Stock Volatility Forecasting Report

This is a report on analyzing and forecasting the stock price volatility based on data using `ARIMA` r **Recurrent Neural Networks (RNNs)**, **Convolutional Neural Network (C** **Generative Adversarial Network (GAN)**. The global structure of the report is:

Part I. Data analysis and cleaning

- Basic manipulation and analysis
- Data cleaning
- Data preparation

Part II. Statistical analysis

- Statistical analysis of the volatility data
- Time series analysis with `ARIMA`

Part III. Deep learning models

- Basic model: single-step, single-feature forecasting with LSTM
- Generalized model: multi-step, multi-feature forcasting with LSTM
- Advanced model: Generative Adversarial Network (GAN) with RNN and CNN.

Part IV. Conclusions and Next steps

- Conclusions
- Next steps

References:

- [My own deep learning project]
- [Using the latest advancements in deep learning to predict stock price movements]

Introduction

1. The Notebook

Follow the notebook, we can recreate all the results, notice the followings

- Upload the `stockdata3.csv` file to the root folder on google colab.
- To navigate better, use the table of contents bottom on the upper-left sidebar.
- **For clarity, all code cells are hiden, double click on the cell to get the**
- Change the parameters as indicated in the comments to create more custom outputs.
- All source code can also be found in the project file folder

2. The stock prices dataset

This report uses a Stock Prices Dataset provided by SIG.

Before analyzing the data with codes, we have the following observations.

- This dataset contains $\mathbf{6}$ different stocks $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$,$\mathbf{e}$ and $\mathbf{f}$.
- Data were collected every $\mathbf{1}$ minute, beginning from $\mathbf{Day\ 1,\ 9{:}30\ am\ to\ Day\ 362,\ 16}$
- In total, we have $\mathbf{98353\ rows\ (minutes)}$ and $\mathbf{8\ columns\ (day\ number,\ time\ s}$

Part I. Data analysis and cleaning

Basic manipulation

Code and examples

```
#@title ```basic.py```

#import numpy, pandas and matplotlib

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')


#Basic checks: find null values, set index, etc.

def basic_check(df, index_name="day"):
  """Find the null values and set index of a given DataFrame.
  :param: df, pd.DataFrame, the data, e.g. df = pd.read_csv("stockdata3.csv")
  :param: index_name, str, name of the index, must be one of the column names, e.g. inde
  :rtype: pd.DataFrame
  """
  df.set_index(index_name, inplace=True)
  df.index = pd.to_datetime(df.index,unit='D')

  #check for null entries
  print("Null values summary:\n")
  print(df.isnull().sum())

  return df

def plot_column(df, feature):
    """Plot the resampled column of df, e.g. plot_column(df, "a") plots the "a" column

    :param: df, pandas.DataFrame, the data, e.g. df = pd.read_csv("stockdata3")
    :param: feature, str, name of column to be plotted.
    """
    y = df[feature]
    y.plot(figsize=(28, 8))
    plt.xlabel('Minute number')
```

```
    plt.ylabel("Stock "+feature)
    plt.show()

def day_check(df, column_name = "day"):
  for ii in df['day'].unique():
    if(len(df[df['day']==ii])!=391):
      print("Day "+str(ii)+" has "+str(len(df[df['day']==ii]))+" minutes data!")
      print(df[df['day']==ii].tail())


def date_check(df):
  for date in df.index.unique():
    if(len(df[df.index==date])!=391):
      print("Day "+str(date)+" has "+str(len(df[df.index==date]))+" minutes data!")
      print(df[df.index==date].tail())
```

```
#@title read the file and show the head

stock_data = pd.read_csv("stockdata3.csv")
stock_data.head()
```

```
#@title days per month, minutes per day and find special day(s)

print("Mean number of days per month: "+str(len(stock_data['day'].unique())/12))
print("Mean number of sample points per day: "+str(len(stock_data)/(12*21)))
day_check(stock_data)
```

```
#@title Basic checks: find null values and fill, set index, etc.
df = basic_check(stock_data)
df.head()
```

Example: plot the stock price columns

Data Analysis

Basic aspects:

- Most days contain data for the full 391 minutes from `9:30 am` to `4:00 pm`
- Day 327 doesn't have the full number ofminutes, the data stops at `1:00 pm` that day, it only c
- Every month contains 21 days.

As a high level overview, some distinguishable features appear when we plot the data:

- Prices of a $\textbf{drop abnormally}$ at some random places.
- Prices of b seem to have relatively stable volatility in the first 7 months and last 3 months ar in the 8-th and 9-th month.
- Prices of c $\textbf{drop drastically}$ in the 6-th month, besides that, the price of c is quite stabl
- Prices of d $\textbf{drop abnormally}$ at some random places.
- Prices of d have a $\textbf{large day-to-day volatility}$ compared to the longer-term volatility
- Prices of f are very $\textbf{illiquid}$, that is, the prices don't move much on a minute-to-minute bas
- Peices of b and e have no null entries, no random drops.

```
#@title check for special day(s)
date_check(stock_data)
```

```
#@title check for random drops in prices of stock a and stock d
```

```
print(f"Stock prices of a have {len(df[df['a']==0])} random drops")
print(f"Stock prices of d have {len(df[df['d']<10])} random drops")
print(f"Stock prices of a and d drop at the same time in {df[(df['a']==0) & (df['d']<10)

df[(df['a']==0) & (df['d']<10)][["a", "d"]]
```

Data analysis

- The times of the price drops in stock a and stock d seem to be random and appear on 93 dif
- The price drops of stock a and d happend $\textbf{at the exactly same places}$.
- Abnormal values of a are all 0.0's, abnormal values of b are all 1.0's.

We conclude that $\textbf{these drops are recording errors}$, the default value of missing a price 1.0.

Data cleaning

Data cleaning: set abnormal values to NaN

We set those abnormal values in the prices of stock a and stock d as $\textbf{NaN}$, which makes it comp functionality of pandas.

```
df["a"] =df["a"].replace(0,np.nan)
df["d"] =df["d"].replace(1.0,np.nan)
#plot_column(df, "a")
```

```
#plot_column(df, "a")
#plot_column(df, "d")
```

Data cleaning: add missing data on day 327

```
#copy base data into new dataframe
df_clean=df.copy()

#replace spurious prices -> NaN
df_clean['a']=df['a'].replace(0,np.nan);
df_clean['d']=df['d'].replace(1.0,np.nan);

#references to days < and > day 327
df_left=df[df.index<"1970-11-24"]
df_right=df[df.index>"1970-11-24"]
df_special_temp=df[df.index=="1970-11-24"].copy() #copy a new version of day 327 data to p

# we miss 391-211=180 rows in the day, so we chose a normal,here 1970-01-05 and take out t
# last 180 rows, that's 211:391 and fill it with NaNs and change the index to be day 327,
df_more =df[df.index=="1970-01-05"][211:391].copy()
df_more.loc[df_more['a'] > 0, ['a','b', 'c', 'd', 'e', 'f']] = np.nan
df_more.index= [ pd.to_datetime("1970-11-24") for _ in range(df_more.shape[0])]

#concatenate
df_new=pd.concat([df_left,df_special_temp,df_more, df_right])
```

Make sure we already have what we want

- no random drops in prices of stock a and stock d
- day 327 has full 391 rows

```
#@title cleaned data

minute_new = len(df_new[df_new.index=="1970-11-24"])
print(f"Day 327 now has {minute_new} minutes of data\n\n")
plot_column(df_new, "a")
plot_column(df_new, "d")
```

Data preparation: from prices to daily volatility of

We first define the quantity(volatility measured in annualized percent return) that we're interested i
the following factors affect our code realization

- Quantities of interest
- Model assumptions
- Frequency of sampling

Quantities of interest: Daily volatility of annualized monthly percent return $\sigma_{d,m}$

Definition: the **Annualized monthly percent return** $A_m(t)$ at time t is given by

$$A_m(t) = (\frac{\text{Price at time a month later than } t}{\text{Price at time } t})^{\frac{1 \text{ year}}{1 \text{ month}}}$$
$$= (\frac{p(t+t_m)}{p(t)})^{12} - 1$$

where

- $p(t)$ denote the price at time t, we use $\mathbf{minute}$ as the unit, the same as in our `stockdata3.`
- t_m denote a month in unit of the dataset, for our `stockdata3.csv` dataset, $t_m = 391 \text{ mins}$,

---

Definition: the **Daily volatility of annualized monthly percent return** $\sigma_{d,m}(t)$ at time t is defined to
$\{A_m(x) | t \leq x \leq t + t_d\}$, where

- t_d denote the number of minutes in a day, for us $t_d = 391$. To be more precise

$$\sigma_{d,m}(t) = \sqrt{\left[\frac{1}{t_d}\sum_{j=0}^{t_d}\left(A_m(t+j) - \frac{1}{t_d+1}\sum_{i=0}^{t_d} A_m(\right.\right.}$$

Model assumptions

- We're interested in the statistics on percent returns after holding an asset for a month (but w
- Volatility is a constant in a day.

The first assumption tells us how to chose window size to compute the annualized percent return. deviation functionality in `pandas` to compute our volatility of annualized monthly percent return. Tl we need

- window size: `t_m = t_d*21=8211`
- window moving step: `1`, that is (every minute has a annualized percent return)

The second assumption let us estimate the volatility by computing sample deviation of the percen

- sample size: `t_d=391`, that is we use a consecutive sequence of 391 data points to compute
- sample frenquency: `t=1`, that is we'll consider the daily movement of the volatility.

```
#@title Volatility in minutes

plist=["a", "b", "c", "d", "e","f"]

t_d = 391 #number of timesteps in a day, used for windowing
t_m = t_d*21 #number of timesteps in a month, used for shift in computing monthly log re
A_m = (df_new[plist].shift(-t_m)/ df_new[plist])**(12)-1
sigma_fine = A_m.rolling(t_d,min_periods=1).std()
sigma_fine.shape[0]==df_new.shape[0]
for stock in plist:
  plot_column(sigma_fine, stock)
```

```
#@title Daily volatility

sigma=pd.DataFrame(np.random.randn(len(A_m.index.unique()),6),columns=plist)
sigma.index=A_m.index.unique().copy()
for date in A_m.index.unique():
  for stock in plist:
    sigma.loc[sigma.index==date, stock]=A_m[A_m.index==date][stock].std(ddof=1,skipna=Tr

for stock in plist:
  plot_column(sigma, stock)
```

```
#@title Final volatility dataset
new_sigma = sigma[list("abcef")][:-21]
new_sigma.isnull().sum()
```

Part II. Statistical analysis and models

Correlation analysis

Though it seems that there's no obvious correlation among the 6 stocks, and some of them even b
of the report, we compute several different correlations (
Naive correlation, Pearson correlation, local Pearson correlation, instant
and related statistics in order to

- Test the validity of our observations (i.e. no two stocks are apprantly correlated).
- Chose source and target stocks for later machine learning(especially deep learning) models.

By doing so, we can get more understanding about the 'quality' and 'inner relations' of the data. If a
the stock that we want to predict (e.g. "f"), then there is no need for us to use it in the training of ou
one stock has higher-than-random correlations to another stock, then it's good to use one of them
this case, **to determine which stock volatility leads, and which stock volatil**
Dynamic time wrapping.

Code and Examples

```
#@title ```correlation.py```

%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

#matplotlib.rcParams['axes.labelsize'] = 14
#matplotlib.rcParams['xtick.labelsize'] = 12
#matplotlib.rcParams['ytick.labelsize'] = 12
#matplotlib.rcParams['text.color'] = 'k'

from pylab import rcParams
rcParams['figure.figsize'] = 18, 8


import statsmodels.api as sm
import warnings
import itertools
warnings.filterwarnings("ignore")


import seaborn as sns
import scipy.stats as stats
from scipy.signal import hilbert, butter, filtfilt
from scipy.fftpack import fft,fftfreq,rfft,irfft,ifft

# For the dynamic_time_warping function
```

```
!pip install dtw
from dtw import dtw, accelerated_dtw


def pearson(df, feature1, feature2):
    """Compute and plot the overall pearson correlation of feature1 and feature2,
    e.g. pearson(df, "b", "f") compute and plot the overall pearson correlation between

    :param: df, pandas.DataFrame, data contains different features (columns)
    :param: feature1, str, name of the column, e.g. "b"
    :param: feature2, str, name of another column e.g. "f"
    """
    overall_pearson_r = df.corr()[feature1][feature2]
    print(f"Pandas computed Pearson r: {overall_pearson_r}")



    r, p = stats.pearsonr(df.dropna()[feature1], df.dropna()[feature2])
    print(f"Scipy computed Pearson r: {r} and p-value: {p}")

    #Compute rolling window synchrony
    f,ax=plt.subplots(figsize=(14,3))
    df[[feature1, feature2]].rolling(window=30,center=True).median().plot(ax=ax)
    ax.set(xlabel='Time',ylabel='Pearson r')
    ax.set(title=f"Overall Pearson r = {np.round(overall_pearson_r,2)}")


def local_pearson(df, feature1, feature2):
    """Compute and plot the local pearson correlation of feature1 and feature2,
    e.g. local_pearson(df, "a", "b") compute and plot the local pearson correlation betw

    :param: df, pandas.DataFrame, data contains different features (columns)
    :param: feature1, str, name of the column, e.g. "a"
    :param: feature2, str, name of another column e.g. "b"

    """
    # Set window size to compute moving window synchrony.
    r_window_size = 120
    # Interpolate missing data.
    df_interpolated = df[[feature1, feature2]].interpolate()
    # Compute rolling window synchrony
    rolling_r = df_interpolated[feature1].rolling(window=r_window_size, center=True).cor
    f,ax=plt.subplots(2,1,figsize=(14,6),sharex=True)
    df[[feature1, feature2]].rolling(window=30,center=True).median().plot(ax=ax[0])
    ax[0].set(xlabel='Frame',ylabel='Smiling Evidence')
    rolling_r.plot(ax=ax[1])
    ax[1].set(xlabel='Frame',ylabel='Pearson r')
    plt.suptitle("Smiling data and rolling window correlation")


def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    return b, a
```

```
def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = filtfilt(b, a, data)
    return y

def instant_phase_sync(df, feature1, feature2):
    """Compute and plot the instantaneous phase synchrony of feature1 and feature2,
    e.g. instant_phase_sync(df, "a", "b") compute and plot the instantaneous phase synch

    :param: df, pandas.DataFrame, data contains different features (columns)
    :param: feature1, str, name of the column, e.g. "a"
    :param: feature2, str, name of another column e.g. "b"
    """
    lowcut  = .01
    highcut = .5
    fs = 30.
    order = 1
    d1 = df[feature1].interpolate().values
    d2 = df[feature2].interpolate().values
    y1 = butter_bandpass_filter(d1,lowcut=lowcut,highcut=highcut,fs=fs,order=order)
    y2 = butter_bandpass_filter(d2,lowcut=lowcut,highcut=highcut,fs=fs,order=order)

    al1 = np.angle(hilbert(y1),deg=False)
    al2 = np.angle(hilbert(y2),deg=False)
    phase_synchrony = 1-np.sin(np.abs(al1-al2)/2)
    N = len(al1)

    # Plot results
    f,ax = plt.subplots(3,1,figsize=(14,7),sharex=True)
    ax[0].plot(y1,color='r',label='y1')
    ax[0].plot(y2,color='b',label='y2')
    ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102),ncol=2)
    ax[0].set(xlim=[0,N], title='Filtered Timeseries Data')
    ax[1].plot(al1,color='r')
    ax[1].plot(al2,color='b')
    ax[1].set(ylabel='Angle',title='Angle at each Timepoint',xlim=[0,N])
    phase_synchrony = 1-np.sin(np.abs(al1-al2)/2)
    ax[2].plot(phase_synchrony)
    ax[2].set(ylim=[0,1.1],xlim=[0,N],title='Instantaneous Phase Synchrony',xlabel='Time
    plt.tight_layout()
    plt.show()



def dynamic_time_warping(df, feature1, feature2):
    """Compute and plot dynamic time warping of feature1 and feature2,
    e.g. instant_phase_sync(df, "a", "b") compute and plot the dynamic_time_wraping betw

    :param: df, pandas.DataFrame, data contains different features (columns)
    :param: feature1, str, name of the column, e.g. "a"
    :param: feature2, str, name of another column e.g. "b"
    """
    d1 = df[feature1].interpolate().values
```

```
    d2 = df[feature2].interpolate().values
    d, cost_matrix, acc_cost_matrix, path = accelerated_dtw(d1,d2, dist='euclidean')


    plt.imshow(acc_cost_matrix.T, origin='lower', cmap='gray', interpolation='nearest')
    plt.plot(path[0], path[1], 'w')
    plt.xlabel(feature1)
    plt.ylabel(feature2)
    plt.title(f'DTW Minimum Path with minimum distance: {np.round(d,2)}')
    plt.show()
```

```
#@title Example: Naive correlation.
new_sigma.corr()
```

```
#@title Example: Pearson correlation
pearson(new_sigma, "a", "c")
```

```
#@title Example: local Pearson correlation
local_pearson(new_sigma, "f", "e")
```

```
#@title Example: instantaneous phase synchronization
instant_phase_sync(new_sigma, "a", "b")
```

```
#@title Example: dynamic time wraping
dynamic_time_warping(new_sigma, "a", "b")
```

Data analysis

Inspecting the correlations from different angles, we find

- $\textbf{Stock } b \textbf{ and stock } c \textbf{ have the highest correlation, 0.864241,}$ among all the
- $\textbf{Stock } b \textbf{ and stock } f$, $\textbf{Stock } c \textbf{ and stock } f$ also have high positive correlation$.
- All other pairs have relatively low correlations.
- $\textbf{Stock } b \textbf{ slightly leads stock } a$.

We conclude that

- $\textbf{The assumption that no two stocks have apparent correlation is wro}$
- It's reasonable to $\textbf{use stock } a \textbf{ and stock } f \textbf{ as targets and the other stocks as source for fo}$.

Statistical models for predicting volatility

As we'll see below, some remarkable patterns (e.g. seasonality pattern) naturally appear in our dat

- We visualize our data using $\textbf{time-series decomposition}$ that allows us to decompose trend, seasonality, and noise.
- We $\textbf{train an ARIMA (Autoregressive Integrated Moving Average) m}$ Volatility values. To get optimal output, we first
- Use $\textbf{grid search}$(in a range) to get the optimal parameters for the ARIMA mode.
- Fit arima models to predict next month's volatility

```
#@title ```time_series.py```


def plot_column(df, feature):
    """Plot the resampled column of df, e.g. plot_column(df, "Inflation") plots the "Inf

    :param: df, pandas.DataFrame, the data, e.g. df = pd.read_excel("USMacroData", "All"
    :param: feature, str, name of column to be plotted.
    """
    y = df[feature]
    y.plot(figsize=(15, 6))
    plt.show()



def plot_component(df, feature):
    """Decompose the time series data into trend, seasonal, and residual components.

    :param: df, pd.DataFram.
    :param: feature, str,column name/feature name we want to decompose
    :rtype: None
    """
    decomposition = sm.tsa.seasonal_decompose(df[feature], model='additive',freq=52)
    fig = decomposition.plot()
    plt.show()
    ###### This section uses ARIMA to analyze the data and make predictions.############

# Grid search to find the best ARIMA parameters
def arima_parameters(df, feature, search_range=2):
    """Grid search for the optimal parameters of the Arima model for given data (df) and
    :param: df, pdf.DataFrame, data
    :param: feature, str, feature name.
    :param: search_range, int, the range for the search of the parameters, the default v
    """
    p = d = q = range(0, search_range)
    pdq = list(itertools.product(p, d, q))
    seasonal_pdq = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))]

    minimal_aic = 0
    optimal_param =[]
    for param in pdq:
        for param_seasonal in seasonal_pdq:
            try:
                mod = sm.tsa.statespace.SARIMAX(df[feature],order=param, seasonal_order=
                results = mod.fit()
                print('ARIMA{}x{}12 - AIC:{}'.format(param, param_seasonal, results.aic)
                if results.aic < minimal_aic:
                    optimal_param = [param, param_seasonal]
                    minimal_aic = results.aic
                    print(minimal_aic)
            except:
                continue
    print('\n Optimal parameters ARIMA{}x{}12 - Minimal AIC:{}'.format(optimal_param[0],
    return optimal_param[0], optimal_param[1]
```

```
plot_component(new_sigma, "a")
```

```
arima_parameters(new_sigma, "a")
```

```
arima_parameters(new_sigma, "b")
```

```
arima_parameters(new_sigma, "f")
```

```
import statsmodels.formula.api as smf
import statsmodels.tsa.api as smt
import statsmodels.api as sm

a_model = sm.tsa.ARMA(new_sigma['a'].values,(1,0,1)).fit(disp=False)
b_model = sm.tsa.ARMA(new_sigma['b'].values,(1,0,0)).fit(disp=False)
f_model = sm.tsa.ARMA(new_sigma['f'].values,(1,0,0)).fit(disp=False)

print(a_model.params)
print(b_model.params)
print(f_model.params)
```

Model predictions

```
beg=len(new_sigma['a'].values)
predict_a = a_model.predict(start=beg,end=beg+21)
predict_b = b_model.predict(start=beg,end=beg+21)
predict_f = f_model.predict(start=beg,end=beg+21)
print(predict_a)
print(predict_b)
print(predict_f)
```

Data Analysis

- Components plot show the obvious seasonality, for example, for stock a, we can find an alte alternate between high values and low values in a period of roughly $3-4$ months.
- The optimal ARIMA parameters for "a" are `(1, 0, 1)x(0, 0, 1, 12)`
- As we forecast further out into the future, we becomes less confident in our values. This is re by our model, which grow larger as we move further out into the future.
- We'll do more carefully analysis of the predictions for the deep learning models.

Part III. Deep learning models

Basic model: single-step, single-feature forecasting with LS

Recurrent Neural Networks (RNNs) are good fits for time-series analysis because R designed to capture patterns developing through time.

However, vanilla RNNs have a major disadvantage---the vanishing gradient problem---"the changes so small, making the network unable to converge to a optimal solution.

LSTM (Long-Short Term Memory) is a variation of vanilla RNNS,it overcomes the va problem by clipping gradients if they exceed some constant bounds.

In this section, we will

- Process the data to fit the LSTM model
- **Build and train the LSTM model for single-step, single-feature predi** volatility with only today's values of the other 5 stocks).

```
#@title imports
import math
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import LSTM, Dense
```

```
#@title Data preparation

def transform_data(df, features, targets, look_back = 0, look_forward = 1, split_ratio =
  """transform the data in a custom form.

  :param: df, pd.DataFram, the data,
    e.g. df = pd.read_excel("USMacroData.xls", "All")
  :param: features, list of strs, the features to be uses as the source features,
     e.g. ["Wage", "Consumption"]
  :param: look_back, int, number of days to look back in historic data,
     e.g. look_back = 11 means we use the last (11+1)=12 months' data to predict the fut
  :param:look_forward, int, num of days to look forward
    e.g. look_forward = 3 means we want to predict next 3 months' data
  :param: split_ratio, float, split the data into training dataset and testing dataset b
    e.g. split ratio=0.7 means we use the first 70% of the data as training data, the la
```

```
  :rtype: np.arrays, x_train, y_train, x_test, y_test
  """
  x, y = [], []
  for i in range(look_back, len(df) - look_forward):
      assert look_back < len(df)-look_forward, "Invalid look_back, look_forward values"

      x.append(np.array(df[i-look_back : i+1][features]))
      y.append(np.array(df[i+1: i+look_forward+1][targets]).transpose())

  # List to np.arrary
  x_arr = np.array(x)
  y_arr = np.array(y)

  split_point = int(len(x)*split_ratio)

  return x_arr[0:split_point], y_arr[0:split_point], x_arr[split_point:], y_arr[split_po

features = ["a","b", "c", "e", "f"]
targets = ["a"]
x_train, y_train, x_test, y_test = transform_data(new_sigma, features=features, targets

#Note that all returned np.arrays are three dimensional.
#Need to reshape y_train and y_test to fit the LSTM

# For the basic model only
y_train = np.reshape(y_train, (y_train.shape[0], -1))
y_test = np.reshape(y_test, (y_test.shape[0], -1))
```

```
#@title Build and train the LSTM model

# To match the Input shape (1,5) and our x_train shape is very important.

def train_model(Optimizer, x_train, y_train, x_test, y_test):
  model = Sequential()
  model.add(LSTM(50, input_shape=(1, 5)))
  model.add(Dense(1))

  model.compile(loss="mean_squared_error", optimizer=Optimizer, metrics =["accuracy"])
  scores = model.fit(x=x_train,y=y_train, batch_size=1, epochs = 100, validation_data =

  return scores, model
```

```
#@title Make sure data forms are correct
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
#@title LSTM with SGD, RMSprop, Adam optimizers, epochs = 100
#SGD_score, SGD_model = train_model(Optimizer = "sgd", x_train=x_train, y_train = y_trai
RMSprop_score, RMSprop_model = train_model(Optimizer = "RMSprop", x_train=x_train, y_tra
#Adam_score, Adam_model = train_model(Optimizer = "adam", x_train=x_train, y_train = y_t
```

```
#@title Plot result

def plot_result(score, optimizer_name, label = "loss"):
  plt.figure(figsize=(18, 8))
  plt.plot(range(1, 101), score.history["loss"], label ="Training Loss")
  plt.plot(range(1,101), score.history["val_loss"], label="Validation Loss")
  plt.axis([1,100, 0, 0.2])
  plt.xlabel("Epoch")
  plt.ylabel("Loss")
  plt.title("Train and Validation Loss using "+optimizer_name + "optimizer")
  plt.legend()
  plt.show()
```

```
#@title Plot result
plot_result(RMSprop_score, "RMSprop")
```

```
#@title Plot predictions
def plot_predict(model, x_train, x_test, y_train, y_test):
  train_predict = RMSprop_model.predict(x_train)
  test_predict = RMSprop_model.predict(x_test)

  # Calculate root mean squared error.
  trainScore = math.sqrt(mean_squared_error(y_train, train_predict))
  print('Train Score: %.2f RMSE' % (trainScore))
  testScore = math.sqrt(mean_squared_error(y_test, test_predict))
  print('Test Score: %.2f RMSE' % (testScore))

  plt.figure(figsize=(18, 8))
  plt.plot(train_predict)
  plt.plot(y_train)
  plt.show()

  plt.figure(figsize=(18, 8))
  plt.plot(test_predict)
  plt.plot(y_test)
  plt.show()
```

```
#@title Plot predictions
plot_predict(RMSprop_model, x_train = x_train, y_train=y_train, x_test = x_test, y_test=
```

Data Analysis

We only trained the model for 100 epochs, feel free to modify it to any number as long as we have results we find during the experiments

- LSTM with Adam or RMSprop optimizers work better than the SGD optimizer in this project.
- Each model fits the training dataset very well.
- **The prediction captures the range and characteristics of the real dat**
- Most importantly, the model predict the large rises or drops in the volatility before they happ investments.

Generalized model: multi-step, multi-feature forecasting v

We build a multi-step, multi-feature LSTM model in this section. That means we can use several-da features in the future.

For example, we can use last 12-day's data of a, b, c, f to predict next thr section, we

- Process the data to fit the requirements of all possible multi-step, multi-feature prediction ta
- We modify the LSTM model accordingly.
- Plot the 3-day prediction for a and f with last 12-day's data of a, b, c and f.

```
#@title Data preparation
```

```
def transform_data(df, features, targets, look_back = 0, look_forward = 1, split_ratio =
  """transform the data in a custom form.

  :param: df, pd.DataFram, the data,
    e.g. df = pd.read_excel("USMacroData.xls", "All")
  :param: features, list of strs, the features to be uses as the source features,
     e.g. ["Wage", "Consumption"]
  :param: look_back, int, number of days to look back in historic data,
     e.g. look_back = 11 means we use the last (11+1)=12 months' data to predict the fut
  :param:look_forward, int, num of days to look forward
    e.g. look_forward = 3 means we want to predict next 3 months' data
  :param: split_ratio, float, split the data into training dataset and testing dataset b
    e.g. split_ratio=0.7 means we use the first 70% of the data as training data, the la
  :rtype: np.arrays, x_train, y_train, x_test, y_test
  """
  x, y = [], []
  for i in range(look_back, len(df) - look_forward):
      assert look_back < len(df)-look_forward, "Invalid look_back, look_forward values"

      x.append(np.array(df[i-look_back : i+1][features]))
      y.append(np.array(df[i+1: i+look_forward+1][targets]).transpose())

  # List to np.arrary
  x_arr = np.array(x)
  y_arr = np.array(y)

  split_point = int(len(x)*split_ratio)

  return x_arr[0:split_point], y_arr[0:split_point], x_arr[split_point:], y_arr[split_po

features = ["a", "b", "c", "f"]
targets = ["a", "f"]
x_train, y_train, x_test, y_test = transform_data(new_sigma, features=features, targets

#Note that all returned np.arrays are three dimensional.
#Need to reshape y_train and y_test to fit the LSTM

# For the multi-step LSTM model only
y_train = np.reshape(y_train, (y_train.shape[0], -1))
y_test = np.reshape(y_test, (y_test.shape[0], -1))


#@title Make the data forms are all correct
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_train.shape)
```

```
#@title Scaling, vectorize and de_vectorize
```

```
def scale(arr, df):
  """Scale the data to range (-1,1) to better fit the LSTM model

  :param: arr, np.array, the array to be scaled
  :param: df, pd.DataFrame, to provide the max and min for us to scale arr
  TODO: maybe we don't need the df parameter?
  """
  global_max = max(df.max())
  global_min = min(df.min())
  arr = -1 + (arr-global_min)*2/(global_max-global_min)
  return arr

def de_scale(arr, df):
  """de Scale the data from range (-1,1) to its original range

  :param: arr, np.array, the array to be scaled
  :param: df, pd.DataFrame, to provide the max and min for us to scale arr
  """
  global_max = max(df.max())
  global_min = min(df.min())
  arr = global_min+(arr+1)*(global_max-global_min)/2
  return arr

def vectorize(y_train):
  """To vectorize an np.array.

  :param: y_train, np.array, the array to be vectorized
  :rtype: np.array, vectorized array.
  """
  return np.reshape(y_train, (y_train.shape[0], -1))

def de_vectorize(y_train, row, col):
  """To de_vectorize an np.array: transfrom from 2-dim np.array to its original form.

  :param: y_train, np.array, the array to be de_vectorized
  :rtype: np.array, de_vectorized array.
  """
  return np.reshape(y_train,(y_train.shape[0], row, col))


def train_multi_step_model(Optimizer, x_train, y_train, x_test, y_test):
  model = Sequential()
  model.add(LSTM(50, input_shape=(12, 4)))
  model.add(Dense(6))

  model.compile(loss="mean_squared_error", optimizer=Optimizer, metrics =["accuracy"])
  scores = model.fit(x=x_train,y=y_train, batch_size=1, epochs = 200, validation_data = (x

  return scores, model



#@title Train the model. Change the optimizer parameter to use other optimizers, e.g. "a
RMS_score, RMS_model = train_multi_step_model(Optimizer = "RMSprop", x_train=x_train, y_
```

```
#@title Make predictions with the trained model

train_predict = RMS_model.predict(x_train)
test_predict = RMS_model.predict(x_test)

#test_predict = SGD_model.predict(x_test)
#test_predict = de_scale(test_predict, df)
#y_origin =de_scale(y_test, df)
```

```
test_predict.shape
```

```
#@title Plot Multi-step, Multi-feature predictions.
def predict_plot(df, y_predict, targets):
  """ Plot the multi-step, multi-result predictions.

  :param: df, pd.DataFrame, e.g. df = pd.read_excel("USMacroData.xls", "All")
  :param: y_predict, 2-dim np.array, the model-predicted values, in each row, it has loo
          In our example, look_forward = 3, number of target features =2 ("Inflation", "
  :param: targets, list, target features, e.g ["Inflation", "Unemployment"]
  """

  y_predict = de_vectorize(y_predict, 2, 3)
  assert y_predict.shape[1] == len(targets), "Incompatible size of targets and dataset"
  assert df.shape[0] == y_predict.shape[0], "Incompatible original data rows and y_predi
```

```
  look_forward = y_predict.shape[2]

  for index, target in enumerate(targets):
    plt.figure(figsize=(17, 8))
    plt.plot(df[0:12][target])
    for i in range(len(y_predict)):
      y = list(y_predict[i][index])
      x = list(df.index[i: i+look_forward])
      data = pd.DataFrame(list(zip(x, y)), columns =[df.index.name, target])
      data = data.sort_values(df.index.name)
      data.set_index(df.index.name, inplace=True)

      if i < 12:
        plt.plot(df[i: i+look_forward][target])
        plt.plot(data)
        plt.xlabel("Date")
        plt.ylabel(target)
        plt.title("3-month predictions of " + target)
    plt.show()
```

```
new_sigma.index.name="day"
new_sigma[165:][["a", "f"]].index.name
```

To read to graph below

- Each short line segment is a 3-day prediction: start, middle, end point of the line segment me data respectively.
- X axies are the data.
- The long line is the real data.

```
y_predict = de_vectorize(test_predict, 2, 3)
y_predict.shape
y_predict.shape[1] == len(targets)
new_sigma[165:][["a", "f"]].shape[0] == y_predict.shape[0]
look_forward = y_predict.shape[2]

for index, target in enumerate(targets):
  plt.figure(figsize=(17, 8))
  plt.plot(new_sigma[165:][["a", "f"]][0:12][target])
  for i in range(len(y_predict)):
    y = list(y_predict[i][index])
    x = list(new_sigma[165:][["a", "f"]].index[i: i+look_forward])
    data = pd.DataFrame(list(zip(x, y)), columns =[new_sigma[165:][["a", "f"]].index.name,
    data = data.sort_values(new_sigma[165:][["a", "f"]].index.name)
    data.set_index(new_sigma[165:][["a", "f"]].index.name, inplace=True)

    if i < 12:
      plt.plot(new_sigma[165:][["a", "f"]][i: i+look_forward][target])
      plt.plot(data)
```

```
    plt.xlabel("Date")
    plt.ylabel(target)
    plt.title("3-day predictions of " + target)
plt.show()
```

Data Analysis

Though the dataset is not big enough, we still successfully capture several features in the predicti

- **Model predictions shows similar trend as the real data**, e.g. from the pred more or less in the most correct range and goes in the same direction as the real data.
- **The model captures the range of the real data very precisely**.
- **All 3-day predictions are liquid**, which means the modell successfully captures th

Advanced model: Generative Adversarial Network (GAN

Generative Adversarial Networks (GAN) have been a successful model in generat **The idea that GANs can to used to predict time-series data is new and ex** in learning characteristics of data, our model is based on the **assumptions**.

- Values of a **feature has certain patterns** and behavior (characteristics).
- **The future values of a feature should follow more or less the same pa** operating in a totally different way, or the economy drastically changes).

Our **goal** is that

- Generate future data that has similar (surely not exactly the same) distribution as the histori

In our model, we use

- **LSTM as a time-series generator**.
- **1-dimensional CNN as a discriminator**.

```
#@title imports

import keras
from keras.layers import Dense, Dropout, Input
from keras.models import Model,Sequential
from tqdm import tqdm
from keras.layers.advanced_activations import LeakyReLU
from keras.layers import LSTM, Conv1D, MaxPool1D, BatchNormalization, Reshape, Flatten
```

```
#@title Data preparation

#@title Data preparation

def transform_data(df, features, targets, look_back = 0, look_forward = 1, split_ratio =
  """transform the data in a custom form.

  :param: df, pd.DataFram, the data,
    e.g. df = pd.read_excel("USMacroData.xls", "All")
  :param: features, list of strs, the features to be uses as the source features,
     e.g. ["Wage", "Consumption"]
```

```
    :param: look_back, int, number of days to look back in historic data,
       e.g. look_back = 11 means we use the last (11+1)=12 months' data to predict the fut
    :param:look_forward, int, num of days to look forward
      e.g. look_forward = 3 means we want to predict next 3 months' data
    :param: split_ratio, float, split the data into training dataset and testing dataset b
      e.g. split_ratio=0.7 means we use the first 70% of the data as training data, the la
    :rtype: np.arrays, x_train, y_train, x_test, y_test
    """
    x, y = [], []
    for i in range(look_back, len(df) - look_forward):
        assert look_back < len(df)-look_forward, "Invalid look_back, look_forward values"

        x.append(np.array(df[i-look_back : i+1][features]))
        y.append(np.array(df[i+1: i+look_forward+1][targets]).transpose())

    # List to np.arrary
    x_arr = np.array(x)
    y_arr = np.array(y)

    split_point = int(len(x)*split_ratio)

    return x_arr[0:split_point], y_arr[0:split_point], x_arr[split_point:], y_arr[split_po

features = ["a","c","e", "f"]
targets = ["a", "b"]
x_train, y_train, x_test, y_test = transform_data(new_sigma, features=features, targets

#Note that all returned np.arrays are three dimensional.
#Need to reshape y_train and y_test to fit the LSTM

# For the multi-step LSTM model only
y_train = np.reshape(y_train, (y_train.shape[0], -1))
y_test = np.reshape(y_test, (y_test.shape[0], -1))


#@title Make sure all data forms are as what we want

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

Model architecture: LSTM generator

It's a 1-layer LSTM model.

- 50 hidden layers of LSTM cells
- 1 dense layer with 6 (2*3) dimensional output, since we have 2 features and 3 months to pre

```
#@title Create generator
def create_generator():
  generator = Sequential()
  generator.add(LSTM(50, input_shape=(12,4)))
  generator.add(Dense(6))
  generator.compile(loss="mean_squared_error", optimizer="RMSprop", metrics=["accuracy"]

  return generator

generator = create_generator()
generator.summary()
```

Model architecture: CNN discriminator

The structure of the discriminator is given by

- Reshape layer. Each row in y_train is acturally 1-dimensional (6,), which is different from (6,1)
- 1-dimensional Convolutional layer with 32, 3×1 filters to capture the characteristics of 3-mo
- LeakyReLU layer
- Dropout layer. Random reconfigurate 10% of the weights to zero to prevent overfitting.
- 1-dimensional Convolutional layer with 64, 3×1 filters to capture more characteristics of the
- Batchnormalization layer. To normalize the data.
- 1 Dense layer with 50 hidden nets.
- Dropout layer.
- 1 Dense layer with 1 net.

```
#@title Create discriminator
# CNN discriminator, Learn the distribution of the price.
# The goal of the gan model is to study the "characteristics" of, for example, the "Infl
# The generator tries to generate "Inflation" data as real as possible based on the othe
FILTER_SIZE = 3
NUM_FILTER = 32
INPUT_SIZE = 3 # num of days we want to predict
MAXPOOL_SIZE = 1 # our data set is small, so we don't even need it
BATCH_SIZE = 1 # our data set is small, we don't need large batch size
STEP_PER_EPOCH = 612//BATCH_SIZE
EPOCHS = 10


def create_discriminator():
```

```
  discriminator = Sequential()
  discriminator.add(Reshape((6,1), input_shape=(6,)))
  discriminator.add(Conv1D(NUM_FILTER, FILTER_SIZE, input_shape = (6,1)))
  discriminator.add(LeakyReLU(0.2))
  discriminator.add(Dropout(0.1))
  discriminator.add(Conv1D(2*NUM_FILTER, FILTER_SIZE))
  discriminator.add(BatchNormalization())

  discriminator.add(Dense(units=50))
  discriminator.add(Dropout(0.1))

  #reduce the dimension of the model to 1
  discriminator.add(Flatten())

  discriminator.add(Dense(units=1, activation="sigmoid"))

  discriminator.compile(loss="mean_squared_error", optimizer="RMSprop")
  return discriminator

discriminator = create_discriminator()
discriminator.summary()
```

```
#@title Create a GAN model with LSTM as the generator and CNN as the discriminator
def create_gan(discriminator, generator):
    discriminator.trainable=False
    gan_input = Input(shape=(12,4))
    x = generator(gan_input)
    gan_output= discriminator(x)
    gan= Model(inputs=gan_input, outputs=gan_output)
```

```
    gan.compile(loss='mean_squared_error', optimizer='adam')
    return gan
gan = create_gan(discriminator, generator)
gan.summary()
```

```
#@title Training function for the entangled GAN model
def training(x_train, y_train, x_test, y_test, epochs=1, random_size=128):

    #Loading the data
    random_count = 4*x_train.shape[0] / random_size

    # Creating GAN
    generator= create_generator()

    #y_lstm = np.reshape(y_train, (y_train.shape[0],1))
    #scores = generator.fit(x=x_train,y=y_lstm, batch_size=1, epochs = 100, validation_d
    #plt.plot(generator.predict(x_train))
    #plt.plot(y_lstm)
    #plt.show()


    discriminator= create_discriminator()
    gan = create_gan(discriminator, generator)

    for e in range(1,epochs+1 ):
        print("Epoch %d" %e)
        for index in tqdm(range(random_size)):
        #generate  random noise as an input  to  initialize the  generator
            feature = x_train[np.random.randint(low=0,high=x_train.shape[0],size=random_

            # Generate fake MNIST images from noised input
            fake_money = generator.predict(feature)
            #print(fake_money)
            #print(fake_money.shape)
            # Get a random set of  real images

            #real_money =y_train[np.random.randint(low=0,high=y_train.shape[0],size=rand
            upper_bound = int(np.random.randint(low=random_size, high=y_train.shape[0],

            real_money = y_train[upper_bound-random_size: upper_bound]
```

```
            real_money = np.reshape(real_money, (real_money.shape[0],6))
            #print(real_money)
            #print(real_money.shape)

            #Construct different batches of  real and fake data
            combination = np.concatenate([real_money, fake_money])

            # Labels for generated and real data
            y_dis=np.zeros(2*random_size)
            y_dis[:random_size]=0.9

            #Pre train discriminator on  fake and real data  before starting the gan.
            discriminator.trainable=True
            discriminator.train_on_batch(combination, y_dis)

            #Tricking the noised input of the Generator as real data
            trick_feature = x_train[np.random.randint(low=0,high=x_train.shape[0],size=r
            y_gen = np.ones(random_size)

            # During the training of gan,
            # the weights of discriminator should be fixed.
            #We can enforce that by setting the trainable flag
            discriminator.trainable=False

            #training  the GAN by alternating the training of the Discriminator
            #and training the chained GAN model with Discriminator’s weights freezed.
            gan.train_on_batch(trick_feature, y_gen)

        if e == 1 or e % 20 == 0:
          #plot generator_predict(x_test) and y_test on the same graph
          plt.figure(figsize=(17,8))
          plt.plot(generator.predict(x_train))
          y_plot = np.reshape(y_train, (y_train.shape[0],6))
          plt.plot(y_plot)
          plt.show()

training(x_train, y_train, x_test, y_test, epochs=100, random_size=128)
```

Data Analysis

- $\textbf{Our LSTM generator is not pre-trained}$, which means $\textbf{The GAN model l}$ get results as good as the previous models, but $\textbf{this experimental model shows p}$
- The GAN model successfully learned the correct range.
- $\textbf{The GAN model learns the most drastic characteristics of the data}$. N

Part IV. Conclusions and Next steps

Conclusions

In this project on analyzing and forecasting the Stock price volatility data we managed to accompl

- $\textbf{Data cleaning in Part I}$

- Basic manipulation: read the file, find null values and set index and some column plotti
- Get rid of the random drops in the prices of stock a and stock d.

- **Statistical analysis in Part II**
  - Correlation analysis: compute different correlations and use to to validate our choice o
  - Time series analysis with `ARIMA`: grid search for optimal parameters and train the ARIM
- **Build 3 Deep learning models from basic one to advanced one in Par**
  - Basic model: single-step, single-feature forecasting with LSTM
  - Generalized model: multi-step, multi-feature forcasting with LSTM
  - Advanced model: Generative Adversarial Network (GAN) with LSTM and CNN. We read
- **The accuracy rate for going up and going down from the LSTM mod**
- **The higher the sample frequency, the harder to train the deep learni**
- **The relative importance of more and less recent samples is handled b**
- Stock volatility shows certain seasonality.
- Several stock pairs show correlations.
- LSTM models capture the trend of the data developed through time
- The GAN model successfully learned the correct range.
- The GAN model learns the most drastic characteristics of the data.

Next steps

The `stockdata3.csv` is not a big dataset, the following are a few furthur steps that we have done b directions we can try to investigate:

- It's natural to **try different combinations of the source-target stocks split** the prediction of stock c's future volatilities, since they have a quite high correlation. It can be code.
- **Pre-train the LSTM model in the GAN model. In this way, the mode** .