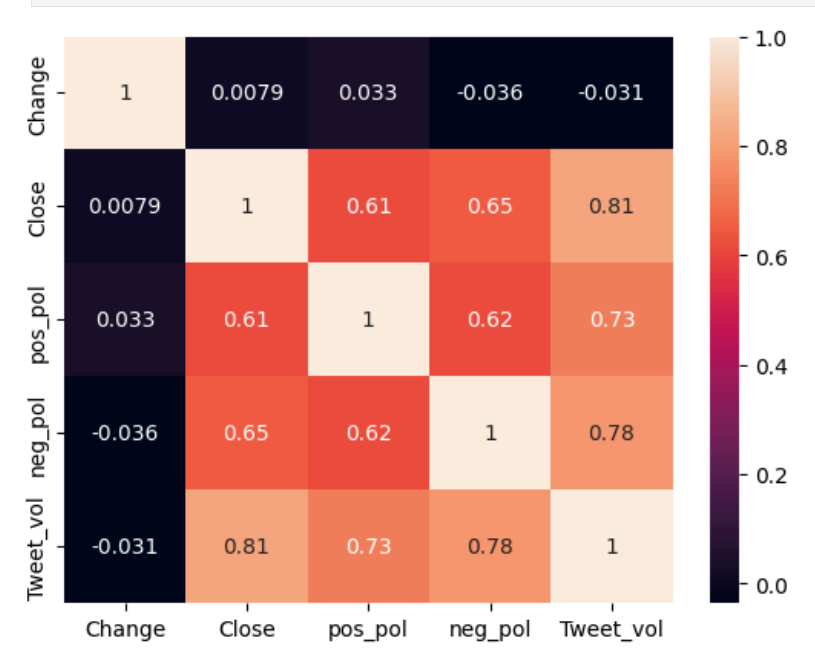
In [ ]:
```python
#read dataset
folder = "./../../datasets/tweets_prices_volumes_sentiment/"+analyser+"/"+dataset_grouped_by+"_datasets/cleaned"
filename = folder+"/final_data_lag_"+lag_granularity+"_"+str(lag)+".csv" if (lag > 0) else folder+"/final_data_no_lag.csv"
df = pd.read_csv(filename)
```

In [ ]:
```python
# Нерабочий исходный код
#group by datetime
# df = df.groupby('DateTime').agg(lambda x: x.mean())

# Convert the 'DateTime' column to datetime format
df['DateTime'] = pd.to_datetime(df['DateTime'])

# Exclude non-numeric columns ('DateTime' and 'dt') from the aggregation
numeric_columns = df.select_dtypes(include=['float64', 'int64']).columns.tolist()

# Perform the aggregation on numeric columns only, grouping by 'DateTime'
df_grouped = df.groupby('DateTime')[numeric_columns].mean()

# Reset the index if you want 'DateTime' back as a column
df_grouped.reset_index(inplace=True)

# Now df_grouped contains the mean of numeric columns, grouped by 'DateTime'
```

In [ ]:
```python
#get change label
df["Change"] = (df["Close"] > df["Close"].shift(1)).astype(int)

add_RSI = False
add_longMAvg = False
add_shortMAvg = False

if(add_RSI):
    #calcualte RSI
    RSI = 14
    df['RSI'] = rsi(df, RSI)
    df = df.iloc[RSI:]

#calculate moving averages
if(add_shortMAvg):
    short_window = 9
    df['short_mavg'] = df.rolling(window=short_window)["Close"].mean()

if(add_longMAvg):
    long_window = 21
    df["long_mavg"] = df.rolling(window=long_window)["Close"].mean()

if(add_longMAvg):
    df = df.iloc[long_window:]
elif(add_RSI):
    df = df.iloc[RSI:]
elif(add_shortMAvg):
    df = df.iloc[short_window:]
```

In [ ]:
```python
#keep only wanted columns
features = ['Change', 'subjectivity', 'polarity','Tweet_vol','Volume_(BTC)'] if analyser == "Textblob" else ['Change', 'Close', 'pos_pol', 'neg_pol', 'Tweet_vol']

if(add_RSI):
    features.append("RSI")

if(add_longMAvg):
    features.append("long_mavg")

if(add_shortMAvg):
    features.append("short_mavg")

df = df[features]
```

In [ ]:
```python
#plot correlation matrix
sn.heatmap(df.corr(), annot=True)
plt.show()
```



In [ ]:
```python
#creating copy so that data is not loaded once again
df_copy = df.copy()
```

In [ ]:
```python
#number of previous records to consider for every example
n_lag = 7
#number of features
n_features = len(features)
#calculate total_features
total_features = n_lag*n_features

if(total_features == 0):
    total_features = n_features
```

In [ ]:
```python
#add lagged data to records
data_with_lagged = create_lagged_features(df_copy, n_lag, 1)
# Check the column names of the DataFrame after creating lagged features

data_with_lagged = data_with_lagged.reset_index()

# Нерабочий исходный код
# data_with_lagged = data_with_lagged.drop(['DateTime'], axis=1)

data_with_lagged = data_with_lagged.drop(['index'], axis=1)
```

In [ ]:
```python
#shuffle data
np.random.seed(1)
#shuffle times
shuffle_times = 5;
for j in range(0, shuffle_times+1):
    data_with_lagged = shuffle(data_with_lagged)
```

In [ ]:
```python
#divide df into train and test
train_ratio = 0.85
data_len = len(data_with_lagged)
train_size = int(data_len*train_ratio)

train = data_with_lagged.iloc[:train_size]
test = data_with_lagged.iloc[train_size:]
```

In [ ]:
```python
#prepare labels
train_y = train["var1(t)"].values
test_y = test["var1(t)"].values

train_y = train_y.reshape(len(train_y), 1)
test_y = test_y.reshape(len(test_y), 1)
```

In [ ]:
```python
#normalise features
xscaler = MinMaxScaler(feature_range=(0, 1))
train = xscaler.fit_transform(train)
test = xscaler.transform(test)
joblib.dump(xscaler, 'saved/scaler.pkl')
```

Out[ ]:
```
['saved/scaler.pkl']
```

In [ ]:
```python
#prepare data
train_labels = train_y
test_labels = test_y
```

In [ ]:
```python
#remove the last set of values(data of time to be predicted)
train = train[:, :total_features]
test = test[:, :total_features]
```

In [ ]:
```python
#keep only prices array
train_X, train_y = train[:, :total_features], train_y[:, -1]
test_X, test_y = test[:, :total_features], test_y[:, -1]
```

In [ ]:
```python
# reshape input to be 3D [samples, timesteps, features]
train_X = train_X.reshape((train_X.shape[0], n_lag, n_features))
test_X = test_X.reshape((test_X.shape[0], n_lag, n_features))
```

In [ ]:
```python
#set labels for training data to categorical
train_y = keras.utils.to_categorical(train_y, 2)
```

In [ ]:
```python
np.random.seed(1)
tf.random.set_seed(1)

# design network
model = Sequential()
neurons = 64
epochs = 10000
dropout = 0.25
batch_size = 80
activ_func = "linear"

model.add(Bidirectional(LSTM(neurons, return_sequences=True, input_shape=(train_X.shape[1], train_X.shape[2]), activation=activ_func)))
model.add(Dropout(dropout))

model.add(Bidirectional(LSTM(neurons, return_sequences=False, activation=activ_func)))
model.add(Dropout(dropout))

model.add(Dense(2, activation="softmax"))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In [ ]:
```python
# early stopping
callback = keras.callbacks.EarlyStopping(monitor='val_loss', patience =50)

# fit network
history = model.fit(train_X, train_y, epochs=10000, batch_size=batch_size, verbose=2, shuffle=True,validation_split=0.2, callbacks=[callback])
```

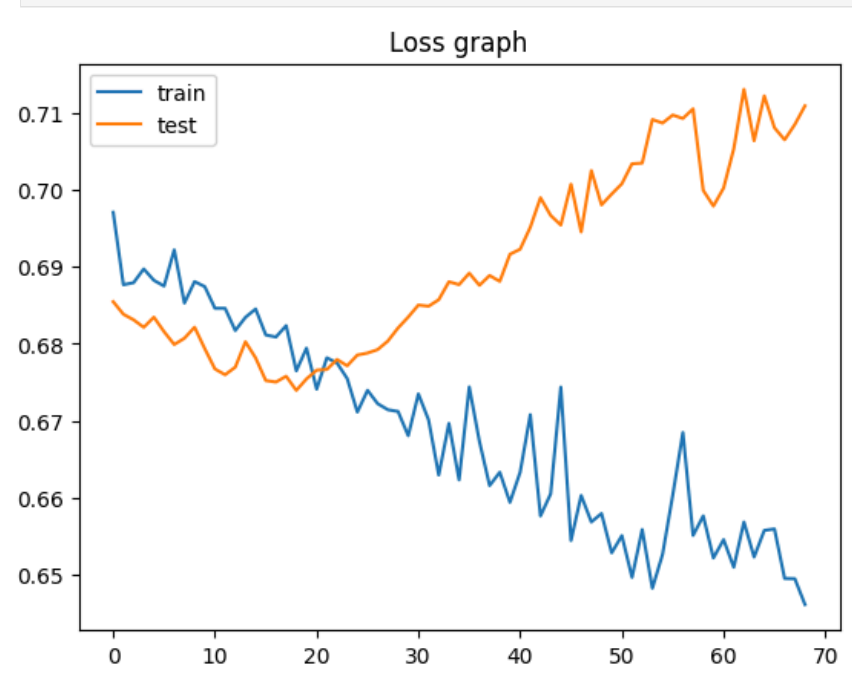
```
Epoch 1/10000
4/4 - 4s - loss: 0.6971 - accuracy: 0.4385 - val_loss: 0.6855 - val_accuracy: 0.5526 - 4s/epoch - 1s/step
Epoch 2/10000
Epoch 2/10000
4/4 - 0s - loss: 0.6877 - accuracy: 0.5449 - val_loss: 0.6838 - val_accuracy: 0.5526 - 54ms/epoch - 14ms/step
Epoch 3/10000
4/4 - 0s - loss: 0.6879 - accuracy: 0.5382 - val_loss: 0.6831 - val_accuracy: 0.5526 - 54ms/epoch - 13ms/step
Epoch 4/10000
4/4 - 0s - loss: 0.6897 - accuracy: 0.5116 - val_loss: 0.6821 - val_accuracy: 0.5526 - 53ms/epoch - 13ms/step
Epoch 5/10000
4/4 - 0s - loss: 0.6882 - accuracy: 0.5349 - val_loss: 0.6835 - val_accuracy: 0.5921 - 53ms/epoch - 13ms/step
Epoch 6/10000
4/4 - 0s - loss: 0.6875 - accuracy: 0.5116 - val_loss: 0.6816 - val_accuracy: 0.5921 - 53ms/epoch - 13ms/step
Epoch 7/10000
4/4 - 0s - loss: 0.6922 - accuracy: 0.4917 - val_loss: 0.6799 - val_accuracy: 0.5526 - 53ms/epoch - 13ms/step
Epoch 8/10000
4/4 - 0s - loss: 0.6853 - accuracy: 0.5449 - val_loss: 0.6807 - val_accuracy: 0.5921 - 52ms/epoch - 13ms/step
Epoch 9/10000
4/4 - 0s - loss: 0.6881 - accuracy: 0.5681 - val_loss: 0.6821 - val_accuracy: 0.6316 - 53ms/epoch - 13ms/step
Epoch 10/10000
4/4 - 0s - loss: 0.6874 - accuracy: 0.5615 - val_loss: 0.6794 - val_accuracy: 0.5658 - 54ms/epoch - 13ms/step
Epoch 11/10000
4/4 - 0s - loss: 0.6846 - accuracy: 0.5548 - val_loss: 0.6767 - val_accuracy: 0.5658 - 53ms/epoch - 13ms/step
Epoch 12/10000
4/4 - 0s - loss: 0.6846 - accuracy: 0.5648 - val_loss: 0.6760 - val_accuracy: 0.5789 - 53ms/epoch - 13ms/step
Epoch 13/10000
4/4 - 0s - loss: 0.6817 - accuracy: 0.5648 - val_loss: 0.6769 - val_accuracy: 0.5658 - 53ms/epoch - 13ms/step
Epoch 14/10000
4/4 - 0s - loss: 0.6834 - accuracy: 0.5349 - val_loss: 0.6803 - val_accuracy: 0.6053 - 54ms/epoch - 14ms/step
Epoch 15/10000
4/4 - 0s - loss: 0.6845 - accuracy: 0.5116 - val_loss: 0.6781 - val_accuracy: 0.5789 - 54ms/epoch - 14ms/step
Epoch 16/10000
4/4 - 0s - loss: 0.6811 - accuracy: 0.5615 - val_loss: 0.6752 - val_accuracy: 0.5658 - 52ms/epoch - 13ms/step
Epoch 17/10000
4/4 - 0s - loss: 0.6809 - accuracy: 0.5548 - val_loss: 0.6750 - val_accuracy: 0.5658 - 52ms/epoch - 13ms/step
Epoch 18/10000
4/4 - 0s - loss: 0.6823 - accuracy: 0.5581 - val_loss: 0.6758 - val_accuracy: 0.5658 - 52ms/epoch - 13ms/step
Epoch 19/10000
4/4 - 0s - loss: 0.6765 - accuracy: 0.5947 - val_loss: 0.6739 - val_accuracy: 0.6053 - 52ms/epoch - 13ms/step
Epoch 20/10000
4/4 - 0s - loss: 0.6794 - accuracy: 0.5648 - val_loss: 0.6754 - val_accuracy: 0.5395 - 53ms/epoch - 13ms/step
Epoch 21/10000
4/4 - 0s - loss: 0.6741 - accuracy: 0.5947 - val_loss: 0.6766 - val_accuracy: 0.5526 - 52ms/epoch - 13ms/step
Epoch 22/10000
4/4 - 0s - loss: 0.6782 - accuracy: 0.5548 - val_loss: 0.6766 - val_accuracy: 0.5658 - 52ms/epoch - 13ms/step
Epoch 23/10000
4/4 - 0s - loss: 0.6775 - accuracy: 0.5748 - val_loss: 0.6779 - val_accuracy: 0.5921 - 53ms/epoch - 13ms/step
Epoch 24/10000
4/4 - 0s - loss: 0.6755 - accuracy: 0.5648 - val_loss: 0.6771 - val_accuracy: 0.5526 - 52ms/epoch - 13ms/step
Epoch 25/10000
4/4 - 0s - loss: 0.6711 - accuracy: 0.5714 - val_loss: 0.6785 - val_accuracy: 0.5526 - 52ms/epoch - 13ms/step
Epoch 26/10000
4/4 - 0s - loss: 0.6739 - accuracy: 0.5980 - val_loss: 0.6788 - val_accuracy: 0.5526 - 53ms/epoch - 13ms/step
Epoch 27/10000
4/4 - 0s - loss: 0.6722 - accuracy: 0.5781 - val_loss: 0.6792 - val_accuracy: 0.5658 - 53ms/epoch - 13ms/step
Epoch 28/10000
4/4 - 0s - loss: 0.6714 - accuracy: 0.5615 - val_loss: 0.6803 - val_accuracy: 0.5658 - 54ms/epoch - 14ms/step
Epoch 29/10000
4/4 - 0s - loss: 0.6712 - accuracy: 0.5914 - val_loss: 0.6820 - val_accuracy: 0.5789 - 55ms/epoch - 14ms/step
Epoch 30/10000
4/4 - 0s - loss: 0.6680 - accuracy: 0.6013 - val_loss: 0.6835 - val_accuracy: 0.5658 - 56ms/epoch - 14ms/step
Epoch 31/10000
4/4 - 0s - loss: 0.6735 - accuracy: 0.5847 - val_loss: 0.6850 - val_accuracy: 0.6053 - 55ms/epoch - 14ms/step
Epoch 32/10000
4/4 - 0s - loss: 0.6701 - accuracy: 0.5615 - val_loss: 0.6849 - val_accuracy: 0.5921 - 56ms/epoch - 14ms/step
Epoch 33/10000
4/4 - 0s - loss: 0.6629 - accuracy: 0.6047 - val_loss: 0.6857 - val_accuracy: 0.5789 - 55ms/epoch - 14ms/step
Epoch 34/10000
4/4 - 0s - loss: 0.6696 - accuracy: 0.5814 - val_loss: 0.6881 - val_accuracy: 0.5921 - 56ms/epoch - 14ms/step
Epoch 35/10000
4/4 - 0s - loss: 0.6623 - accuracy: 0.5947 - val_loss: 0.6877 - val_accuracy: 0.6184 - 55ms/epoch - 14ms/step
Epoch 36/10000
4/4 - 0s - loss: 0.6744 - accuracy: 0.5615 - val_loss: 0.6892 - val_accuracy: 0.5921 - 56ms/epoch - 14ms/step
Epoch 37/10000
4/4 - 0s - loss: 0.6673 - accuracy: 0.6013 - val_loss: 0.6876 - val_accuracy: 0.5921 - 55ms/epoch - 14ms/step
Epoch 38/10000
4/4 - 0s - loss: 0.6615 - accuracy: 0.5914 - val_loss: 0.6889 - val_accuracy: 0.5921 - 53ms/epoch - 13ms/step
Epoch 39/10000
4/4 - 0s - loss: 0.6633 - accuracy: 0.5914 - val_loss: 0.6881 - val_accuracy: 0.5789 - 53ms/epoch - 13ms/step
Epoch 40/10000
4/4 - 0s - loss: 0.6594 - accuracy: 0.6113 - val_loss: 0.6916 - val_accuracy: 0.5921 - 53ms/epoch - 13ms/step
Epoch 41/10000
4/4 - 0s - loss: 0.6633 - accuracy: 0.5947 - val_loss: 0.6923 - val_accuracy: 0.5921 - 53ms/epoch - 13ms/step
Epoch 42/10000
4/4 - 0s - loss: 0.6708 - accuracy: 0.5914 - val_loss: 0.6951 - val_accuracy: 0.6184 - 53ms/epoch - 13ms/step
Epoch 43/10000
4/4 - 0s - loss: 0.6576 - accuracy: 0.6213 - val_loss: 0.6990 - val_accuracy: 0.5658 - 54ms/epoch - 13ms/step
Epoch 44/10000
4/4 - 0s - loss: 0.6605 - accuracy: 0.6113 - val_loss: 0.6967 - val_accuracy: 0.5526 - 55ms/epoch - 14ms/step
Epoch 45/10000
4/4 - 0s - loss: 0.6744 - accuracy: 0.5615 - val_loss: 0.6954 - val_accuracy: 0.5921 - 56ms/epoch - 14ms/step
Epoch 46/10000
4/4 - 0s - loss: 0.6544 - accuracy: 0.5714 - val_loss: 0.7007 - val_accuracy: 0.5263 - 54ms/epoch - 13ms/step
Epoch 47/10000
4/4 - 0s - loss: 0.6603 - accuracy: 0.6047 - val_loss: 0.6945 - val_accuracy: 0.5526 - 53ms/epoch - 13ms/step
Epoch 48/10000
4/4 - 0s - loss: 0.6568 - accuracy: 0.6146 - val_loss: 0.7025 - val_accuracy: 0.5789 - 54ms/epoch - 14ms/step
Epoch 49/10000
4/4 - 0s - loss: 0.6579 - accuracy: 0.5648 - val_loss: 0.6980 - val_accuracy: 0.5526 - 53ms/epoch - 13ms/step
Epoch 50/10000
4/4 - 0s - loss: 0.6528 - accuracy: 0.5947 - val_loss: 0.6995 - val_accuracy: 0.5395 - 53ms/epoch - 13ms/step
Epoch 51/10000
4/4 - 0s - loss: 0.6550 - accuracy: 0.6080 - val_loss: 0.7008 - val_accuracy: 0.5789 - 53ms/epoch - 13ms/step
Epoch 52/10000
4/4 - 0s - loss: 0.6496 - accuracy: 0.5980 - val_loss: 0.7034 - val_accuracy: 0.5132 - 52ms/epoch - 13ms/step
Epoch 53/10000
4/4 - 0s - loss: 0.6558 - accuracy: 0.5748 - val_loss: 0.7035 - val_accuracy: 0.5132 - 52ms/epoch - 13ms/step
Epoch 54/10000
4/4 - 0s - loss: 0.6482 - accuracy: 0.6013 - val_loss: 0.7092 - val_accuracy: 0.5658 - 53ms/epoch - 13ms/step
Epoch 55/10000
4/4 - 0s - loss: 0.6526 - accuracy: 0.5947 - val_loss: 0.7087 - val_accuracy: 0.5263 - 52ms/epoch - 13ms/step
Epoch 56/10000
4/4 - 0s - loss: 0.6603 - accuracy: 0.6013 - val_loss: 0.7097 - val_accuracy: 0.5000 - 53ms/epoch - 13ms/step
Epoch 57/10000
4/4 - 0s - loss: 0.6685 - accuracy: 0.5980 - val_loss: 0.7093 - val_accuracy: 0.4868 - 52ms/epoch - 13ms/step
Epoch 58/10000
4/4 - 0s - loss: 0.6551 - accuracy: 0.5814 - val_loss: 0.7105 - val_accuracy: 0.5658 - 53ms/epoch - 13ms/step
Epoch 59/10000
4/4 - 0s - loss: 0.6576 - accuracy: 0.5814 - val_loss: 0.6999 - val_accuracy: 0.5263 - 51ms/epoch - 13ms/step
Epoch 60/10000
4/4 - 0s - loss: 0.6521 - accuracy: 0.6047 - val_loss: 0.6979 - val_accuracy: 0.5000 - 53ms/epoch - 13ms/step
Epoch 61/10000
4/4 - 0s - loss: 0.6545 - accuracy: 0.5880 - val_loss: 0.7002 - val_accuracy: 0.5132 - 53ms/epoch - 13ms/step
Epoch 62/10000
4/4 - 0s - loss: 0.6509 - accuracy: 0.6113 - val_loss: 0.7053 - val_accuracy: 0.4868 - 52ms/epoch - 13ms/step
Epoch 63/10000
4/4 - 0s - loss: 0.6568 - accuracy: 0.5615 - val_loss: 0.7131 - val_accuracy: 0.5658 - 60ms/epoch - 15ms/step
Epoch 64/10000
4/4 - 0s - loss: 0.6523 - accuracy: 0.5781 - val_loss: 0.7064 - val_accuracy: 0.5000 - 52ms/epoch - 13ms/step
Epoch 65/10000
4/4 - 0s - loss: 0.6557 - accuracy: 0.6080 - val_loss: 0.7122 - val_accuracy: 0.5395 - 53ms/epoch - 13ms/step
Epoch 66/10000
4/4 - 0s - loss: 0.6559 - accuracy: 0.5880 - val_loss: 0.7081 - val_accuracy: 0.5132 - 52ms/epoch - 13ms/step
Epoch 67/10000
4/4 - 0s - loss: 0.6495 - accuracy: 0.5980 - val_loss: 0.7065 - val_accuracy: 0.5000 - 53ms/epoch - 13ms/step
Epoch 68/10000
4/4 - 0s - loss: 0.6494 - accuracy: 0.5781 - val_loss: 0.7085 - val_accuracy: 0.5000 - 52ms/epoch - 13ms/step
Epoch 69/10000
4/4 - 0s - loss: 0.6461 - accuracy: 0.5781 - val_loss: 0.7109 - val_accuracy: 0.5132 - 53ms/epoch - 13ms/step
```

In [ ]:
```python
#save model
model.save("saved/ckpt")
```

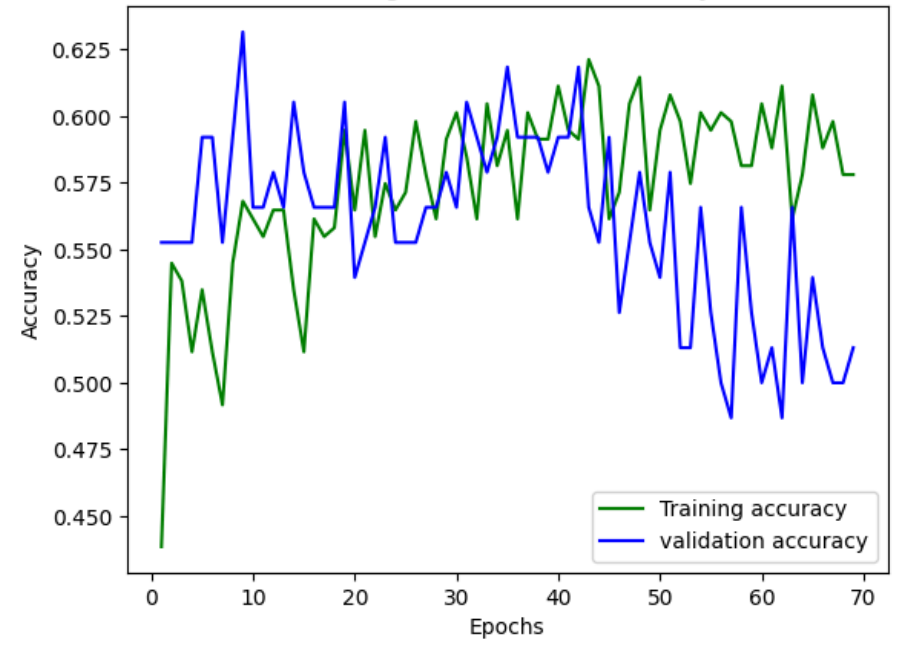
In [ ]:
```python
#save model
model_file = 'model.png'
tf.keras.utils.plot_model(model, to_file=model_file, show_shapes=True)
```

```
You must install pydot (`pip install pydot`) and install graphviz (see instructions at https://graphviz.gitlab.io/download/) for plot_model to work.
```

In [ ]:
```python
#plot loss graph
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='test')
plt.legend()
plt.title("Loss graph")
plt.show()
```



In [ ]:
```python
#plot training and validation accuracy
loss_train = history.history['accuracy']
loss_val = history.history['val_accuracy']
epochs = range(1,len(loss_val) + 1)
plt.plot(epochs, loss_train, 'g', label='Training accuracy')
plt.plot(epochs, loss_val, 'b', label='validation accuracy')
plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



In [ ]:
```python
#reshape
test_X = test_X.reshape((test_X.shape[0], n_lag, n_features))

#predict values for test data
pred = model.predict(test_X)

#reshape again
test_X = test_X.reshape((test_X.shape[0], n_lag* n_features,))
```

```
3/3 [==============================] - 0s 3ms/step
3/3 [==============================] - 0s 3ms/step
```

In [ ]:
```python
#change back from categorical
pred = np.argmax(pred, axis=1)
```

In [ ]:
```python
#calculate accuracy
prices = pd.DataFrame()
prices["Actual"] = test_y
prices["Predicted"] = pred

prices["Correct"] = (prices["Actual"] - prices["Predicted"]) == 0
incorrect = prices.loc[prices['Correct'] == False]
incorrect_len = len(incorrect)
prices_len = len(prices)

print("Accuracy = ",((prices_len-incorrect_len)/prices_len))
```

```
Accuracy =  0.6268656716417911
```

In [ ]:
```python
sklearn.metrics.classification_report(test_y, pred, zero_division=0,output_dict=True)
```

Out[ ]:
```
{'0': {'precision': 0.8,
  'recall': 0.43243243243243246,
  'f1-score': 0.5614035087719298,
  'support': 37.0},
 '1': {'precision': 0.5531914893617021,
  'recall': 0.8666666666666667,
  'f1-score': 0.6753246753246753,
  'support': 30.0},
 'accuracy': 0.6268656716417911,
 'macro avg': {'precision': 0.676595744680851,
  'recall': 0.6495495495495496,
  'f1-score': 0.6183640920483026,
  'support': 67.0},
 'weighted avg': {'precision': 0.6894887265798666,
  'recall': 0.6268656716417911,
 'f1-score': 0.6124129863328607,
 'support': 67.0}}
```

```
In [ ]:
```