Initialize ndarray: np.zeros((2,3)), np.ones((4,2)), np.arrange(0,10,2) = [0, 2, 4, 6, 8], np.arrange(x) = [0, 1, 2, ⋯, x-1]
np.linspase(0,10,5) = [0, 2.5, 5, 7.5, 10] 0 到 10 等距离 5 个数 包括 10
np.random.random(3) 3 个随机数 np.random.random((4,2)) tuple(4,2)4 行 2 列随机数矩阵
reshape: b=a.reshape(4,3) 省略了 tuple 括号 不改变 a reshape 前后 size 必须不变
Python list 相加 = append，ndarray 相加 = 对应元素相加
a = np.arrange(2, 10, 2) = [2, 4, 6, 8] a+4 = [6, 8, 10, 12] a*2 = [4, 8, 12, 16] a/2 = [1., 2., 3., 4.] /浮点数除法 //整数除法 np.sqrt(a) np.log(a)
A[0:2, 0:4] 取前两行，每行前四个元素
负 index: 尾部为 0，从尾部开始数第几个 [0] = [-length] [-1] = 最后一个元素
a[start:stop:step] a[start:stop] = start to stop-1 a[start:] = from start to end a[:stop] = from beginning to stop-1 a[:] = copy of whole string
A = np.random.random((4,4)) cond = A < 0.5 = [[T,F,F,T],[...],[...],[...]] A[cond] = [所有 cond 是 true 的数的集合] = [A<0.5]

t = pd.Series([12,-4,7,9], index=['a','b','c','d'] 默认的 index = [0,1,2,3] t.values = [12,-4,7,9]
pd.Series(python dic) python dic = {key1:value1, k2:v2, k3:v3, ...}
t[0:3] 含左不含右 t['Tom':'Mary']包含 Mary t[[0,2]] = 下标 0 和下标 2 = t[['Tom','Mary']]
pandas series element-wise
np.square(t) t[t<=7]
t.unique()=[unique items in t] t.sum() t.mean() t.max() t.min()
t.value_counts()
Creating a DataFrame: data = {'name':['a','b','c'], 'mark':[80,90,100]} a dictionary, frame = pd.DataFrame(data)
frame = pd.DataFrame(data, columns=['object','price'], index=['one','two',⋯])
frame = pd.DataFrame(np.arange(16).reshape((4,4)), index=[⋯], columns=[⋯])
frame = pd.DataFrame([[1,2,3],[4,5,6]], columns=['a','b','c'])
frame.columns 显示 Index([⋯],dtype='object')
frame.index 显示 RangeIndex(start=0, stop=5, step=1)
frame.values 显示 array([[]]二维数组显示每一行所有列的值, dtype=object)
frame.sum() 显示每一列所有值的和（字符串的和等于 append） frame.mean() 显示每一列的平均 frame.max() min() median()
frame.sort_values(by='price'一个列名)
frame.iloc[[0,1,2]] = frame.iloc[0:3] selecting by index ———loc selecting by label
pd.read_csv() _excel() _table() _clipboard()
df = pd.read_csv("name.csv", na_values=':')
df.info()
df.head()
df.tail()
df.any()
df.isnull().sum()

```
Amount         0
TranNo        44
Status        41
Department    14
Fiscal Year   55
Month         22
RedFlag        0
dtype: int64
```

df.dropna() # drop missing values
df.fillna(df.mean()) # replace missing value by the mean of its column
df=DataFrame({}) df.to_csv() _excel() _table()
pd.read_excel("⋯",na_values=':') replace missing values with NaN
上传文件到 colab,用 open()打开文件: from google.colab import files files.upload()
fin=open("") fin.close() 也可以直接 pd.read_csv()读上传过的文件
下载文件 files.download("路径"

Unsupervised Learning		Supervised Learning		Reinforcement Learning	
Clustering	Consumer segmentation	Classification	Fraud detection	-	Real-time decisions
	Recommendation System		Image classification		Robot Navigation
Density Estimation	-	Regression	Price prediction		Skill acquisition
			Weather forecast		Game AI

Supervised Learning: to learn a function that maps an input to an output based on examples of input + label (which is the correct output) pairs. The model is trained on a input+label dataset (build model that gives good prediction on the label for that inputs in the dataset). Then it can use this model to predict the outcome (ie the labels) of out-of-sample data.
Supervised Learning: Regression: refers to the general statictical method for determining the relationship between input x and output y, where x and y can be high-dimensional vectors

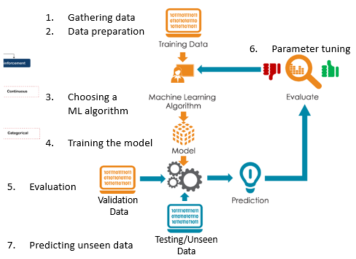
Find the line $y=mx+c$ that is closest to the n points $(x_1,y_1), (x_2,y_2), \cdots, (x_n,y_n)$
or more mathematically, determine m, c such that

$$\sum_{1\le i\le n} |y_i-(mx_i+c)|$$

is minimized.
To difficult for math. Easier for the following

$$\sqrt{\sum_{1\le i\le n} (y_i-(mx_i+c))^2}$$

Unsupervised Learning: Clustering: to train and construct a model from which we can determine a natural grouping (according to some distance metric) in data.
Reinforcement Learning: learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.



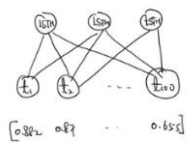
Split the data into three sets: training, validation, testing. Overfitting: the model fits perfectly for the training data, but gives poor results for other data. Repeat the training-validation cycles until get a model that give satisfactory results for both the training and validation dataset. Evaluate the model by testing dataset and the result gives much more reliable measure of the accuracy of the model in general.
Supervised learning: **Linear Regression** model for predicting ice cream sales:
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
x = np.array([10,12,16,20,24,26]).reshape((-1,1)) # reshape to 2D array
y = np.array([4.5.7.10.15.16])
model = LinearRegression()
model.fit(x,y) or model = LinearRegression().fix(x,y)
y_pred = model.predict(x)

```
model=Sequential()
model.add(LSTM(50,return_sequences=True,input_shape=(100,1)))
model.add(LSTM(50,return_sequences=True))
model.add(LSTM(50))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
```

```
model.summary()
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_30 (LSTM)               (None, 100, 50)           10400
_________________________________________________________________
lstm_31 (LSTM)               (None, 100, 50)           20200
_________________________________________________________________
lstm_32 (LSTM)               (None, 50)                20200
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 51
=================================================================
Total params: 50,851
Trainable params: 50,851
Non-trainable params: 0
```

Note that each of the LSTM cells will generate 100 different h's (one for each day). By setting return_sequence to True, all these 100 h's will be the output of that LSTM. Since we have 50 LSTM cells in this layer, the overall shape of this network output is (100, 50).



!pip install yfinance from

model.predict([[30]]) return array([18.84615385])

plt.figure(figsize=(10,6))
plt.scatter(x,y)
plt.scatter(x,y_pred, color='r')
x_seq = np.linspace(x.min(),x.max(),15).reshape((-1,1))
plt.plot(x_seq, model.predict(x_seq))
plt.show()

Supervised learning: **Classifying** Apple vs Pear
from sklearn import tree
import numpy as np
features = ['color','width','height']
fruitnames=['apple','pear']
data = np.array([[10,5.3,4,8], [⋯], [⋯], [⋯]])
labels = np.array([0,1,0,1,0,1,0,1])
fruit_classifier = tree.DecisionTreeClassifier()
fruit_classifier = fruit_classifier.fit(data, labels)
predict = fruit_classifier.predict(np.array([[101,6.3,7.9], [⋯], [⋯]]))
predict # 得到 array([1,0,0])
list(map(lambda i: fruitnames[i], predict)) 得到['pear', 'apple', 'apple']
print(fruitnames[int(fruit_classifier.predict([[101,6.3,7.9]]))]) # 得到 pear
tree.plot_tree(fruit_classifier, feature_names = features, class_names = fruit_names, filled = True) # 显示 decision tree

Unsupervised learning: **clustering** points on 2D plane
from pylab import plt
import numpy as np
import pandas as pd
plt.style.use('seeborn')

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
x,y = make_blobs(n_samples=100, centers=4, random_state=500, cluster_std=1.25)
model = KMeans(n_clusters=4, random_state = 0)
model.fit(x)
y_ = model.predict(x)

plt.figure(figsize = (10,6))
plt.scatter(x[:,0],x[:,1], c=y, cmap='coolwarm') 还能再 plot 一个 c=y_的图

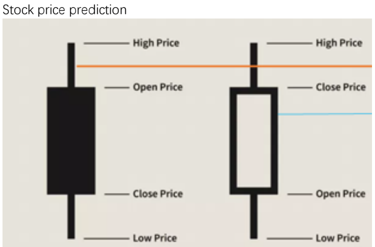
Deep Learning: Train a neural network for digit classification
import numpy as np
import matplotlib.pyplot as plt
from keras import models
from keras import layers
from keras.utils.np_utils import to_categorical
from keras.datasets import mnist -----the MNIST(Modified National Institute of Standards and Technology) dataset for training

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images.shape, train_labels.shape, test_images.shape, test_labels.shape

train_images = train_images.reshape((60000, 28*28))/255
test_images = test_images.reshape((60000, 28*28))/255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

network = models.Sequential()
network.add(layers.Dense(512, activation = 'relu', input_shape=(28*28,)))
network.add(layers.Dense(10, activation = 'softmax'))
network.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['accuracy'])
network.fit(train_images, train_labels, epochs=10, batch_size=128)
test_loss, test_acc = network.evaluate(test_images, test_labels)

Stock price prediction



import mplfinance as mpf

```
mpf.available_styles()
```

```
['binance',
 'blueskies',
 'brasil',
 'charles',
 'checkers',
 'classic',
 'default',
 'ibd',
 'kenan',
 'mike',
 'nightclouds',
 'sas',
 'starsandstripes',
 'yahoo']
```

pandas_datareader import data
import yfinance as yf
yf.pdr_override() -------override yfinance with pandas

mtr = data.get_data_yahoo(tickers="0066.HK", start="2020-01-01", end="2020-12-31")
apple = data.get_data_yahoo(tickers="AAPL", start="2020-01-01", end="2020-12-31")

visualization:
import matplotlib.pyplot as plt
mtr['20d']=np.round(mtr['Open'].rolling(20).mean(),2)
mtr['40d']=np.round(mtr['Open'].rolling(40).mean(),2)
plt.figure(figsize=(15,4))
plt.grid(True)
plt.plot(mtr[['Open','20d','40d']])
plt.show()

import mplfinance as mpf # candle stick chart
mpf.plot(mtr,type='candle',style='charles',figratio=(10,6),mav=(20,40), volume=True)
mpf.plot(mtr[:100],type='candle',style='charles',figratio=(10,6),mav=(20,40), volume=True) # 一百天

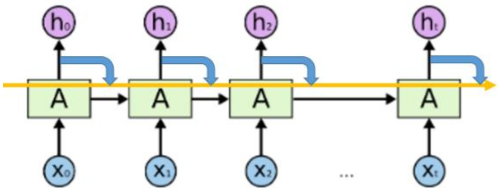
LinearRegression model for MTR stock price prediction
!pip install yfinance
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas_datareader import data
from datetime import datetime
import yfinance as yf
yf.pdr_override()

start = datetime.strptime('2010-01-01', '%Y-%m-%d')
end = datetime.strptime('2020-06-30', '%Y-%m-%d')
mtr = data.DataReader('MTR', start=start, end=end, data_source='yahoo')
today = mtr['Open'].iloc[:len(mrt)-1].reset_index(drop=True)
nextday = mtr['Open'].iloc[1:].reset_index(drop=True)

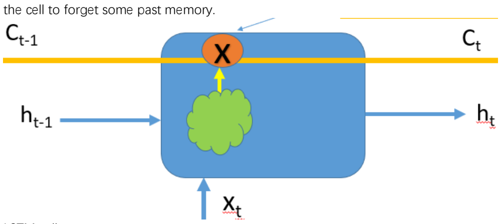
from sklearn.model_selection import train_test_split
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score

today = np.array(today).reshape(-1,1)
nextday = np.array(nextday)
X_train, X_test, y_train, y_test = train_test_split(today, nextday, test_size=0.2)
reg = linear_model.LinearRegression()
reg.fit(X_train, y_train)
y_pred = reg.predict(X_test)
res_msr = mean_squared_error(y_test, y_pred)
res_r2_score = r2_score(y_test, y_pred)

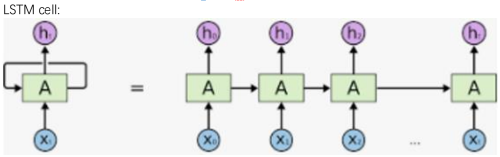
LSTM: Long Short-Term Memory Network, Capable of learning long-term dependencies, especially in sequence prediction problems. Application: speech recognition, machine translation, handwriting recognition, speech synthesis, music modelling, protein structure prediction
Long-Term: Conveyor belt: carry the output straight down to the entire chain, with only minor interactions



Short-Term: Gate optionally let the information flow in and out of the cell; i.e., enable the cell to forget some past memory.



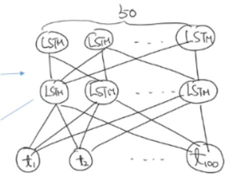
LSTM cell:



LSTM for HengSeng long short-term memory
!pip install yfinance
from pandas_datareader import data
from pylab import plt
import yfinance as yf

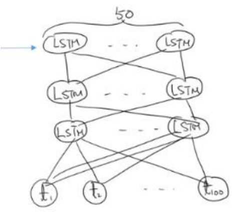
```
yf.pdr_override()
df = data.get_data_yahoo(tickers='^HIS',start='…',end='…')
df = df['Close']

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0,1))
rdata = scaler.fit_transform(np.array(df).reshape(-1,1))
training_size = int(len(rdata)*0.65)
train_period = rdata[0:training_size,:]
test_period = rdata[training_size:len(rdata),:]
from keras.preprocessing.sequence import TimeseriesGenerator
train = TimeseriesGenerator(train_period, train_period, length=100, batch_size=5000)
test = TimeseriesGenerator(test_period, test_period, length=100, batch_size=5000)
X_train, y_train = list(train)[0][0], list(train)[0][1]
X_test, y_test = list(test)[0][0], list(test)[0][1]

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape=(100,1)))
model.add(LSTM(50, return_sequences=True))
model.add(LSTM(50))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
model.summary()          # 第一页大图
```

第二层:



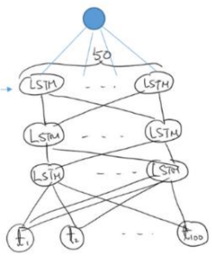
Since return_sequence=True, each of the 50 LSTM cells will ouput 100 h's, the shape of the network output is (100,50)

第三层:



Since return_requence is NOT set to True, each of the 50 LSTM will only output the last h. Thus, the output shape is (50,)

最后一层:




```
model.fit(X_train,y_train,validaton_data=(X_test,y_test),epochs=100,batch_size=64,verbose=1)
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)
train_predict = scaler.inverse_transform(train_predict)     # inverse-map(0,1) to true stock prices
test_predict = scaler.inverse_transform(test_predict)
```

 # 如果用过去 10 天数据 predict 未来第五天的数据
test_predict = test_predict[4:]
train_predict = train_predict[4:]
look_back = 14

```
look_back=100
trainPredictPlot=np.empty_like(rdata)          # create an array with same shape and type as data
trainPredictPlot[:,:]=np.nan          # initialize every entry of trainPredictPlot to NaN
trainPredictPlot[look_back:len(train_predict)+look_back,:] = train_predict
testPredictPlot=np.emptylike(rdata)
testPredictPlot[:,:]=np.nan
testPredictPlot[len(train_predict)+(look_back*2)+1:len(rdata)-1,:] = test_predict

#  循环遍历
res = 0
for i in range(len(test_predict)):
   res += abs(scaler.inverse_transform(rdata)[len(train_predict)+(look_back)*2+i] - test_predict[i])
print("Accuracy: ", res)
```

Reinforcement Learning

To train a system (an **agent**) to have some "desirable" behavior for some "task", we let the agent learn by doing the task step by step (even though initially the agent does not know what to do and may just give random actions).

After completing every single step, we (the **environment**) do the following:

reward the agent (by giving some points) if its action taken for this step is "desirable"

punish the agent (by taking away some points) if the action is not "desirable"

tell the agent the new "state" in the environment after its action.

The agent is running some algorithm such that depending on the reward/punishment it gets, it changes its behavior (by changing, for example, an action table, some decision rules, neural networks, …) hoping to make some better move when seeing similar situation later.

Environment: defines the problem at hand. can be a computer game to be played or a financial market to be traded in

State: subsumes all relevant parameters that describe the current state of the environment. In a computer game might be the whole screen with all its pixels. In a financial market might include current and historical price levels or financial indicators such as moving averages, macroeconomic variables and so on

Agent: subsumes all elements of the RL aogorithm that interacts with the environment and that learns from these interactions. In a gaming context might represent a player playing the game. In a financial context could represent a trader placing bets on rising or falling markets

Action: an agent can choose one action from a (limited) set of a allowed actions. In a computer game, movements to the left or right might be allowed actions, while in a financial market, going long or short could be admissible actions.

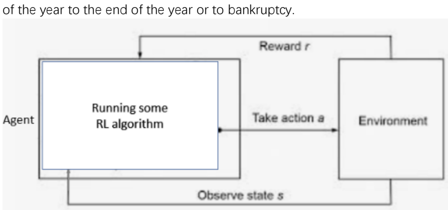
Step: given an action of an agent, the state of the environment is updated. One such update is generally called a step. The concept of a step is general enough to encompass both heterogeneous and homogeneous time intervals between two steps. While I computer games, real-time interaction with the game environment is simulated by rather short, homogeneous time intervals ("game clock"), a trading bot interacting with a financial market environment could take actions at longer, heterogeneous time intervals, for instance.

Reward: Depending on the action an agent chooses, a reward (or penalty) is awarded. For a computer game, points are a typical reward. In a financial context, profit (or loss) is a standard reward (or penalty)

Target: specifies what the agent tries to maximize. In a computer game, this in general is the score reached by the agent. For a financial trading bot, this might be the accumulated trading profit.

Policy: defines which action an agent takes given a certain state of the environment. Given a certain state of a computer game, represented by all the pixels that make up the current scene, the policy might specifu that the agent chooses "move right" as the action. A trading bot that observes three price increases in a row might decide, according to its policy, to short the market.

Episode: a set of steps from the initial state of the environment until success is achieved or failure is observed. In a game, this is from the start of the start of the game until a win or loss. In the financial world, for example, this is from the beginning of the year to the end of the year or to bankruptcy.



Problem: To illustrate how reinforcement learning work, we need to provide an "environment" before we can desigh, implement and test our RL algorithms, or compare different RL algorithms. Since we may test various RL algorithms hundreds or even thousands of times, it is not feasible to wait for the environment's real responses.

Q-Learning: Bellman's principle of optimality: an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

Idea: We try to learn the Q-value Q(S, A) for all possible states S, and actions A. Intuitively, Q(S, A) tells us how good it is if we take action A when we are current in state S. Initially, Q(S,A) is assigned an arbitrary number, say 0, and we will improve it during the learning process.

Suppose when at state S, we have chosen action A and the action moves us to S'.

Suppose there are k possible actions A1, A2, ... Ak. Then, after we move to S', the next action can be A1, A2,..., Ak. Which one?

By Bellman's principle of optimality, we will choose the one MA such that Q(S', A') = max Q(S', Ai)

Then, we need to update Q(S, A). How?

- Compare Q(S',A') and Q(S,A) as we have looked a step further.
  - If Q(S',A') > Q(S, A), then we need to increase Q(S,A) somewhat
  - If Q(S',A') < Q(S, A), we need to decrease Q(S,A) somewhat.

Intuitive reasoning:

- Q(S,A) = Q(S,A) + (Q(S',A') – Q(S, A))
- But this will deterministically set Q(S,A) to Q(S',A').

We need some variability, For example,

$$Q(S,A) = Q(S,A) + \alpha \cdot (\texttt{reward} + \gamma \cdot Q(S',A') - Q(S,A))$$

Here, α is called the *learning rate* and γ is called *discount factor.*

```
from sklearn.preprocessing import KBinsDiscretizer
import numpy as np
import gym
import time, math, random
from typing import Tuple
env = gym.make('CartPole-v1')
n_bins = (6, 12)
lower_bounds = [env.observation_space.low[2], -math.radians(50)]
upper_bounds = [env.observation_space.high[2], math.radians(50)]

def discretizer(_, __, angle, pole_velocity):
      est = KBinsDiscretizer(n_bins = n_bins, encode = 'ordinal', strategy='uniform')
      est.fit([lower_bounds, upper_bounds])
      return tuple(map(int, est.transform([[angle, pole_velocity]])[0]))
Q_table = np.zeros(n_bins + (env.action_space.n,))
def policy(state: tuple):
      return np.argmax(Q_table[state])
def new_Q_value(reward: float, state_new: tuple, discount_factor=1):
      future_optimal_value = np.max(Q_table[state_new])
      learned_value = reward + discount_factor * future_optimal_value
      return learned_value
def learning_rate(n:int, min_rate=0.01):
      return max(min_rate, min(1.0, 1.0-math.log10((n+1)/25)))
def exploration_rate(n:int, min_rate=0.1):
      return max(min_rate, min(1, 1.0-math.log10((n+1)/25)))
n_episodes = 250
for e in range(n_episodes):
      current_state, done = discretizer(*env.reset()), False
      while done == False:
            action = policy(current_state)
            if np.random.random() < exploration_rate(e):
                  action = env.action_space.sample()
            obs, reward, done, _ = env.step(action)
            new_state = discretizer(*obs)

            lr = learning_rate(e)
            learnt_value = new_Q_value(reward, new_state)
            old_value = Q_table[current_state][action]
            Q_table[current_state][action] = (1-lr)*old_value + lr*learnt_value

            current_state = new_state
            if e % 3 == 0:
                  env.render()
      time.sleep(0.5)
env.close()
```

gym-anytrading: gym environment simulation tool, collection of OpenAI Gym environments for reinforcement learning trading algorithms. provides three Gym environments, TradingEnv, ForexEnv, and StocksEnv, and facilitate many useful methods for developers to implement and test RL-based algorithms for trading in the FOREX and the Stock markets.

Environment Properties:

Trading Positions: Short (=0) and Long (=1) **Long** position wants to buy shares when prices are low and profit by sticking with them while their value is going up, and **Short** position wants to sell shares with high value and use this value to buy shares at a lower value, keeping the difference as profit.

Long position -> buy action

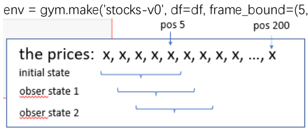
Short position -> sell action.

prices: "Real" prices over time. Used to calculate rewards and final profit.

window_size: Number of ticks (previous ticks + current tick) in an observation.

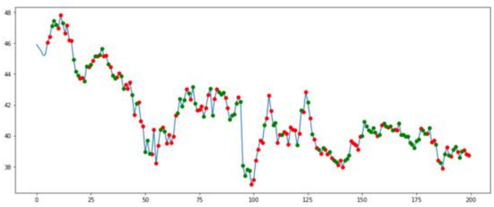
profit: The amount of units of currency achieved by starting with 1.0 unit (profit = FinalMoney/StartingMoney).

```
import gym
import gym_anytrading
import numpy as np
import pandas as pd
from pylab import plt
from pandas_datareader import data
from datetime import datetime
import yfinance as yf
yf.pdr_override()

start = datetime.strptime('2020-01-01','%Y-%m-%d')
end = datetime.strptime('2021-06-30','%Y-%m-%d')
df = data.DataReader('0066.HK', start=start, end=end, data_source='yahoo')
plt.figure(figsize=(15,6))
plt.plot(df['Close'])
plt.show()

env = gym.make('stocks-v0', df=df, frame_bound=(5,200), window_size=5)
```



Note: In this environment, the state is independent of the actions.

```
state = env.reset()
while True:
      action = env.action_space.sample()         # [1,0,1,0,1,0]
      obser, reward, done, info = env.step(action)
# reward: Presumbly, gym-anytrading has designed a way to calculate the rewards such that an agent that maximizes the toral rewards will get the max final profit.
      if done:
            print('info', info)
            break
plt.figure(figsize=(15,6))
env.render_all()
plt.show()
```



stable_baselines: library of RL model building algorithms, provides many state-of-the-art RL model building algorithms

A2C: advanced actor critic RL algorithm

```
import gym
import gym_anytrading
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import A2C
!pip install yfinance
from pandas_datareader import data
from datetime import datetime
import yfinance as yf
yf.pdr_override()

start = datetime.strptime('2020-01-01','%Y-%m-%d')
end = datetime.strptime('2021-06-30','%Y-%m-%d')
df = data.DataReader('MTR', start=start, end=end, data_source='yahoo')
env_maker = lambda: gym.make('stocks-v0', df=df, frame_bound=(5,100), window_size=5)
env = DummyVecEnv([env_maker])
model = A2C('MlpLstmPolicy', env, verbose=1)
model.learn(total_timesteps=2000)
env = gym.make('stocks-v0',df=df, frame_bound=(90,110), window_size=5)
# test the performance by focus of the stock prices from pos 90 to 110
obs = env.reset()
while True:
      obs = obs[np.newaxis, ...]
      action, _states = model.predict(obs)
      obs, rewards, done, info = env.step(action)
      if done:
            print("info", info)
            break
plt.figure(figsize=(15,6))
plt.cla()
env.render_all()
plt.show()
```

Sentiment Analysis

Approach: text input, tokenization, stop word filtering, negation handling, stemming, classification

Tokenization

['I', 'love', 'this', 'movie', 'it', 'sweet', ...

Stopwords filtering

- Stop words are a set of most commonly used words in a language. They usually do not convey very important information on the semantic of a sentence.
- Examples of stopwords: are, and, am, at, be, but, so, such, then, ...

Negation handling

- It is not bored: "not bored" is opposite of "bored".

Stemming

- Reduce words with same stem to a common form.
- Examples: wait, waits, waiting, waited ➔ wait

Classification:

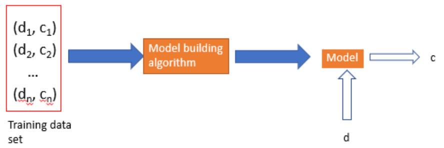
Input:

- A document: a long sequence of "words" after the previous preprocessing.
- A fixed set of classes C = { c_1, c_2, ..., c_n }.

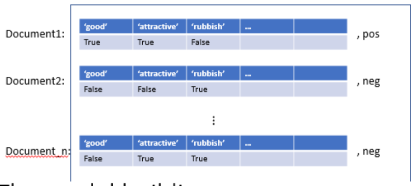
Output:

- A predicted class c ∈ C.

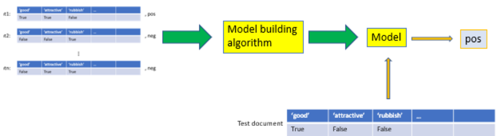
Can be treated as ML problem:



The training set for this supervised learning task:



The model building process

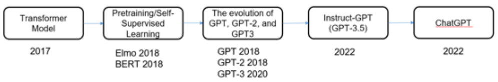


NLTK: a suite of libraries and Python programs for NLP for English

```
import nltk
from nltk.corpus import movie_reviews
import random
nltk.download('movie_reviews')
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
words_features = list(all_words)[:2000] # top2000 most used words in movie reviews
documents = [(list(movie_reviews.words(fileid)),category)
        for category in movie_reviews.categories()
        for fileid in movie_reviews.fileids(category)]   # a list of tuples
def document_features(document):
    document_words = set(document)
    features = {}
    for word in words_features:
        features[word] = (word in document_words)
    return features
random.shuffle(documents)
featuresets = [(document_features(d),c) for (d,c) in documents]
train_set, test_set = featuresets[100:], featuresets[:100]
model = nltk.NaiveBayesClassifier.train(train_set)
model.classify(test_set[1][0])
nltk.classify.accuracy(model, test_set)
model.show_most_informative_features(5)
```

```
Most Informative Features
          outstanding = True            pos : neg    =     13.2 : 1.0
                mulan = True            pos : neg    =      7.8 : 1.0
               seagal = True            neg : pos    =      7.7 : 1.0
          wonderfully = True            pos : neg    =      7.6 : 1.0
                damon = True            pos : neg    =      6.2 : 1.0
```

Transformer: a deep neural network architecture aiming to solve sequence-to-sequence tasks. -> ChatGPT



ChatGPT: Self-supervised learning: model train itself by learning one part of the input from another part of the input.

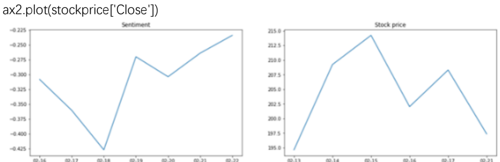
High-level idea: Given a training input, which sequence of tokens, we "randomly" mask off some of the tokens, and this create a new training data: the sequence with the mask token, and the label is the sequence of masked token.

Flair: a NLP tool supports SA

```
import pandas as pd
import requests
import flair
from google.colab import files
def get_data(tweet):
    data = {
        'id':tweet['id_str'],
        'created_at':tweet['created_at'],
        'text':tweet['full_text']
    }
    return data
params = {'q':'tesla', 'tweet_mode':'extended', 'lang':'en', 'count':'100'}
fin = open('BearerToken.txt','r')
BearToken = fin.readlines()[0].strip()
response = requests.get(
'https://api.twitter.com/1.1/search/tweets.json', params=params,
headers = {'authorization':'Bearer '+ BearToken})
tweets = pd.DataFrame()
# .json() convert the API response into a python dictionary through the JSON file format
# response.json() = a list of dictionaries, one for a tweet
for tweet in response.json()['statuses']:
    row = get_data(tweet)
    tweets = tweets.append(row, ignore_index=True)
sentiment_model = flair.models.TextClassifier.load('en-sentiment')
probs = []
sentiments = []
for tweet in tweets['text']:
        sentence = flair.data.Sentence(tweet)     # tokenization
        sentiment_model.predict(sentence)
        probs.append(sentence.labels[0].score)
        sentiments.append(sentence.labels[0].value)
tweets['probability'] = probs
tweets['sentiment'] = sentiments
```

relate sentiment with stock prices:

```
import requests
import pandas as pd
import re
from pylab import plt
whitespace = re.compile(r"\s+")
web_address = re.compile(r"(?i)https(s):\/\/[a-z0-9.~_\-\/]+")
tesla = re.compile(r"(?i)@Tesla(?=\b)")
user = re.compile(r"(?i)@[a-z0-9_]+")

def clean(tweet):
        tweet = whitespace.sub(' ', tweet)
        tweet = web_address.sub('', tweet)
        tweet = tesla.sub('Tesla', tweet)
        tweet = user.sub('', tweet)
        return tweet
BearerToken = "AAAAAAAAAAAAAAAAAAAAAAA"
api = 'https://api.twitter.com/2/tweets/search/recent'
headers = {'authorization': f'Bearer {BearerToken}'}
params = {
        'query': '(TESLA OR tsla OR Musk) (lang:en)',
        'max_results': '10',
        'tweet.fields':'created_at,lang'
}
from datetime import datetime, timedelta
dtformat = '%Y-%m-%dT%H:%M:%S.000Z'
def go_back(now, mins):
        return now - timedelta(minutes = mins)
df = pd.DataFrame()
now = datetime.now() - timedelta(days=1)
last_week = now - timedelta(days=10)
while True:
        if now <= last_week:
            break
        pre60 = go_back(now, 60)
        params['start_time'] = pre60.strftime(dtformat)
        params['end_time'] = now.strftime(dtformat)
        response = requests.get(api, params = params, headers =headers)
        tweetdict = response.json()
        if 'data' in tweetdict:
            for tweet in response.json()['data']:
                row = {'created_at':tweet['created_at'][5:10],
                       'text':clean(tweet['text'])}
                df = df.append(row, ignore_index=True)
        now = pre60
import flair
sentiment_model = flair.models.TextClassifier.load('en-sentiment')
wsentiments = []
for tweet in df['text']:
        sentence = flair.data.Sentence(tweet)
        sentiment_model.predict(sentence)
        prob = sentence.labels[0].score
        sentiment = 1 if sentence.labels[0].value == 'POSITIVE' else -1
        wsentiments.append(prob * sentiment)
df['wsentiments'] = wsentiments
df_sent = df.groupby('created_at')['wsentiments'].mean()
import yfinance as yf
yf.pdr_override()
from pandas_datareader import data
start = datetime.now() - timedelta(days=10)
end = datetime.now() - timedelta(days=1)
stockprice = data.get_data_yahoo('TSLA', start=start, end=end)
indexlist = stockprice.index.to_list()
for i in range(len(indexlist)):
    indexlist[i] = str(indexlist[i])[5:10]
stockprice.index = indexlist
fig = plt.figure(figsize=(18,5))
ax1 = fig.add_subplot(121)
ax1.plot(df_sent)
ax1.title.set_text('Sentiment')
ax2 = fig.add_subplot(122)
ax2.title.set_text('Stock price')
ax2.plot(stockprice['Close'])
```



Fraud detection and prevention: FDP

Challenge is to quickly identify and separate anomalous transactions from those legitimate, without impacting on customer experience

High accuracy, avoid overfitting

Money Laundering:

Detecting outliers:

Percentiles: For any 0<=p<=100, the pth-percentile of a set S of values is the value x_p in S such that

a fraction of p of the data values in S are smaller than or equal to x_p, and the remaining fraction (1-p) is greater than x_p.

For example, the median of S is the 50th percentile of S.

Quantiles:

The first quantile of S(Q_1)= the 25th percentile of S

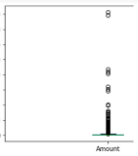
The second quantile of S(Q_2)= the median of S = the 50th percentile of S

The third quantile of S(Q_3)= the 75th percentile of S

5-number summary: xmin , Q1 , Q2 , Q3 , xmax

```
dfa = df['Amount']
dfa.min(), dfa.quantile(q=0.25), dfa.quantile(q=0.5), \
    dfa.quantile(q=0.75), dfa.max()
```

```
dfa.plot.box()
```



Formal definition of outliers

Based on the 5-number summary: xmin , Q1 , Q2 , Q3 , xmax

Define the interquartile range IQR to be the value Q3 – Q1

An outlier is the one with value either

<= (Q1 – 1.5 * IQR), or

>= (Q3 + 1.5 * IQR)

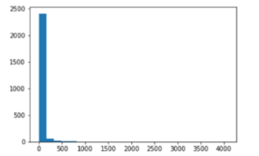
convert to numeric values:

```
d = {"Jan":1, "Feb":2, "Mar":3, "Apr":4, "May":5,
     "Jun":6, "Jul":7, "Aug":8,
    "Sep":9, "Oct":10, "Nov":11, "Dec":12}
df['Month'] = df['Month'].map(d)
df['Month'].head(10)
```

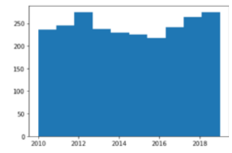
```
d = {"Paid_Reconciled":0, "Paid_UnReconciled":1}
df['Status'] = df['Status'].map(d)
df['Status'].head(10)

0    0.0
```

distributions of data:

```
pic = plt.hist(df['Amount'],bins=25)
```

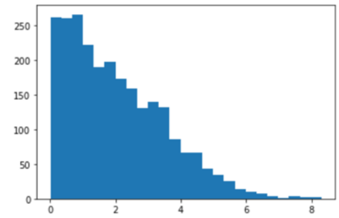


```
pic = plt.hist(df['Fiscal Year'], bins=10)
```



transformation of data: log-transformation

```
df['Amount'] = df['Amount'].apply(lambda x: np.log(x+1))
pic = plt.hist(df['Amount'], bins=25)
```



feature selection:

compute covariance between column and the output (i.e. the "RedFlag")

```
np.cov(df['Amount'].fillna(df['Amount'].mean()), df['RedFlag'])

array([[2.22031009, 0.31095493],
       [0.31095493, 0.11373685]])
```

X, Y

Cov(X, X) (aka Variance(X)

Cov(X, Y)

Cov(Y, X)

Cov(Y, Y)

Note: Cov(X,Y) = Cov(Y,X)

- x_i = a given x value in the data set
- x_m = the mean, or average, of the x values
- y_i = the y value in the data set that corresponds with x_i
- y_m = the mean, or average, of the y values
- n = the number of data points

Given this information, the formula for covariance is: Cov(x,y) = SUM [(x_i - x_m) * (y_i - y_m)] / (n - 1)

It is expected that in our DS, there are only a small fraction of redflag transactions. If we sample the whole DS normally, a major of the training data are non-redflag, and this makes the ML model favor prediction of non-redflag. Thus, in our training dataset, the number of redflag and non-redflag inputs should be more or less equal. How to do it? By oversamples. E.g. duplicate the redflag inputs in the DS to make the size non-redflag and redflag inputs more or less equal

But we have to do it carefully. There are many good methods: Navie Random over-sample, ROSE: Random Over-Sample Examples, SMOTE: Synthetic Minority, Oversample Technique, ADASYN: Adaptive Synthetic Method

```
from sklearn.preprocessing import MinMaxScaler
data = pd.DataFrame(df)
scaler = MinMaxScaler()
numerical = ['Amount', 'Month', 'Status']
data[numerical] = scaler.fit_transform(data[numerical])
from sklearn.model_selection import train_test_split
data = data.dropna()
X_train, X_test, y_train, y_test = train_test_split(data[['Amount', 'Month', 'Status']],
            data['RedFlag'], test_size=0.2, random_state=0)
print(f"Training set has {X_train.shape[0]} samples")
print(f"Testing set has {X_test.shape[0]} samples")
```

```
Training set has 1863 samples
Testing set has 466 samples
```

```
summary = pd.DataFrame(columns=['Learner', 'Train Time', 'Pred Time', 'Acc score', 'F1 score',
                                'Precision', 'Recall'])

from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, f1_score, precision_score, \
        recall_score, classification_report, confusion_matrix

NB = GaussianNB()
start = time()
NB.fit(X_train, y_train)
mid = time()
pred = NB.predict(X_test)
end = time()

pred_res = pd.DataFrame(pred)
pred_res = pred_res.set_index(y_test.index)

summary = summary.append({'Learner':'GaussianNB','Train Time':mid-start, 'Pred Time':end-mid,
                'Acc score':accuracy_score(y_test, pred),
                'F1 score':f1_score(y_test,pred,average='macro'),
                'Precision':precision_score(y_test,pred,average='macro'),
                'Recall': recall_score(y_test,pred,average='macro')}, ignore_index=True)
summary
```

	Learner	Train Time	Pred Time	Acc score	F1 score	Precision	Recall
0	GaussianNB	0.003991	0.001993	0.959227	0.915621	0.935524	0.89824

Given a transaction, we say that it is a

- true positive: ML predicts positive, and it is indeed redflag in our DS
- true negative: ML predicts negative, and it is indeed no-redflag
- false positive: ML predicts positive, but it is no-redflag
- false negative: ML predicts negative, but it is redflag

Let tp, tn, fp and fn to be the total number of transactions in the training data that are true positive, true negaive, false positive and false negative, respectively. Then

$$\text{precision} = \frac{tp}{tp+fp}, \text{ and}$$

$$\text{recall} = \frac{tp}{tp+fn}$$

$F1$-score is the *harmonic mean* of precision and recall, i..e,

$$F1\text{-score} = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}}$$

F1-score is better than precision when there is a large class imbalance

```python
from sklearn.linear_model import LogisticRegression

LR = LogisticRegression(random_state=0)
start = time()
LR.fit(X_train, y_train)
mid = time()
pred = LR.predict(X_test)
end = time()

summary = summary.append({'Learner':'LogisticReg','Train Time':mid-start, 'Pred Time':end-
                'Acc score':accuracy_score(y_test, pred),
                'F1 score':f1_score(y_test,pred,average='macro'),
                'Precision':precision_score(y_test,pred,average='macro'),
                'Recall': recall_score(y_test,pred,average='macro')}, ignore_index=True)
summary
```

	Learner	Train Time	Pred Time	Acc score	F1 score	Precision	Recall
0	GaussianNB	0.002990	0.002992	0.959227	0.915621	0.935524	0.89824
1	LogisticReg	0.013963	0.001996	0.933476	0.846246	0.924379	0.79931

```python
from sklearn.tree import DecisionTreeClassifier

DT = DecisionTreeClassifier(max_features=0.2, max_depth=2,
                            min_samples_split=2, random_state=0)
start = time()
DT.fit(X_train, y_train)
mid = time()
pred = DT.predict(X_test)
end = time()
summary = summary.append({'Learner':'DecisionTree','Train Time':mid-start, 'Pred Time':end-mi
               'Acc score':accuracy_score(y_test, pred),
               'F1 score':f1_score(y_test,pred,average='macro'),
               'Precision':precision_score(y_test,pred,average='macro'),
               'Recall': recall_score(y_test,pred,average='macro')}, ignore_index=True)
summary
```

	Learner	Train Time	Pred Time	Acc score	F1 score	Precision	Recall
0	GaussianNB	0.003990	0.001997	0.959227	0.915621	0.935524	0.898240
1	LogisticReg	0.012967	0.001997	0.933476	0.846246	0.924379	0.799310
2	DecisionTree	0.002992	0.003001	0.869099	0.568224	0.933406	0.557971

```python
from sklearn.ensemble import RandomForestClassifier

RF = RandomForestClassifier(max_depth=2)
start = time()
RF.fit(X_train, y_train)
mid = time()
pred = RF.predict(X_test)
end = time()

summary = summary.append({'Learner':'RandomForest','Train Time':mid-start, 'Pred Time':end-mi
               'Acc score':accuracy_score(y_test, pred),
               'F1 score':f1_score(y_test,pred,average='macro'),
               'Precision':precision_score(y_test,pred,average='macro'),
               'Recall': recall_score(y_test,pred,average='macro')}, ignore_index=True)
summary
```

	Learner	Train Time	Pred Time	Acc score	F1 score	Precision	Recall
0	GaussianNB	0.003990	0.001997	0.959227	0.915621	0.935524	0.898240
1	LogisticReg	0.012967	0.001997	0.933476	0.846246	0.924379	0.799310
2	DecisionTree	0.002992	0.003001	0.869099	0.568224	0.933406	0.557971
3	RandomForest	0.144354	0.011113	0.976395	0.949861	0.986520	0.920290

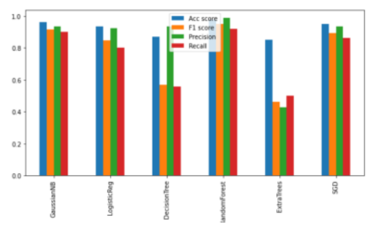
```python
from sklearn.ensemble import ExtraTreesClassifier

ET = ExtraTreesClassifier(max_depth=2)
start = time()
ET.fit(X_train, y_train)
mid = time()
pred = ET.predict(X_test)
end = time()

summary = summary.append({'Learner':'ExtraTrees','Train Time':mid-start, 'Pred Time':end
               'Acc score':accuracy_score(y_test, pred),
               'F1 score':f1_score(y_test,pred,average='macro'),
               'Precision':precision_score(y_test,pred,average='macro'),
               'Recall': recall_score(y_test,pred,average='macro')}, ignore_index=True)
summary
```

	Learner	Train Time	Pred Time	Acc score	F1 score	Precision	Recall
0	GaussianNB	0.003990	0.001997	0.959227	0.915621	0.935524	0.898240
1	LogisticReg	0.012967	0.001997	0.933476	0.846246	0.924379	0.799310
2	DecisionTree	0.002992	0.003001	0.869099	0.568224	0.933406	0.557971
3	RandomForest	0.144354	0.011113	0.976395	0.949861	0.986520	0.920290
4	ExtraTrees	0.094443	0.011971	0.856223	0.489261	0.927802	0.514493

```python
from sklearn.linear_model import SGDClassifier

SGD = SGDClassifier(loss='hinge', penalty="l2")
start = time()
SGD.fit(X_train, y_train)
mid = time()
pred = SGD.predict(X_test)
end = time()

summary = summary.append({'Learner':'SGD','Train Time':mid-start, 'Pred Time':end-mi
               'Acc score':accuracy_score(y_test, pred),
               'F1 score':f1_score(y_test,pred,average='macro'),
               'Precision':precision_score(y_test,pred,average='macro'),
               'Recall': recall_score(y_test,pred,average='macro')}, ignore_index=Tru
summary
```

	Learner	Train Time	Pred Time	Acc score	F1 score	Precision	Recall
0	GaussianNB	0.004987	0.001994	0.959227	0.915621	0.935524	0.898240
1	LogisticReg	0.010970	0.001995	0.933476	0.846246	0.924379	0.799310
2	DecisionTree	0.004987	0.002994	0.869099	0.568224	0.933406	0.557971
3	RandomForest	0.148247	0.012256	0.976395	0.949861	0.986520	0.920290
4	ExtraTrees	0.096461	0.011185	0.851931	0.460023	0.425966	0.500000
5	SGD	0.005951	0.001030	0.957082	0.914942	0.914942	0.914942

```python
summary[['Learner','Acc score','F1 score','Precision','Recall']].plot(kind='bar', x='Learner', figsize=(1(
<AxesSubplot:xlabel='Learner'>
```



Option pricing
Options are financial instruments that are based on the values of underlying tradable financial assets such as stocks.
An option contract offers the buyer the opportunity to buy/sell the underlying asset (we focus on stock here).
Each option contract will specify a date called expiry date, and a stated price called strike price.
Each option contract comes with a **premium**. When the contract is signed, the contract buyer needs to pay this premium and the contract seller receives it. Option pricing is the process of determining the amount of this premium.
There are two types of options.
Call option: allow the holder to buy the stock at the strike price on the expiry date†.
Put option: allow the holder to sell the stock at the strike price on the expiry date.
Buyer has the right not to exercise the contract, and in such case, he loses nothing, except for the premium he paid when he bought the option.
Seller has the obligation to execute the contract if the buyer decides to do so.
For call options buyers: they spend premium, and their losses are limited to premium spent. And the profits they can make do not have bounds.
For put options buyers: Again, losses are limited to the premium spent. But the profits they can make is between 0 and the strike price.

Objective of option pricing: The premium must be fair so that the buyer and seller will have an equal chance to win (i.e., gain profit).
Risk-free rate (RFR): It is the theoretical rate on an investment with zero risk. It can be the bank rate, or the government bond yields.
We now determine the stock price on the expiry date with the assumption that the stock price is only determined by RFR r.
For example, if r=0, then the stock price does not change over time, and we return to the simplest case.
Black-Scholes-Merton:
Compare $P = S - Ke^{-rT}$ with black-scholes-merton

$$C = S \times N(d_1) - Ke^{-rT} \times N(d_2)$$

We get black-scholes-merton from $P = S - Ke^{-rT}$ by considering further the situation that the stock prices fluctuate according to some probability distribution.

$$C = S \times N(d_1) - Ke^{-rT} \times N(d_2)$$

$$\text{where } d_1 = \frac{\ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}},$$

$$\text{and } d_2 = d_1 - \sigma\sqrt{T}.$$

- Stock Price $S = \$62$.
- Strik Price $K = \$60$.
- Time to Expiration $T - t = 40$ days, which is a fraction of $40/365 = 0.10959$ of a year.
- Volatility $\sigma = 32\% = 0.32$.
- Risk-Free rate $r = 4\% = 0.04$ (daily compounding).

Pluggin in the Black-Scholes equation, we get

$$d_1 = \frac{\ln(62/60) + [0.04 + 0.5(0.32)^2](40/365)}{0.32\sqrt{\frac{40}{365}}} = 0.40$$

$$d_2 = 0.404 - 0.32\sqrt{40/365} = 0.30$$

$$N(0.40) = 0.6554, \quad N(0.30) = 0.6179$$

Now the price of the option is

$$C = (62)(0.6554) - [60/e^{0.04(40/365)}](0.6179) = \$3.72.$$

Option pricing by ML
Basically, it is a problem supervising learning for regression, i.e., train a model (e.g., polynomial) to approximate the relation between the input and output.
Recent researches on this problem focus on designing and training neural network models that beat the Black-Scholes-Merton model.
Based on the outputs, these researches can be characterized in three categories:
1.directly predict the option premium
2.predict the volatility of the stock, and then use it as an input to Block-Scholes-Merton to determine the option premium
3."guess" the ratio between the option premium and strike price.

MLP1 model:
Dataset: A collection of million 12 million examples of roughly half calls and half puts.
To train the MLP1 model, 98% of the data is used as a training set, 1% for validation set during training, and 1% for testing.

option premim = (best_bid + best_offer)/2
T r S σ
he input
K

	strike_price	best_bid	best_offer	date_ndiff	treasury_rate	closing_price	sigma_20
0	600.0	28.000	29.000	47	5.17	624.22	0.007761
10	475.0	152.875	153.875	145	5.12	624.22	0.007761

sigma: standard deviation of the stock prices of the last 20 days

```python
from keras.models import Sequential
from keras.layers import Dense, Activation, LeakyReLU, BatchNormalization
from keras import backend
from keras.callbacks import TensorBoard
from keras.optimizers import Adam
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

```
Using TensorFlow backend.
```

```python
# Hyperparams
n_units = 400
layers = 4
n_batch = 4096
n_epochs = 10
```

```python
df = pd.read_csv('../options-df-sigma.csv')
df = df.dropna(axis=0)
df = df.drop(columns=['date', 'exdate', 'impl_volatility', 'volume', 'open_interest'])
df.strike_price = df.strike_price / 1000
call_df = df[df.cp_flag == 'C'].drop(['cp_flag'], axis=1)
put_df = df[df.cp_flag == 'P'].drop(['cp_flag'], axis=1)
```

```python
call_df.head()
```

	strike_price	best_bid	best_offer	date_ndiff	treasury_rate	closing_price	sigma_20
0	600.0	28.000	29.000	47	5.17	624.22	0.007761
10	475.0	152.875	153.875	145	5.12	624.22	0.007761
13	600.0	52.375	53.375	327	5.05	624.22	0.007761
17	610.0	20.000	20.750	47	5.17	624.22	0.007761
18	675.0	34.000	35.000	691	5.10	624.22	0.007761

call_x_train, call_x_test, call_y_train, call_y_test = train_test_split(call_df.drop(['best_bid', 'best_offer'], axis=1), (call_df.best_bid + call_df.best_offer) / 2, test_size=0.01, random_state=42)

```python
model = Sequential()
model.add(Dense(n_units, input_dim=call_X_train.shape[1]))
model.add(LeakyReLU())

for _ in range(layers - 1):
    model.add(Dense(n_units))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

model.add(Dense(1, activation='relu'))

model.compile(loss='mse', optimizer=Adam())
```

```python
history = model.fit(call_X_train, call_y_train,
                    batch_size=n_batch, epochs=n_epochs,
                    validation_split = 0.01,
                    callbacks=[TensorBoard()],
                    verbose=1)
```

```python
call_y_pred = model.predict(call_X_test)
```

```python
diff = (call_y_test.values - call_y_pred.reshape(call_y_pred.shape[0]))
```

```python
np.mean(np.square(diff))
```

To design a system that **applies machine learning techniques to predict fraud transactions** based on the file tranrecords.cvs, the following steps can be taken:

1. **Data Cleaning**: The first step is to clean up the data by removing any unnecessary columns, checking for missing values, and replacing them with appropriate values. In this case, the missing values are represented by the character ":" which can be replaced with NaN values using pandas library in Python.

python
import pandas as pd
import numpy as np
data = pd.read_csv('tranrecords.csv', delimiter=',', na_values=[':'])

2. **Exploratory Data Analysis (EDA)**: The next step is to perform exploratory data analysis to pick important columns for training. EDA involves understanding the data distribution, checking for outliers, and identifying relationships between variables.

python
data.describe() # summary statistics of the numerical columns
data.corr() # correlation matrix of the numerical columns

3. **Feature Engineering**: Feature engineering involves creating new features from existing ones that may improve the performance of the model. In this case, we can create a new feature that combines the amount and department columns to capture the spending behavior of each department.

python
data['Amount_Department'] = data['Amount'] * data['Department']

4. **Model Selection**: The next step is to select a suitable machine learning model to train and test the data. In this case, we can use a binary classification model such as logistic regression, decision tree, or random forest.

python
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier() # initialize the model

5. **Data Preparation**: The next step is to prepare the data for training and testing. This involves splitting the data into training and testing sets and scaling the numerical features.

python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data.drop(columns=['RedFlag']), data['RedFlag'], test_size=0.2)

# scale the numerical features
scaler = StandardScaler()
X_train[['Amount', 'Fiscal Year']] = scaler.fit_transform(X_train[['Amount', 'Fiscal Year']])
X_test[['Amount', 'Fiscal Year']] = scaler.transform(X_test[['Amount', 'Fiscal Year']])

6. **Model Training and Testing**: The next step is to train the model using the training data and test its performance using the testing data.

python
model.fit(X_train, y_train) # train the model
y_pred = model.predict(X_test) # predict the RedFlag for the testing data

7. **Model Evaluation**: The final step is to evaluate the performance of the model using evaluation metrics such as accuracy, precision, recall, and F1-score.

python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1-score:', f1)

With these steps, we can build a machine learning system that predicts fraud transactions based on the file tranrecords.cvs.

To find the values of f(1), f(2),..., f(n) efficiently using **Bellman's** principle of optimality, we need to determine how to express f(j) in terms of other appropriate f(i)'s.
We can express f(j) in terms of f(j-1) as follows:
f(j) = max(prod(ai, ai+1, ..., aj), f(j-1) * aj, aj)
The first term prod(ai, ai+1, ..., aj) represents the maximum product of a contiguous subsequence that ends at aj. The second term f(j-1) * aj represents the maximum product of a contiguous subsequence that ends at aj-1 and includes aj. The third term aj represents the product of a single number, which is also a contiguous subsequence that ends at aj.
We can use dynamic programming to calculate the values of f(j) efficiently. We start by initializing f(1) to be a1. Then, for each j from 2 to n, we calculate f(j) using the above formula. The maximum of all f(j)'s is the solution to the problem.

Here's a Python program that implements the above approach:

def max_product_subsequence(nums):
    n = len(nums)

```
    f = [0] * n
    f[0] = nums[0]
    res = f[0]
    for i in range(1, n):
        f[i] = max(nums[i], f[i-1] * nums[i], nums[i-1] * nums[i])
        res = max(res, f[i])
    return res

# example usage
nums = [0.5, 1.5, 30, 10, 5, 0.4, 10]
print(max_product_subsequence(nums)) # output: 4500.0
```

This program takes a list of positive numbers as input and returns the maximum product of a contiguous subsequence. The time complexity of this program is O(n), which is efficient.

```
!pip install gym


import numpy as np
import pandas as pd
import random
from pylab import plt, mpl
import time


import gym
env = gym.make('CartPole-v0')


env.seed(100)
env.action_space.seed(100)
random.seed(100)
np.random.seed(100)

def run_one_episode(env):
  state = env.reset()
  for step in range(200):
    a = random.randint(0,1)
    state, reward, done, info = env.step(a)
    if done:
      break
    #env.render()
    #time.sleep(0.05)
    print(f'step={step:2d} | state={state} | action={a} | reward={reward}')
  return step
```

To **improve the performance of the program by using Q-learning**, we need to create a Q-table that stores the expected rewards for each state-action pair. Then, we can use the Q-table to choose the best action at each step, based on the maximum expected reward.
Here's the modified program that uses Q-learning:

```
import numpy as np
import gym

env = gym.make('CartPole-v0')
env.seed(108)
np.random.seed(160)

# Q-learning parameters
alpha = 0.4
gamma = 0.999
epsilon = 0.1

# initialize Q-table
n_states = 4
n_actions = 2
q_table = np.zeros((n_states, n_actions))

def choose_action(state, q_table):
    # choose action based on epsilon-greedy policy
    if np.random.uniform(0, 1) < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(q_table[state])
    return action

def update_q_table(state, action, reward, next_state, q_table):
    # update Q-table using Q-learning algorithm
    max_q = np.max(q_table[next_state])
    q_table[state, action] += alpha * (reward + gamma * max_q - q_table[state,
action])

def run_one_episode(env, q_table):
    state = env.reset()
    total_reward = 0
    for step in range(200):
        action = choose_action(state, q_table)
        next_state, reward, done, _ = env.step(action)
        update_q_table(state, action, reward, next_state, q_table)
        state = next_state
        total_reward += reward
        if done:
            break
    return total_reward

# train the Q-learning agent
for i in range(1000):
    total_reward = run_one_episode(env, q_table)
    if i % 100 == 0:
        print(f'Episode {i}: total reward = {total_reward}')

# test the Q-learning agent
total_reward = 0
state = env.reset()
for step in range(200):
    action = np.argmax(q_table[state])
    state, reward, done, _ = env.step(action)
    total_reward += reward
    if done:
        break
print(f'Total reward = {total_reward}')
```

This program first initializes the Q-table with zeros and then trains the agent using Q-learning algorithm. In each episode, the agent chooses actions based on an epsilon-greedy policy and updates the Q-table using the Q-learning algorithm. After training, the program tests the agent by choosing actions based on the maximum expected reward from the Q-table. The performance of the agent should be much better than the previous random agent.

One simple and effective idea to improve the performance of the program is to use a simple heuristic to determine the next action based on the current state. For example, we can use the following heuristic:

- If the pole is leaning to the left, move the cart to the left.
- If the pole is leaning to the right, move the cart to the right.
- Otherwise, choose a random action.

To implement this idea, we can modify the existing program as follows:

```
import gym
import time
import random
import numpy as np

env = gym.make('CartPole-v1')
env.seed(100)
env.action_space.seed(1)
random.seed(100)
np.random.seed(100)

def run_one_episode(env):
    state = env.reset()
    for step in range(200):
        angle = state[2]
        if angle < 0:
            a = 0 # move cart to left
        elif angle > 0:
            a = 1 # move cart to right
        else:
            a = random.randint(0, 1) # choose random action
        state, reward, done, info = env.step(a)
        if done:
            print(f'Episode finished after {step+1} steps')
            time.sleep(1.5)
            break
        print(f'step={step+1:2d} | state={state} | action={a} | reward={reward}')
    return step+1

if __name__ == '__main__':
    num_episodes = 4
    for i in range(num_episodes):
        print(f'Running episode {i+1}...')
        num_steps = run_one_episode(env)
        if num_steps >= 206:
            print(f'SUCCESS after {num_steps} steps!')
            break
    else:
        print(f'FAILED after {num_steps} steps.')

    env.close()
```

In this modified program, we use the `angle` variable to store the pole angle of the current state. Then, we use the heuristic described above to determine the next action based on the value of `angle`. If the pole is leaning to the left, we move the cart to the left (`a = 0`), and if the pole is leaning to the right, we move the cart to the right (`a = 1`). Otherwise, we choose a random action (`a = random.randint(0, 1)`).

The rest of the program is similar to the original program. We run four episodes, and if any episode lasts for 206 steps or more, we consider it a success and print a success message. If all four episodes fail, we print a failure message. Finally, we close the environment using `env.close()`.

a) Relu: max(0, ...)

$x_1 = 0.2\times1 + 0.8\times0 + 0.7\times1 = 0.9$

Relu(1) = $\max(0, x_1) = 0.9$

$x_2 = 0.3\times1 + 1.2\times0 + 4.5\times1 = 4.8$

Relu(2) = $\max(0, x_2) = 4.8$

$y = \max(1.7\times0.9,\ 6\times4.8) = \boxed{28.8}$

b) Let $x = [-3, -2, -1, 0, 1, 2, 3]$

$y = [3, 2, 1, 0, 1, 2, 3]$

i) we can see that x and y are closely related

$y = |x|$

ii) $\bar{x} = 0$, $\bar{y} = 12/7$

$$\mathrm{cov}(x,y) = (-3)(3-\tfrac{12}{7}) + (-2)(2-12/7) + (-1)(1-\tfrac{12}{7}) + (0)(0-\tfrac{12}{7}) + (1)(1-\tfrac{12}{7}) + (2)(2-\tfrac{12}{7}) + (3)(3-\tfrac{12}{7})$$

$$= \boxed{0}$$

To show that the intuition that the covariance measures the relationship between two sequences is not always true, we can construct two sequences x and y such that (i) the two sequences are closely related, and (ii) the covariance of the two sequences is zero. Consider the following two sequences:
x = [-3, -2, -1, 0, 1, 2, 3, ...]
y = [3, 2, 1, 0, -1, -2, -3, ...]
These two sequences are closely related because given a value of x, we can determine the corresponding value of y correctly by taking the negative of x. For example, if x = 2, then y = -2.
However, the covariance of the two sequences is zero because the positive and negative deviations from the means of x and y cancel each other out. Specifically, we have:
xmean = (1 - 3)/2 = -1
ymean = (1 - 3)/2 = -1
n = infinity (since the sequences go on forever)

Covariance = [(x1 - xmean)(y1 - ymean) + (x2 - xmean)(y2 - ymean) + ...]/n
= [(-3 - (-1))(3 - (-1)) + (-2 - (-1))(2 - (-1)) + ...]/infinity
= (-4)(4) + (-1)(3) + (-2)(2) + (-1)(1) + ...]/infinity
= -infinity/infinity
= 0

Therefore, we have shown that the covariance of two sequences can be zero even if the two sequences are closely related, and thus, the intuition that the covariance measures the relationship between two sequences is not always true.

Here's a Python program that computes the average sentiment scores of tweets related to "Lenovo" and "Dell" for each day between May 7, 2023, and May 16, 2023:

```
import tweepy
from flair.models import TextClassifier
from flair.data import Sentence
from datetime import datetime, timedelta

# read bearer token from file
with open('BearerToken.txt', 'r') as f:
    bearer_token = f.read().strip()

# authenticate with Twitter API
auth = tweepy.AppAuthHandler(consumer_key, consumer_secret)
api = tweepy.API(auth, wait_on_rate_limit=True, wait_on_rate_limit_notify=True)

# initialize sentiment classifier
classifier = TextClassifier.load('en-sentiment')

# define query keywords
keywords = ['lenovo', 'dell']

# define start and end dates
start_date = datetime(2023, 5, 7)
end_date = datetime(2023, 5, 16)

# initialize results dictionary
results = {}

# loop over days between start and end dates
current_date = start_date
while current_date <= end_date:
    # initialize scores for each keyword
    scores = {keyword: [] for keyword in keywords}

    # search for tweets containing each keyword
    for keyword in keywords:
        query = f'{keyword} since:{current_date.strftime("%Y-%m-%d")}
until:{(current_date + timedelta(days=1)).strftime("%Y-%m-%d")}'
        tweets = tweepy.Cursor(api.search, q=query, lang='en',
tweet_mode='extended').items()

        # classify sentiment of each tweet and append score to list
        for tweet in tweets:
            text = tweet.full_text
            sentence = Sentence(text)
            classifier.predict(sentence)
            scores[keyword].append(sentence.labels[0].score)

    # compute average score for each keyword
    avg_scores = {keyword: sum(scores[keyword])/len(scores[keyword]) if
len(scores[keyword]) > 0 else 0 for keyword in keywords}
    results[current_date.strftime('%Y-%m-%d')] = avg_scores

    # move to next day
    current_date += timedelta(days=1)

# print results
for date, scores in results.items():
    print(f'{date}: Lenovo={scores["lenovo"]:.2f}, Dell={scores["dell"]:.2f}')
```

This program uses the Tweepy library to authenticate with the Twitter API and search for tweets containing the keywords "lenovo" and "dell" for each day between May 7, 2023, and May 16, 2023. It then uses the Flair library to classify the sentiment of each tweet and computes the average sentiment score for each keyword for each day. The results are stored in a dictionary and printed at the end of the program.

```
import numpy as np

# Input r, c, and s
r = int(input("Enter the number of rows (r): "))
c = int(input("Enter the number of columns (c): "))
s = int(input("Enter the starting integer (s): "))

# Construct array A
A = np.arange(s, s + r*c).reshape(r, c)

# Construct array B
B = np.sqrt(A)

# Print arrays A and B
print("Array A:")
print(A)
print("\nArray B:")
print(B)
```

`teams_wins.groups` gives

```
{'CU': [3, 4, 5], 'HKU': [0, 1, 2], 'UST': [6, 7, 8]}
```

`teams_wins.sum()` gives

```
team
CU     87
HKU    90
UST    57
Name: wins, dtype: int64
```

```
team
HKU    90
Name: wins, dtype: int64
```

```
# Create and assign the following DataFrame to the variable `table`:
data = {
    'year': [2018, 2019, 2020, 2018, 2019, 2020, 2018, 2019, 2020],
    'team': ['HKU', 'HKU', 'HKU', 'CU', 'CU', 'CU', 'UST', 'UST', 'UST'],
    'wins': [30, 28, 32, 29, 32, 26, 21, 17, 19],
    'draws': [6, 7, 4, 5, 4, 7, 8, 10, 8],
    'losses': [2, 3, 2, 4, 2, 5, 9, 11, 11]
}
# Create the DataFrame
table = pd.DataFrame(data)
# (c) Apply the groupby method to `table` to get the data structure `teams_wins`,
# which groups the wins in terms of teams.
teams_wins = table.groupby('team')['wins']
# (d) Use `teams_wins` to write a single Python statement that gives the team with the
largest number of wins.
team_with_most_wins = teams_wins.sum().idxmax()
# Display the created table and the team with most wins
table, team_with_most_wins
```

General observations and potential conclusions based on common trends and characteristics of ML models:

1. Performance Comparison: By comparing the Mean Squared Error (MSE) values on the graphs, we can determine which model performs better in terms of accuracy. A lower MSE indicates better performance, so the model with the lowest MSE is likely the most accurate.

2. Overfitting: If any of the models consistently show a decreasing MSE on the training data but an increasing MSE on the validation or test data, it suggests overfitting. Overfitting occurs when a model becomes too specialized to the training data and fails to generalize well to unseen data.

3. Model Complexity: If the LSTM model consistently outperforms the MLP1 and MLP2 models, it suggests that the temporal dependencies captured by the LSTM's recurrent connections are valuable for option pricing. This indicates that the LSTM model's ability to retain and utilize historical information is advantageous in this context.

4. Training Time: Comparing the training times of the models can provide insights into their computational efficiency. If one model consistently requires significantly more time to train than the others, it may indicate that the model is more complex or computationally intensive.

5. Convergence Rate: Comparing the rate at which the MSE decreases for each model can indicate how quickly they converge to an optimal solution. A model that converges faster may be more efficient and require fewer training iterations to achieve a certain level of performance.

6. Generalization: If the models consistently demonstrate similar MSE values on both the training and validation/test datasets, it suggests that they generalize well and are not overfitting or underfitting. This indicates that the models have learned the underlying patterns in the data and can make accurate predictions on unseen data.

7. Robustness: Examining the stability of the MSE values across different data samples or time periods can provide insights into the robustness of the models. If the models consistently perform well across various samples or time periods, it suggests that they are robust and can handle different market conditions or variations in the data.