Abstract:

In this part 1 of report, we will try to identify the stock market regimes by analyzing VIX data from 1990 to 2019. To achieve this, we construct a Gaussian mixture distribution based on monthly log-returns of VIX and employ Genetic Algorithm to find the most appropriate means and variance for each distribution.In part 2, an analyzation on the factors causing VIX to change collected by EMV tracker is reported using various regression models.

Introduction:

The VIX, as known as CBOE volatility index, measures the market's expect volatility implied by S&P500 options. The past VIX data may tell us what are the underlying regimes of the stock market and provide an insight of how these regimes changing from one to others. Based on the regimes we found, we are curious to see whether we can infer future stock market behaviors based on the current stock market.

The EMV means the equity market volatity, which obtained counts of categorized terms's existences in over 1000 newspapers. By letting the EMV be independent variable and VIX be dependent variable, we can see how the changes in EMV corresponding to the changes in VIX.

Part 1

Model Specification:

The Gaussian mixture distribution is used in identifying the underlying regimes. The Gaussian mixture distribution is an combination of multiple normal distributions with different weights (or probrability), which commonly used if the data points can hardly be interpret as produced by a simple distribution. In this case, let k representing the number of underlying regimes and assuming that our data is collected from a normal distribution of each regime, we can combine k normal distributions into a Gaussian mixture distribution as a model of stock market.

To start with, the data will be seperated into bins, while each bin representing an interval, and calculate the probrability of having a data point in each bin. This is done by counting the numbers of data points in each bin, and then divide the count by the total number of data points collected. Later, to evaluate the Gaussian mixture distribution derived, a likelihood ratio test will be performed, which compares the observed probability and expect probability of each bin. The following are the code for pre-processing the data.

In [1]:
```
import warnings
import numpy as np
import matplotlib as plt
from scipy.stats.distributions import chi2
from genetic_algorithm import GeneticAlgorithm as ga
import _pickle as pickle
import scipy.integrate as integrate
from kmeans import KMeans
```

get_X is a function for getting raw VIX data from the given path, then return the monthly log-return of the data, which used to constuct the Gaussian mixture distribution.

In [2]:
```
def get_X(path):
    data=pickle.load(open(path, "rb"))
    size=(2019-1990+1)*12-1
    X=np.zeros(shape=(size,1))
    for date in range(1,size):
        X[date-1]=np.log(data.iloc[date]['Adj Close']/data.iloc[date-1]['Adj Close'])
    return X
```

kmeans_pre and bin_pre_kmeans are used together to perform the pre-processing using KMeans method, the clustering method used in last assignment. These two functions calls the Kmeans code to cluster the data. KMeans ensure that each bin is not empty, since it does clustering based on the centers selected. In the employment of KMeans of last assignment, the centers are selected randomly in data points. So in each bin, there is at least one data point, the center itself. After clustering using KMeans, the bound of each bin is computed based on its maximum and minimum data point value, while recording the count. However, this may lead to serious problem because of an unlucky initialization. It is possible to have a bin that only contains the center itself, which means that the maximum and minimum of this is the same. Later when calculating expect probability for having data point in this bin, the bin will have an expect probability of 0, since it is only a point on our one dimensional data.

In [3]:
```
def kmeans_pre(data,K):
    init_centers = data[np.random.randint(data.shape[0], size=K)]
    kmeans=KMeans(init_centers)
    label=kmeans.train(data)
    return label

def bin_pre_kmeans(data,K):
    label=kmeans_pre(data,K)
    info=np.zeros(shape=(K,3))#K rows with [p, max, min]
    for i in range(data.shape[0]):
        for j in range(K):
            if label[i][0]==j:
                if info[j][0]==0:
                    info[j][2]=data[i][0]
                    info[j][1]=data[i][0]
                info[j][0]+=1
                if data[i][0]<info[j][2]:
                    info[j][2]=data[i][0]
                if data[i][0]>info[j][1]:
                    info[j][1]=data[i][0]
    for j in range(K):
        info[j][0]=info[j][0]/data.shape[0]
    return info
```

However,using these KMeans functions may lead to serious problem because of an unlucky initialization. It is possible to have a bin that only contains the center itself, which means that the maximum and minimum of this is the same. Later when calculating expect probability for having data point in this bin, the bin will have an expect probability of 0, since it is only a point on our one dimensional data. So a simple alternative is derive. bin_pre does the pre-processing in a simple way that seperate the range of all data points into k interval of equal length. This may have empty bins, but at least makes sure that each bin can have expect pobrability later. bin_pre and bin_pre_kmeans are designed to have same input and output format, so that a quick switch between two pre-processing methods can be performed.

In [4]:
```
def bin_pre(data,K):
    left=0
    right=0
    for i in range(data.shape[0]):
        if data[i][0]<left:
            left=data[i][0]
        if data[i][0]>right:
            right=data[i][0]
    info=np.zeros(shape=(K,3))
    for j in range(K):
        info[j][1]=right-(right-left)*(j)/K
        info[j][2]=right-(right-left)*(j+1)/K
    for i in range(data.shape[0]):
        for j in range(K):
            if info[j][2]<=data[i][0] and info[j][1]>=data[i][0]:
                info[j][0]+=1
                break
    for j in range(K):
        info[j][0]=info[j][0]/data.shape[0]
    return info
```

The following are the functions for calculating the log-likelihood function. f is a function evaluating the pdf of a normal distribution with given variance, mean, and x. p_theta is the function used for calculating the expected probability of each bin, and return the sum of all bins after multiply with the weight. obj_func is the log-likelihood function, which gives a value of how likely the model match the data observed. But the obj_func is not directly used when using Genetic Algorithm to solve the distribution infomation, since there exists extra constrant needed to be satisfied while optimizing the log-likelihood function.

In [5]:
```
def f(sigma,mu,x):
    return (1./(np.sqrt(2*np.pi)*sigma))*(np.exp(-((x-mu)**2)/(2*(sigma**2))))

def p_theta(bin_start,bin_end,theta,k):
    result=0
    for i in range(k):
        result+=theta[i][2]*(integrate.quad(lambda x: f(theta[i][1],theta[i][0],x)
                                             ,bin_start,bin_end)[0])
    return result

def obj_func(theta,binInfo,k):
    theta=theta.reshape((k,3))
    result=0
    for j in range(binInfo.shape[0]):
        if binInfo[j][0]!=0:
            result+=binInfo[j][0]*np.log(p_theta(binInfo[j][2],binInfo[j][1],theta,k)
                                          /binInfo[j][0])
    result*=(-2)*((2019-1990+1)*12-1)
    return result
```

class obj is a modified version of obj_func, which will be used directly in Genetic Algorithm for optimizing. The eval_obj is the core function of this class, including a lagrangean to release the constraint that sum of weights should be 1. The lagrange multiplier is 10^10, much larger than the log-likelihood function result. So that the sum_p, representing the sum of the weight of each regimes, is as close to 1 as possible.

In [6]:
```
class obj:
    def __init__(self, binInfo,k):
        self.bin=binInfo
        self.k=k

    def eval_obj(self,theta):
        theta=theta.reshape((self.k,3))
        result=0
        binInfo=self.bin
        for j in range(self.bin.shape[0]):
            if binInfo[j][0]!=0:
                result+=binInfo[j][0]*
                np.log(p_theta(binInfo[j][2],binInfo[j][1],theta,self.k)/binInfo[j][0
        result*=(-2)*((2019-1990+1)*12-1)

        #relax constraint: sum of p =1
        sum_p=0
        for i in range(self.k):
            sum_p+=theta[i][2]
        result+= (10**10)*(np.absolute(sum_p-1))
        return result
```

After setting up all functions needed, the Genetic Algorithm is called to optimize the eval_obj. The GA solver is the one provided on Quercus but with some modification to make showing results optional. In solveGA, the parameters of Genetic Algorithm are specified in a relatively counter-intuitive way. The parameters value are derived based on the problem may have in evaluating and optimizing our functions.

The population_size is set as 200 because if the size is too small or very unlucky, we may met problem in evaluating f and p_theta. That is caused by large mean and small variance in the mixture distribution. Note that based on running get_X, the range of X is obtained, which is basically between -1 and 1. As a result, the term of ((x-mu)^2)/(2*(sigma^2)) in f will be huge, and causing the exponential part to underflow. Then, the p_theta will have a result of 0 and cause the log-likelihood function to be infinity. If all population have this problem, that iteration will be wasted and likely to have the same problem in next iteration.

The varbound is set in a way that GA solver and find the optimal faster. At first, the bound of mean is [-10^100, 10^100] and variance is [0,10^100], while 10^100, a value too large to reach, representing the infinity. The problem is even after 3000 iterations, the GA solver cannot provide a reasonable output. The GA solver treats objective function as a blackbox and optimizes it in a brute-force way. So the low efficiency causes it to have bad performance when the bound is large. To improve this, a few iterations of 1. running GA 2. observing result returned 3. lower the bound while containning all variable value returned, was experienced. And finally observed that the means and variances in [-1,1],[0,1] can give a good enough result of log-likelihood function.

In [7]:
```
def solveGA(binInfo,k):
    algorithm_param = {'max_num_iteration': 1000,\
                       'population_size':200,\
                       'mutation_probability':0.1,\
                       'elit_ratio': 0.01,\
                       'crossover_probability': 0.5,\
                       'parents_portion': 0.3,\
                       'crossover_type':'uniform',\
                       'max_iteration_without_improv':None}
    varbound = np.array([[-1,1],[0,1],[0,1]]*k)
    func=obj(binInfo,k)
    model = ga(function=func.eval_obj, dimension=3*k,\
              variable_type='real',\
              variable_boundaries=varbound,
              algorithm_parameters=algorithm_param,
               convergence_curve=False,progress_bar=False,print_result=False)

    model.run()
    return model.best_variable
```

The model and optimization are ready, the analyzation can then be perform.

Fitting and Diagnostics:

First, a certain k should be decided. So run the optimization for k between 1 to 5, then calculate the p-value of the log-likelihood in a Chi-Square distribution with m-3k-1 degree of freedom, which approximate the real world model.

In [8]:

```
    X=get_X("MVIX.pkl")
    np.random.seed(2)
    binInfo=bin_pre(X,48)
    warnings.filterwarnings("ignore")
    theta=[]
    for i in range(1,6):
        theta.append(solveGA(binInfo,i).reshape((i,3)))
        value=obj_func(theta[i-1],binInfo,i)
        print("\nlog-likelihood =",value,"@ k =",i)
        print("Corresponding Chi-Square P-value is",chi2.sf(value,48-3*i-1))
```

```
log-likelihood = 63.381006953854836 @ k = 1
Corresponding Chi-Square P-value is 0.02927793002492517

log-likelihood = 45.182584066201265 @ k = 2
Corresponding Chi-Square P-value is 0.3014768650036335

log-likelihood = 44.834230880023256 @ k = 3
Corresponding Chi-Square P-value is 0.20701663948309162

log-likelihood = 44.20621988005867 @ k = 4
Corresponding Chi-Square P-value is 0.1368461395811883

log-likelihood = 45.93178838372019 @ k = 5
Corresponding Chi-Square P-value is 0.05271618415432073
```

As printed by the code, the P-value of k=2,3,4,5 are all larger than 0.05. So we can conclude that there is sufficient evidence based on the training data to accept an hypothesis that the stock market was governed by 2-4 regimes during the past 20 years. Since k=2 has largest P-value, which means that it is most likely to be the underlying regimes number, let's assume k=2 for the rest of the report. And the theta, the parameters of underlying Gaussian mixture distribution, is shown in form of [mean, variance, weight] for each row representing 1 regime.

In [9]:

```
print("the theta of k = 2 is:\n",theta[1])
```

```
the theta of k = 2 is:
 [[-0.04448881  0.13963195  0.70857822]
 [ 0.09579115  0.23584365  0.29142185]]
```

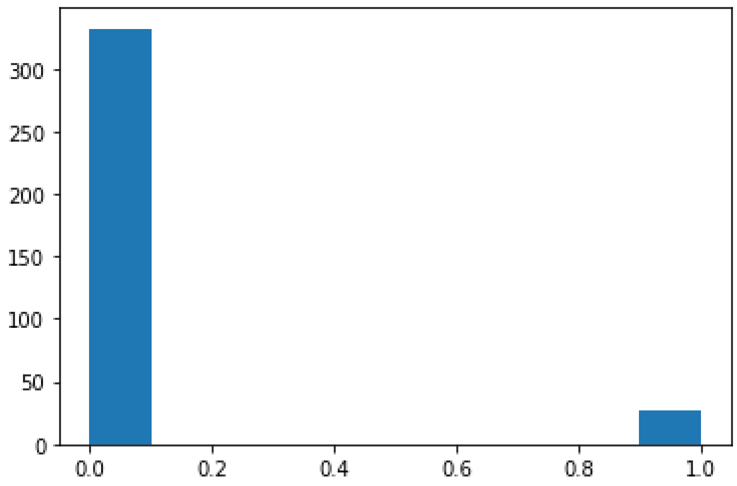
Forecasting:

Then, based on the Gaussian mixture distribution we found, the behaviors of these regimes in the future can be analyze. The function regime computes and compare the probability of an observation x in each regime, using the pdf function of normal distribution multiple by the weight of each regime.

In [10]:
```
def regime(x,theta,k):
    prob=np.ndarray(shape=(k,1))
    reg=0
    for j in range(k):
        prob[j]=theta[j][2]*f(theta[j][1],theta[j][0],x)
        if prob[j]>prob[reg]:
            reg=j
    return reg
```

In [11]:
```
Xreg=np.ndarray(shape=(X.shape[0],1))
theta[1]= [[-0.03574829 ,  0.15181101 ,  0.82710494],
  [ 0.15853556 ,  0.2355648  ,  0.17289517]]
for i in range(X.shape[0]):
    Xreg[i]=regime(X[i],theta[1],2)
Xreg=Xreg.astype(int)
plt.pyplot.hist(Xreg)
```

Out[11]:
```
(array([332.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,  27.]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <BarContainer object of 10 artists>)
```



As shown on the plot, the data contains nearly 92.5% of regime 0, and 7.5% of regime 1. In the following code, a probability matrix will be computed, which shows the probability of changing regime or staying the same regime. Let [i,j] in the matrix represent the probability of regime i changes to regime j.

In [12]:
```
trans=np.zeros(shape=(2,2))
for i in range(X.shape[0]-1):
    trans[Xreg[i][0]][Xreg[i+1][0]]+=1
trans[0]=trans[0]/sum(trans[0])
trans[1]=trans[1]/sum(trans[1])
print(trans)
```

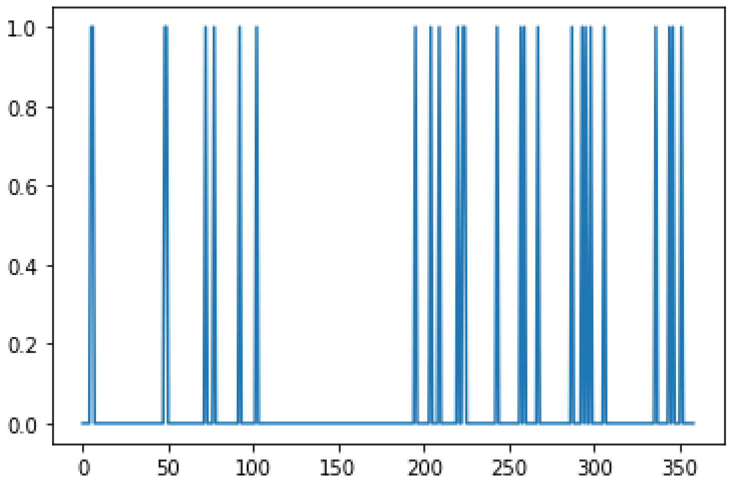
```
[[0.92749245 0.07250755]
 [0.88888889 0.11111111]]
```

The matrix shows that both regimes are more likely to stay, and sometimes changes to other one. Following is a plot where x axis is the months and y axis is the regimes. We can see that the market stayed in regime 0 commonly, and after a period of time it switch to regime 1 for a short time, then back to regime 0.

The phenomenon seems reasonable based on the data from X. In the histogram of X, the distribution is skewing to the right, and the regime 1 has higher mean comparing with regime 0. So it is likely to have low return in stock market, while the probability becomes smaller as the return higher.

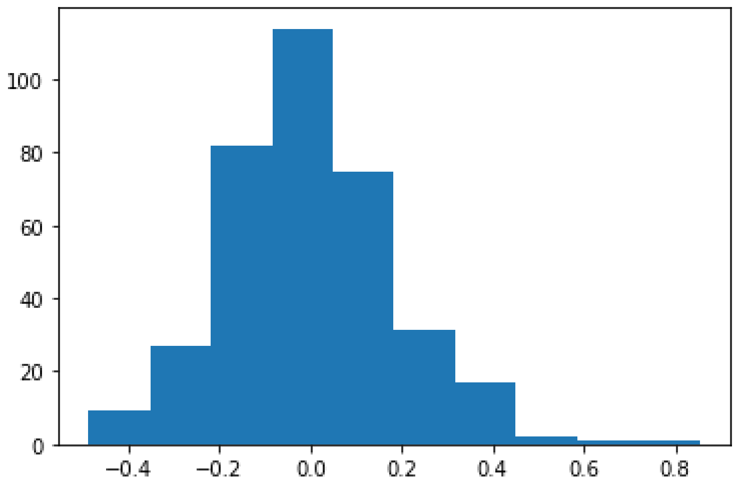
In [13]:
```
plt.pyplot.plot(Xreg)
```

Out[13]:
```
[<matplotlib.lines.Line2D at 0x1ab7151ddf0>]
```



In [14]:
```
plt.pyplot.hist(X)
```

Out[14]:
```
(array([  9.,  27.,  82., 114.,  75.,  31.,  17.,   2.,   1.,   1.]),
 array([-0.4859671 , -0.3521116 , -0.21825609, -0.08440059,  0.04945492,
         0.18331042,  0.31716593,  0.45102143,  0.58487694,  0.71873244,
         0.85258795]),
 <BarContainer object of 10 artists>)
```



Discussion:

As observed previously, the stock market is more likely to have reducing return, and high return regime only has around 7.5%. So that is probably to keep having reducing return in stock market in the future, but there is also possible for the higher return regime to appear in a longer time basis. Note that the period of having high return regime have never be long, and the more common low return regime comes follow.

This approach also have some problems. An important one is using Genetic Algorithm. The Genetic Algorithm optimizes the objective function in a brute-force approach, and the process is significantly affected by randomness. So it is very possible that the result we have is not perfect or general. The other one comes with it is the low efficiency of optimization. Altough the max_iteration is 1000 and population is 200, it is still relatively small for getting a perfect result.