HW3

September 20, 2020

1

Attached is a file containing the heights of men and women at Hope College, see file for details. Consider the two component Gaussian mixture model,

$$X = \begin{cases} \mathcal{N}(\mu_1, \sigma_1^2) & \text{with probability } p_1 \\ \mathcal{N}(\mu_2, \sigma_2^2) & \text{with probability } 1 - p_1 \end{cases}$$

where $\mathcal{N}(\mu, \sigma^2)$ is the normal distribution and X models the height of a person when gender is unknown. Let $\theta = (\mu_1, \mu_2, \sigma_1^2, \sigma_2^2, p_1)$.

a

The data file specifies gender, but pretend you don't have this information. Write down the log-likelihood function $\ell(\theta)$ and $\nabla\ell(\theta)$ given the height samples, i.e. in terms of $\hat{X}_i$. Write R (or Python) functions that calculate $\ell(\theta)$ and $\nabla_\theta\ell(\theta)$.

$$\ell(\theta; x_1, x_2, \ldots, x_n) = \sum_{i=1}^{N} \log(p_1 \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + (1-p_1)\frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}})$$

$$= \sum_{i=1}^{N} \log(p_1 \mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} + (1-p_1)\mathcal{N}(\mu_1, \sigma_1^2)|_{x_i})$$

where the $\mathcal{N}(\mu_1, \sigma_1^2)|_{x_i}$ notation means the value of the denisty of a normal random variable with mean μ_1 and variance σ_1^2 evaluated at the point x_i.

$$\nabla_\theta(\theta) = \begin{pmatrix} \frac{\partial\ell(\theta; x_1, x_2, \ldots, x_n)}{\partial\mu_1} \\ \frac{\partial\ell(\theta; x_1, x_2, \ldots, x_n)}{\partial\mu_2} \\ \frac{\partial\ell(\theta; x_1, x_2, \ldots, x_n)}{\partial\sigma_1^2} \\ \frac{\partial\ell(\theta; x_1, x_2, \ldots, x_n)}{\partial\sigma_2^2} \\ \frac{\partial\ell(\theta; x_1, x_2, \ldots, x_n)}{\partial p_1} \end{pmatrix}$$

$$\frac{\partial \ell(\theta; x_1, x_2, \ldots, x_n)}{\partial \mu_1} = \sum_{i=1}^{N} \frac{p_1 \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} (\frac{x_i-\mu_1}{\sigma_1^2})}{p_1 \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + (1-p_1) \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}}}$$

$$= \sum_{i=1}^{N} \frac{p_1 \mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} (\frac{x_i-\mu_1}{\sigma_1^2})}{p_1 \mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} + (1-p_1) \mathcal{N}(\mu_2, \sigma_2^2)|_{x_i}}$$

By symmetry,

$$\frac{\partial \ell(\theta; x_1, x_2, \ldots, x_n)}{\partial \mu_2} = \sum_{i=1}^{N} \frac{(1-p_1) \mathcal{N}(\mu_2, \sigma_2^2)|_{x_i} (\frac{x_i-\mu_2}{\sigma_2^2})}{p_1 \mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} + (1-p_1) \mathcal{N}(\mu_2, \sigma_2^2)|_{x_i}}$$

$$\frac{\partial \ell(\theta; x_1, x_2, \ldots, x_n)}{\partial \sigma_1^2} = \sum_{i=1}^{N} \frac{p_1 \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} (\frac{(x_i-\mu_1)^2}{\sigma_1^3} - \frac{1}{\sigma_1})}{p_1 \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + (1-p_1) \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}}}$$

$$= \sum_{i=1}^{N} \frac{p_1 \mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} (\frac{(x_i-\mu_1)^2}{\sigma_1^3} - \frac{1}{\sigma_1})}{p_1 \mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} + (1-p_1) \mathcal{N}(\mu_2, \sigma_2^2)|_{x_i}}$$

By symmetry,

$$\frac{\partial \ell(\theta; x_1, x_2, \ldots, x_n)}{\partial \sigma_2^2} = \sum_{i=1}^{N} \frac{(1-p_1) \mathcal{N}(\mu_2, \sigma_2^2)|_{x_i} (\frac{(x_i-\mu_2)^2}{\sigma_2^3} - \frac{1}{\sigma_2})}{p_1 \mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} + (1-p_1) \mathcal{N}(\mu_2, \sigma_2^2)|_{x_i}}$$

Lastly,

$$\frac{\partial \ell(\theta; x_1, x_2, \ldots, x_n)}{\partial p_1} = \sum_{i=1}^{N} \frac{\frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} - \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}}}{p_1 \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + (1-p_1) \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}}}$$

$$= \sum_{i=1}^{N} \frac{\mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} - \mathcal{N}(\mu_2, \sigma_2^2)|_{x_i}}{p_1 \mathcal{N}(\mu_1, \sigma_1^2)|_{x_i} + (1-p_1) \mathcal{N}(\mu_2, \sigma_2^2)|_{x_i}}$$

Let's gather all the terms:

$$\nabla_\theta(\theta) = \begin{pmatrix} \frac{\partial \ell(\theta;x_1,x_2,\ldots,x_n)}{\partial \mu_1} \\ \frac{\partial \ell(\theta;x_1,x_2,\ldots,x_n)}{\partial \mu_2} \\ \frac{\partial \ell(\theta;x_1,x_2,\ldots,x_n)}{\partial \sigma_1^2} \\ \frac{\partial \ell(\theta;x_1,x_2,\ldots,x_n)}{\partial \sigma_2^2} \\ \frac{\partial \ell(\theta;x_1,x_2,\ldots,x_n)}{\partial p_1} \end{pmatrix}$$

$$= \begin{pmatrix} \sum_{i=1}^N \frac{p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}(\frac{x_i-\mu_1}{\sigma_1^2})}{p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}+(1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}} \\ \sum_{i=1}^N \frac{(1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}(\frac{x_i-\mu_2}{\sigma_2^2})}{p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}+(1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}} \\ \sum_{i=1}^N \frac{p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}(\frac{(x_i-\mu_1)^2}{\sigma_1^3}-\frac{1}{\sigma_1})}{p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}+(1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}} \\ \sum_{i=1}^N \frac{(1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}(\frac{(x_i-\mu_2)^2}{\sigma_2^3}-\frac{1}{\sigma_2})}{p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}+(1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}} \\ \sum_{i=1}^N \frac{\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}-\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}}{p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}+(1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}} \end{pmatrix}$$

$$= \sum_{i=1}^N \frac{1}{p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}+(1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}} \begin{pmatrix} p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}(\frac{x_i-\mu_1}{\sigma_1^2}) \\ (1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}(\frac{x_i-\mu_2}{\sigma_2^2}) \\ p_1\mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}(\frac{(x_i-\mu_1)^2}{\sigma_1^3}-\frac{1}{\sigma_1}) \\ (1-p_1)\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i}(\frac{(x_i-\mu_2)^2}{\sigma_2^3}-\frac{1}{\sigma_2}) \\ \mathcal{N}(\mu_1,\sigma_1^2)|_{x_i}-\mathcal{N}(\mu_2,\sigma_2^2)|_{x_i} \end{pmatrix}$$

And so now $\ell(\theta)$ and $\nabla_\theta\ell(\theta)$ are easy to code:

[1]:
```
def logl(mu_1, mu_2, sigma_1, sigma_2, p1):
    """ Computes the log likelihood of a mixture of gaussians
    Args:
        mu_1, sigma_1: parameters of normal distribution 1
        mu_2, sigma_2: parameters of normal distribution 2

    Returns:
        Log likelihood value
    """
    logl = sum(np.log(p1*norm.pdf(x, loc=mu_1, scale=np.sqrt(sigma_1)) +␣
↪(1-p1)*norm.pdf(x, loc=mu_2, scale=np.sqrt(sigma_2))))
    return logl
```

[2]:
```
def grad_logl(mu_1, mu_2, sigma_1, sigma_2, p1):
    """Computes the gradient of the log likelihood of a mixture of gaussians
    Args:
        mu_1, sigma_1: parameters of normal distribution 1
        mu_2, sigma_2: parameters of normal distribution 2

    Returns:
```

```
        Gradient of the log likelihood
    """
    grad = np.array([
        p1*np.multiply(norm.pdf(x, mu_1, np.sqrt(sigma_1)), (x-mu_1)/sigma_1),
        (1-p1)*np.multiply(norm.pdf(x, mu_2, np.sqrt(sigma_2)), (x-mu_2)/
↪sigma_2),
        p1*np.multiply(norm.pdf(x, mu_1, np.sqrt(sigma_1)), (x-mu_1)**2/(np.
↪sqrt(sigma_1)**3)-1/(np.sqrt(sigma_1))),
        p1*np.multiply(norm.pdf(x, mu_2, np.sqrt(sigma_2)), (x-mu_2)**2/(np.
↪sqrt(sigma_2)**3)-1/(np.sqrt(sigma_2))),
        np.multiply(norm.pdf(x, mu_1, np.sqrt(sigma_1)), norm.pdf(x, mu_2, np.
↪sqrt(sigma_2)))
    ])


    grad = np.multiply(1/(p1*norm.pdf(x, mu_1, np.sqrt(sigma_1)) + (1-p1)*norm.
↪pdf(x, mu_2, np.sqrt(sigma_2))),
                       grad)
    grad = np.sum(grad, axis=1)
    return grad
```

b.

Find the MLE for θ by

1. Applying a steepest ascent iteration $\theta^{(i+1)} = \theta^{(i)} + s\nabla\ell(\theta)$.

2. Using nlm or an equivalent in Python.

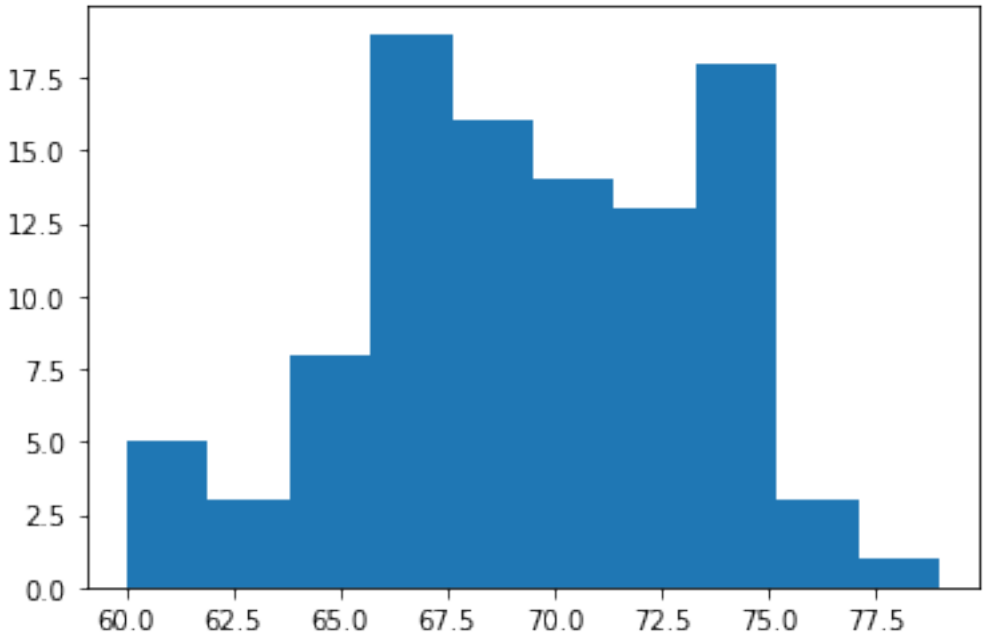
[3]:
```
def steepest_ascent(theta, eps = 10**-4, max_iter = 1000):
    """Steepest ascent algorithm
    Args:
        theta: initial value for parameters, a vector or list of
            size (5, )
        eps: epsilon used to test norm of log likelihood gradient
        max_iter: number of iterations to run algorithm
    Returns:
        i: total number of iterations taken
        logl_list: list with log likelihood values of size (i, )
        theta: parameter reached with algorithm, vector or list of size (5, )

    """
    l = logl(*theta)
    g = grad_logl(*theta)/np.linalg.norm(grad_logl(*theta))
    s = 1.
    i = 0
    logl_list = []
```

```
    while(np.linalg.norm(g) > eps and i < max_iter):

        logl_list.append(l)

        while((theta+s*g)[-1] < 0
              or (theta+s*g)[-1] > 1
              or (theta+s*g)[2] < 0
              or (theta+s*g)[3] < 0):
            s = s/2

        while (logl(*(theta + s*g)) < l):
            s = s/2

        theta = theta + s*g
        l = logl(*theta)
        g = grad_logl(*theta)/np.linalg.norm(grad_logl(*theta))
        i += 1

    return i, logl_list, theta
```

[4]:
```
import pandas as pd
from scipy.stats import norm
import numpy as np

data = pd.read_csv("Hope Heights.txt", sep='\s+').to_numpy()
data.shape
x = data[:, 1]
y = data[:, 0]
print(data.shape, x.shape, y.shape)
```

```
(100, 2) (100,) (100,)
```

[5]:
```
import matplotlib.pyplot as plt
plt.hist(x)
plt.show()
```



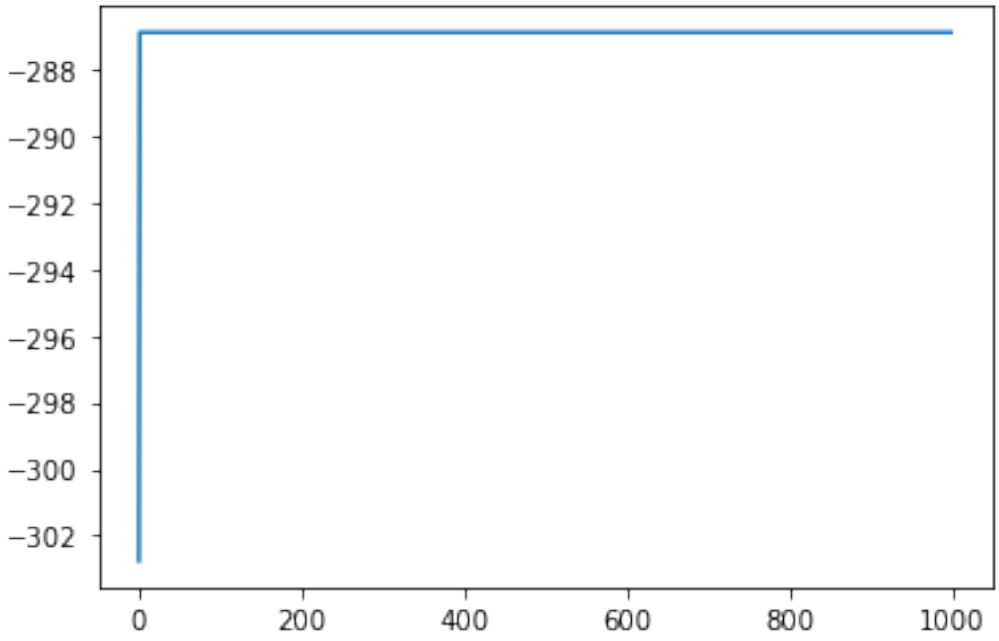
[6]:
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2)
kmeans.fit(x.reshape(-1,1))

theta_0 = np.array([kmeans.cluster_centers_[0].item(),
                    kmeans.cluster_centers_[1].item(),
                    x.var(), x.var(), 0.1])
theta_0
```

[6]:
```
array([72.85714286, 66.09803922, 16.7219    , 16.7219    ,  0.1       ])
```

[7]:
```
iterations, logl_list, theta_star = steepest_ascent(theta_0)
logl_list[-1]
```

[7]:
```
-286.88489162565526
```

[8]:
```
plt.plot(range(iterations), logl_list)
plt.show()
```

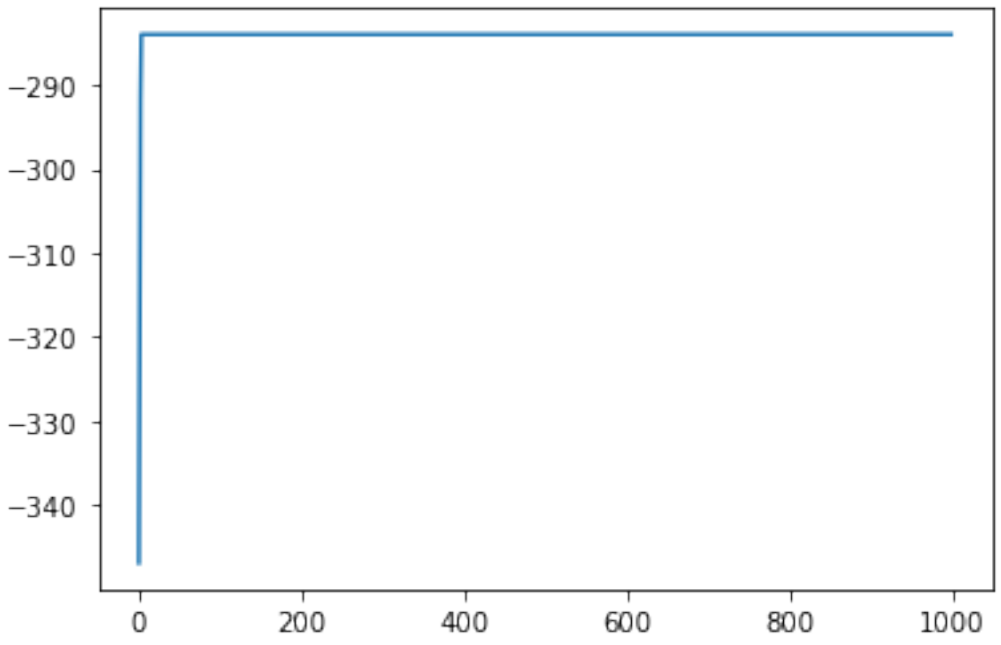


Let's try other initializations since the optimization is non-convex.

[9]:
```
theta_0 = np.array([70, 65, 5, 5, 0.2])
iterations, logl_list, theta_star = steepest_ascent(theta_0)
logl_list[-1]
```

[9]:
```
-284.0289087685018
```

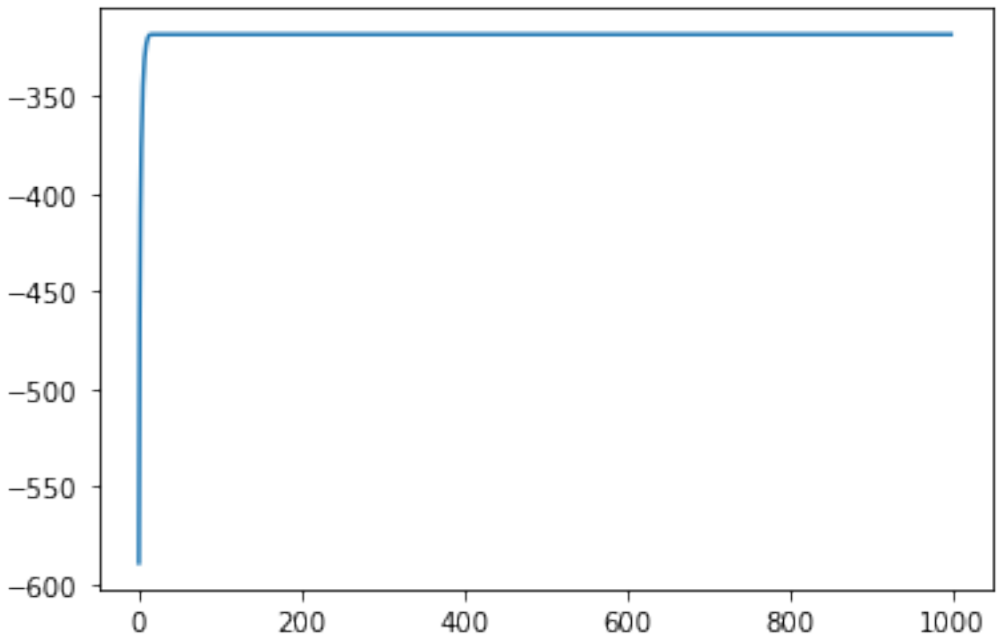
[10]:
```
plt.plot(range(iterations), logl_list)
plt.show()
```



[11]:
```
theta_0 = np.array([70, 40, 2, 2, 0.7])
iterations, logl_list, theta_star = steepest_ascent(theta_0)
logl_list[-1]
```

[11]: -318.3973095925437

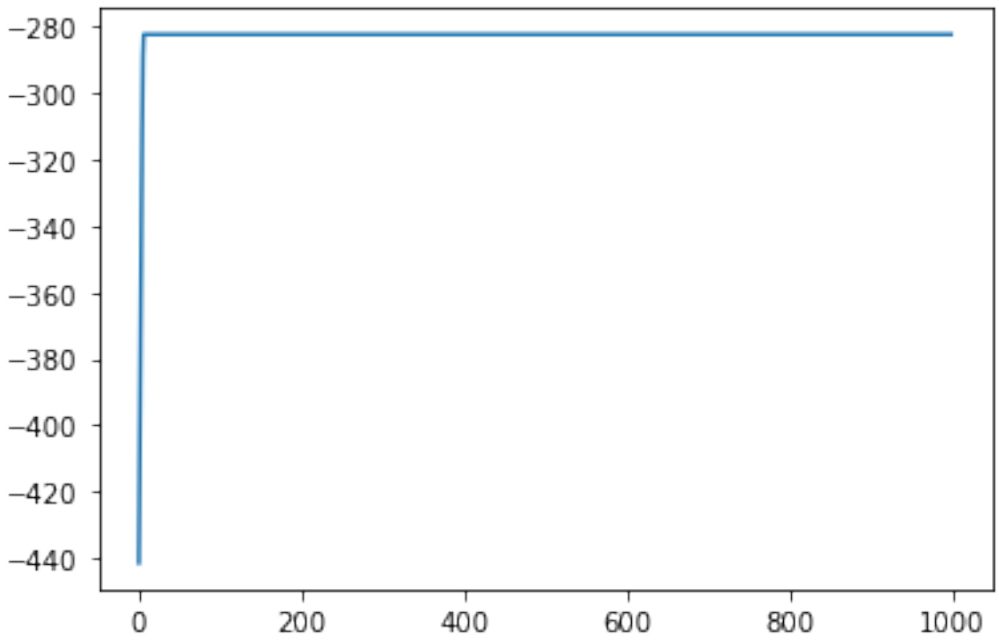
[12]:
```
plt.plot(range(iterations), logl_list)
plt.show()
```



[13]:
```
theta_0 = np.array([69, 76, 5, 2, 0.1])
iterations, logl_list, theta_star = steepest_ascent(theta_0)
logl_list[-1]
```

[13]: -282.4998735842561

[14]:
```
plt.plot(range(iterations), logl_list)
plt.show()
```

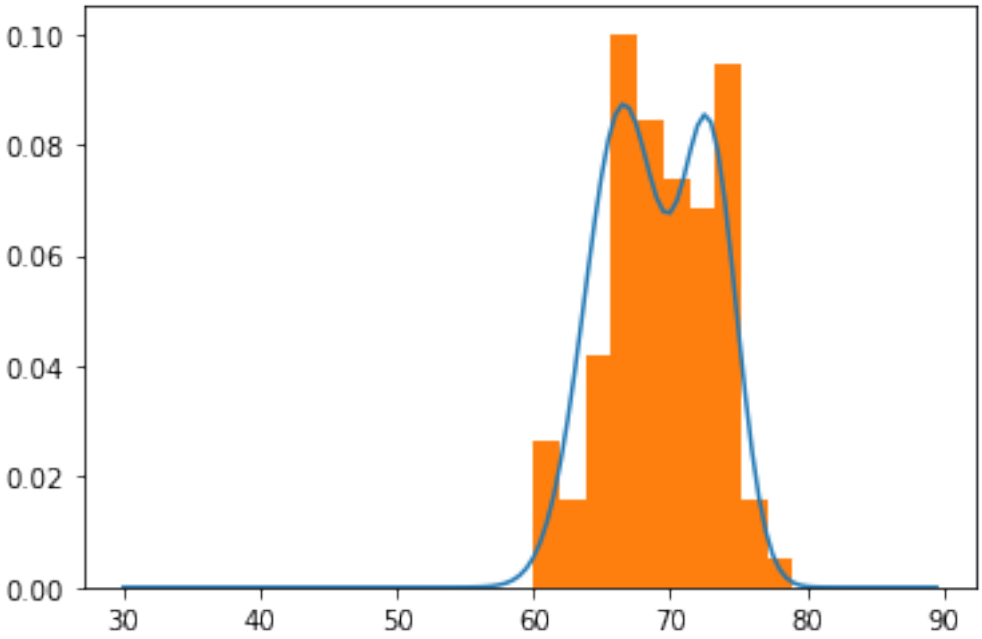


Out of the 4 different intializations, the last one provided the highest log likelihood value so let us choose its corresponding theta_star as our MLE.

[15]:
```
theta_star
```

[15]:
```
array([66.52595766, 72.89819447,  7.67304356,  4.09686584,  0.60212145])
```

[16]:
```
# let's plot the mixture model density given theta_star
# with the height data points
x_points = np.arange(30, 90, 0.5)
y_points = theta_star[-1]*norm.pdf(x_points, theta_star[0], np.
 ↪sqrt(theta_star[2]))+(1-theta_star[-1])*norm.pdf(x_points, theta_star[1], np.
 ↪sqrt(theta_star[3]))
plt.plot(x_points, y_points)
plt.hist(x, density=True)
plt.show()
```



[17]:
```
# changing the log likelihood function so that it can be applied
# to scipy.optimize.minimize
def logl2(theta, x):
    logl = sum(np.log(theta[4]*norm.pdf(x, loc=theta[0], scale=np.
 ↪sqrt(theta[2])) + (1-theta[4])*norm.pdf(x, loc=theta[1], scale=np.
 ↪sqrt(theta[3])))))
    return logl
```

[18]:
```
from scipy.optimize import minimize
minimize(logl2, x0=theta_0, args=x, method='BFGS')
```

```
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:4: RuntimeWarning:
invalid value encountered in log
  after removing the cwd from sys.path.
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:4: RuntimeWarning:
invalid value encountered in sqrt
  after removing the cwd from sys.path.
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:4: RuntimeWarning:
invalid value encountered in log
  after removing the cwd from sys.path.
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:4: RuntimeWarning:
invalid value encountered in sqrt
  after removing the cwd from sys.path.
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:4: RuntimeWarning:
invalid value encountered in sqrt
  after removing the cwd from sys.path.
```

[18]:
```
      fun: nan
 hess_inv: array([[1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1]])
      jac: array([nan, nan, nan, nan, nan])
  message: 'Desired error not necessarily achieved due to precision loss.'
     nfev: 672
      nit: 1
     njev: 112
   status: 2
  success: False
        x: array([  101.59906106,   114.91304062,    -9.89378681,
-17.25884544,
       -1032.60650529])
```

As the errors point out, an invalid value was entered in a log and a sqrt (probably nonnegative for log and negative for sqrt). Choosing any method other than 'BFGS' results in the same (some require that you provide the jacobian, so did not try those out).

The reason as to why this maybe happens is because the python built-in optimizers do not fix the issue of potential negative variances and probabilites (we fix this by backtracking). So when introducing a negative into a log or square root python gives an error.

c

Given your MLE in (b), use the distribution of X to predict whether a given sample is taken from a man or woman. Intuitively, the two normal distributions of X correspond to the male and female height distributions. Given a sample, x, decide which normal the sample is most likely to come from and assign the gender accordingly. Determine what percentage of individuals are classified correctly.

From Bishop section 9.2, we can formulate a gaussian mixture in terms of discrete latent variables. We can introduce a 2-dimensional binary random variable Z in which each entry is either 0 or 1, i.e., $z_i \in \{0, 1\}$ for $i = 1, 2$ and $z_1 + z_2 = 1$. The distribution of Z is in terms of the mixing coefficients:

$$p(z) = \begin{cases} p_1 & \text{if } z_1 = 1 \text{ and } z_2 = 0 \\ 1 - p_1 & \text{if } z_1 = 0 \text{ and } z_2 = 1 \\ 0 & \text{otherwise} \end{cases}$$

And so, to decide which normal the sample is most likely to come from, we want the conditional probability of Z given X, where X was defined in the prompt and its distribution can also be obtained by summing the joint distribution $p(x, z) = p(z)p(x|z)$ over the two possible states of Z : $p(x) = \sum_z p(z)p(x|z) = p_1\mathcal{N}(\mu_1, \sigma_1^2) + (1 - p_1)\mathcal{N}(\mu_2, \sigma_2^2)$.

By Bayes' theorem:

$$p(z_1 = 1, z_2 = 0|x) = \frac{p(z_1 = 1, z_2 = 0)p(x|z_1 = 1, z_2 = 0)}{p(z_1 = 1, z_2 = 0)p(x|z_1 = 1, z_2 = 0) + p(z_1 = 0, z_2 = 1)p(x|z_1 = 0, z_2 = 1)}$$

$$= \frac{p_1\mathcal{N}(\mu_1, \sigma_1^2)}{p_1\mathcal{N}(\mu_1, \sigma_1^2) + (1 - p_1)\mathcal{N}(\mu_2, \sigma_2^2)}.$$

And so,

$$p(z_1 = 0, z_2 = 1|x) = 1 - p(z_1 = 1, z_2 = 0|x).$$

Let's let $\mathcal{N}(\mu_1, \sigma_1^2)$ be the women height distribution and $\mathcal{N}(\mu_2, \sigma_2^2)$ be the men height distribution.

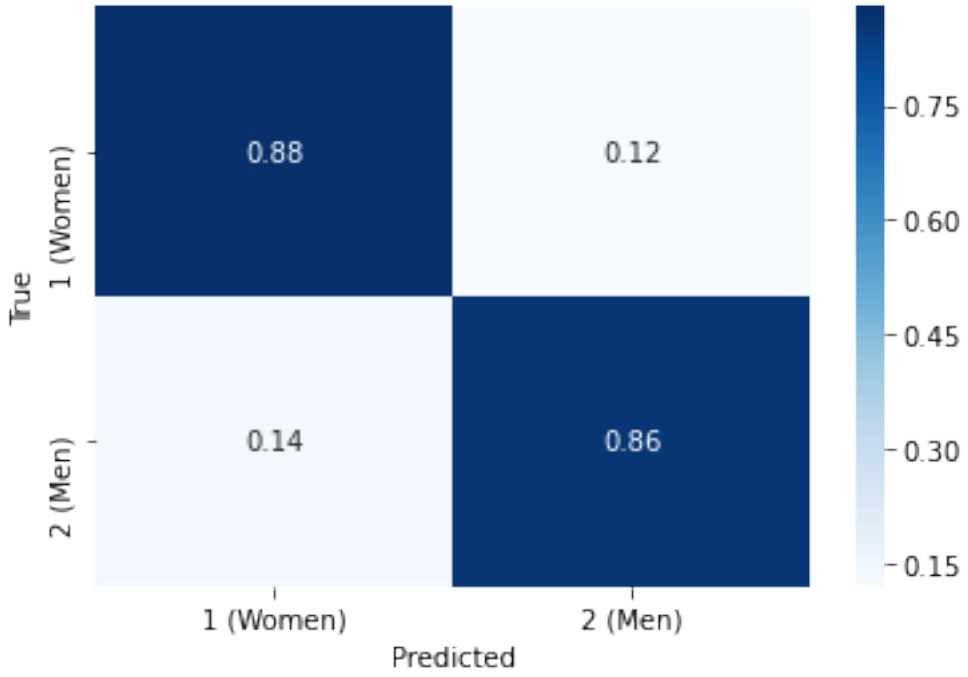
[19]:
```
def classify(cutoff, theta_star):
    num = theta_star[-1]*norm.pdf(x, theta_star[0], np.sqrt(theta_star[2]))
    den = 1/(theta_star[-1]*norm.pdf(x, theta_star[0], np.sqrt(theta_star[2])) \
    + (1-theta_star[-1])*norm.pdf(x, theta_star[1], np.sqrt(theta_star[3])))
    p = np.multiply(num, den)
    return np.where(p > cutoff, 1, 2)
```

[20]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

def plot_confusion_matrix(y_pred):
    c = confusion_matrix(y_true=y, y_pred=y_pred)
    c = c/c.astype(np.float).sum(axis=1)
    fig, ax = plt.subplots(figsize=(6,4))
    sns.heatmap(c, annot=True, fmt='.2f', cmap="Blues",
                xticklabels=["1 (Women)", "2 (Men)"],
                yticklabels=["1 (Women)", "2 (Men)"])
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.show()
```

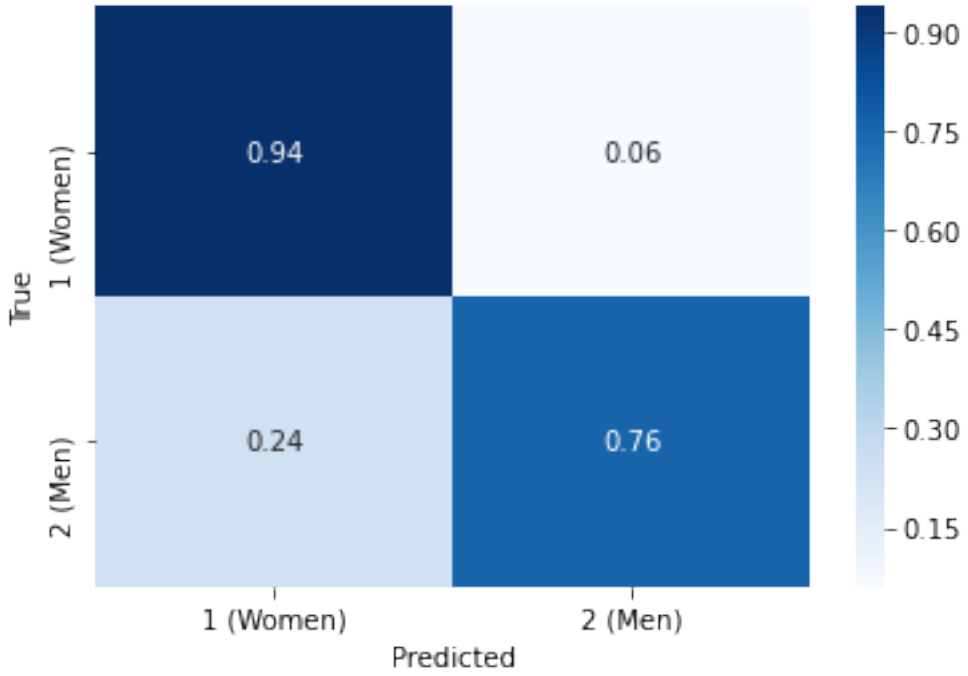
```
//anaconda3/lib/python3.7/site-packages/statsmodels/tools/_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the
public API at pandas.testing instead.
  import pandas.util.testing as tm
```

Let's try a couple of different cutoffs.

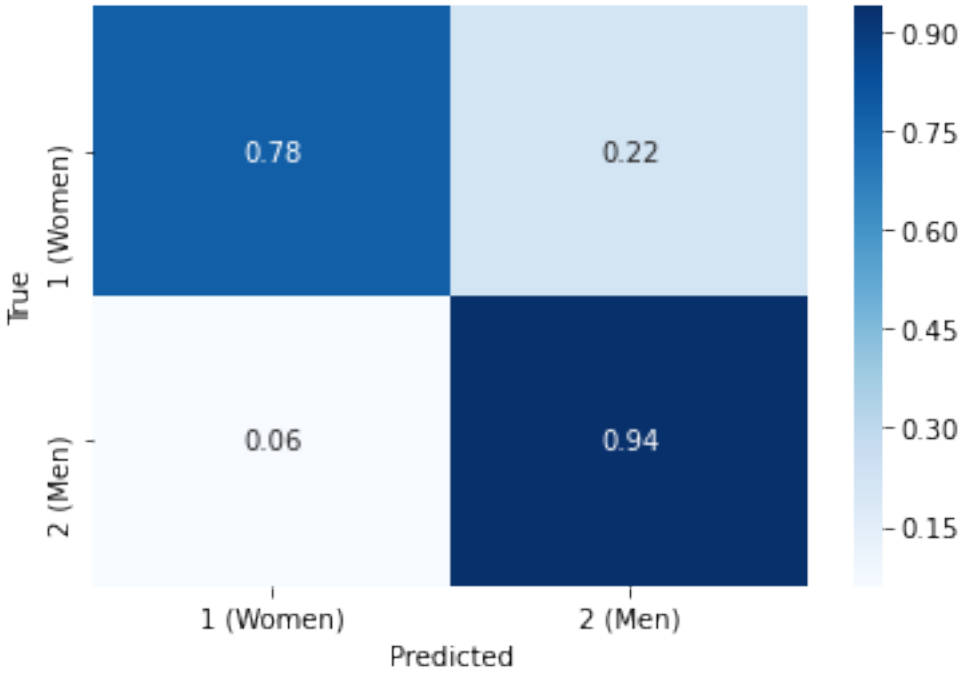
[21]:
```
plot_confusion_matrix(classify(0.8, theta_star))
```



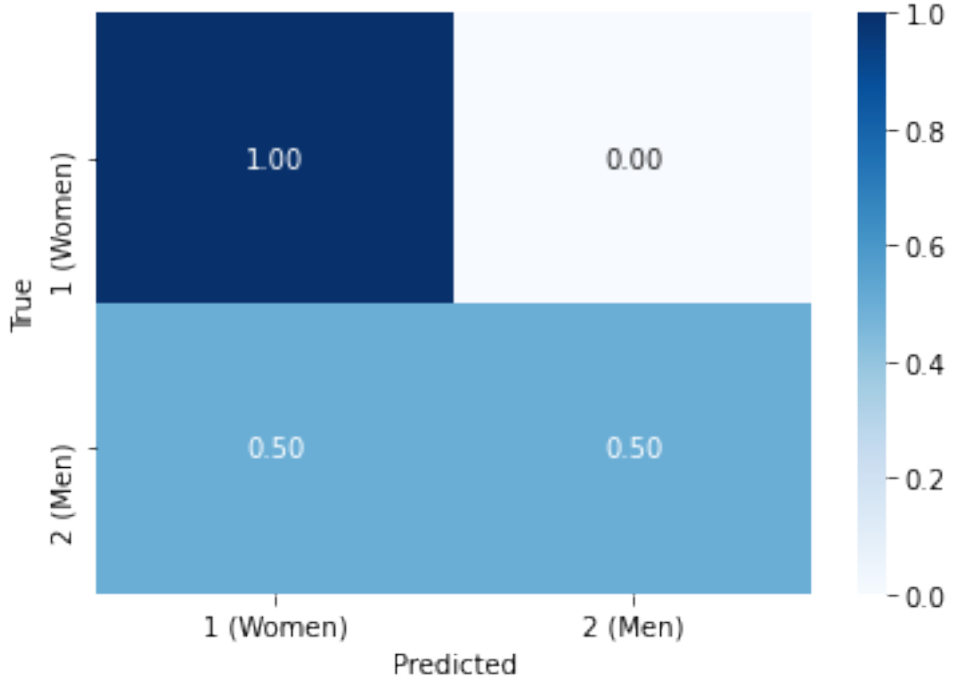
[22]:
```
plot_confusion_matrix(classify(0.4, theta_star))
```



[23]:
```
plot_confusion_matrix(classify(0.9, theta_star))
```

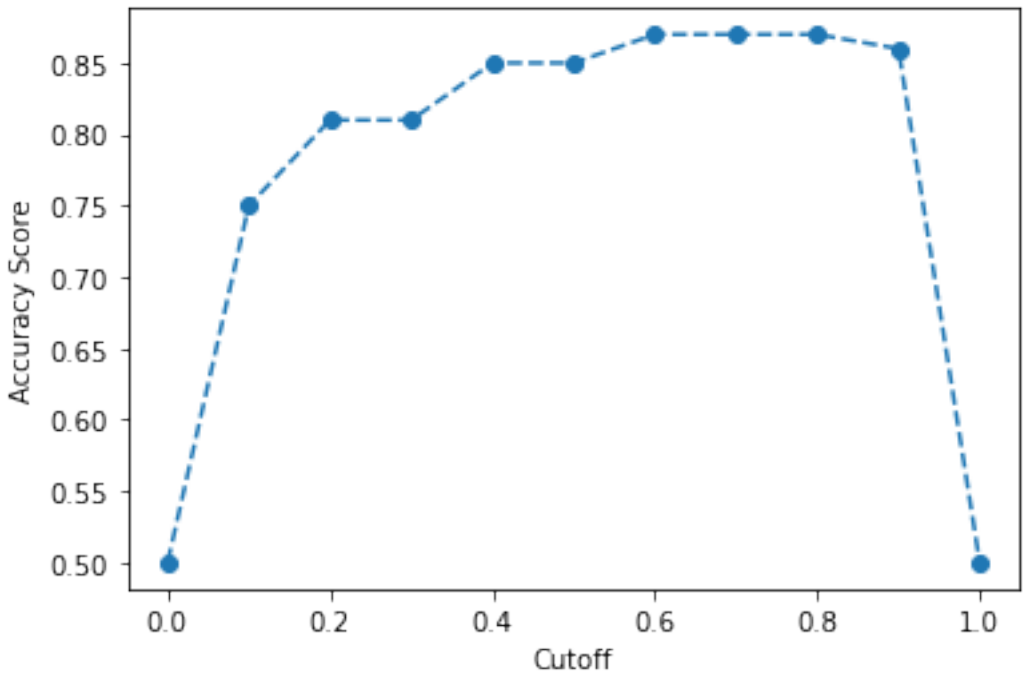


[24]:
```
plot_confusion_matrix(classify(0.1, theta_star))
```



Now let's plot accuracy for a couple of different cutoffs.

[25]:
```
from sklearn.metrics import accuracy_score

accuracies = [accuracy_score(y_true=y, y_pred=classify(p, theta_star)) for p in␣
 ↪np.arange(0,1.1,0.1)]
plt.plot(np.arange(0,1.1,0.1), accuracies, "o--")
plt.xlabel("Cutoff")
plt.ylabel("Accuracy Score")
plt.show()
```



A cutoff of 0.8 seems to give us the best accuracy of 0.87 with a decent confusion matrix.

2

Consider the Chutes and Ladders Markov chain of the previous homework. Assume throughout that a single player is playing.

a

Compute the probability of winning in n steps for $n = 1, 2, ..., 50$. Plot the probabilities. Don't use a Monte Carlo approach, compute these probabilities exactly using matrix computations.

Using the *build_P* function used in HW 2:

[26]:
```
def build_P(chutes_and_ladder_locations):
    """Builds the transition probability matrix for the chutes and ladders game

    Args:
        chutes_and_ladder_locations: Dataframe containing the info from
        the chutes_and_ladder_locations.csv

    Returns:
        A numpy array of shape (101, 101)

    """
    # helper function
    def shift(array):
        return(np.concatenate((np.array([0]), array[:-1])))

    # creating the transition probability
    # matrix, P, assuming no chutes and ladders
    pdf = np.zeros(101)
    pdf[1:7] = 1/6
    P = pdf.copy()

    for _ in range(100):
        pdf = shift(pdf)
        P = np.concatenate((P, pdf))

    P = P.reshape(101, 101)

    P[100, 100] = 1

    # fixing P for chutes and ladders
    for i, j in zip(chutes_and_ladder_locations.start,␣
↪chutes_and_ladder_locations.end):
        P[:,j] += P[:,i]
        P[:,i] = np.zeros(101)

    # fixing P for last squares in game
    for i in [95, 96, 97, 98, 99]:
        P[i, -1] = 1 -  np.sum(P[i, :-1])

    return P
```

[27]:
```
import pandas as pd
chutes_and_ladder_locations = pd.read_csv('../HW2/chutes_and_ladder_locations.
↪csv')
chutes_and_ladder_locations.head()
```

[27]:

	start	end
0	1	38
1	4	14
2	9	31
3	21	42
4	28	84

[28]:
```
P = build_P(chutes_and_ladder_locations)
```

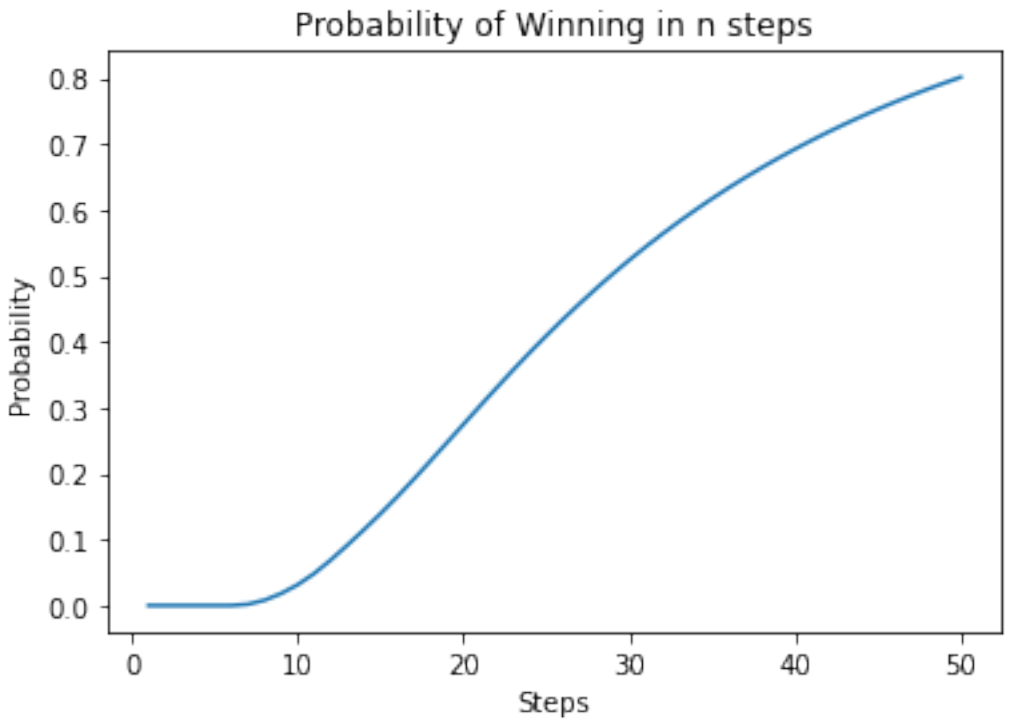
Note that $X_0 = 0$. We want the probability of winning in n steps for $n = 1, 2, 3, ..., 50$, which is

$$\mathbb{P}\{X_n = 100 | X_0 = 0\} = P^n_{(0,100)}$$

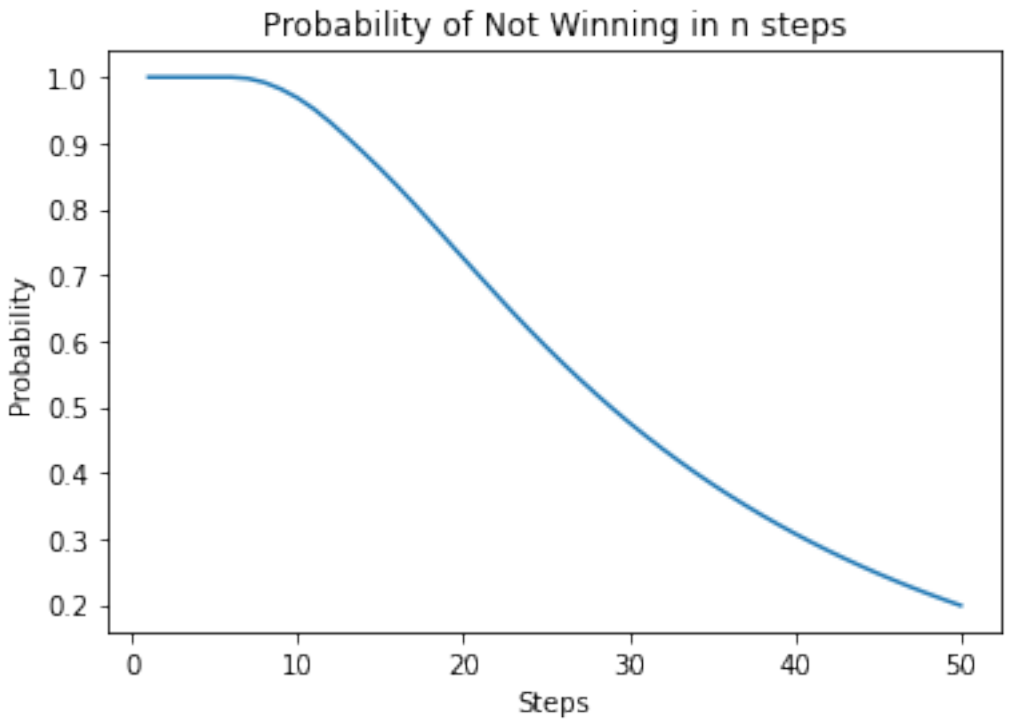
for $n = 1, 2, 3, ..., 50$, where P^n means matrix multiplying P n times.

[29]:
```
Pn = P.copy()
probabilities = []
for _ in range(1, 51):
    probabilities.append(Pn[0,100])
    Pn = P.dot(Pn)
```

[30]:
```
# plot the probabilities of winning in n steps
plt.plot(range(1, 51), probabilities)
plt.title("Probability of Winning in n steps")
plt.xlabel("Steps")
plt.ylabel("Probability")
plt.show()
```



[31]:
```
# Let's plot the probabilities of not winning in n steps too
plt.plot(range(1, 51), [1-p for p in probabilities])
plt.title("Probability of Not Winning in n steps")
plt.xlabel("Steps")
plt.ylabel("Probability")
plt.show()
```



b

What's the probability of a game lasting more than 1000 moves?

[32]:
```
Pn = P.copy()
probabilities = []
for _ in range(1, 1001):
    probabilities.append(Pn[0,100])
    Pn = P.dot(Pn)
```

The probability of a game lasting more than 1000 moves is

[33]:
```
1 - Pn[0, 100]
```

[33]: 2.3314683517128287e-15

c

Suppose we modify Chutes and Ladders so that the player wraps back to square 1 once they go beyond square 100. So for example, if the player is on square 99 and roles a 3, they end up on square 2.

i. Compute the stationary distribution. After 1 billion moves, on what square is a

player's piece most likely to be on?

Modifying *build_P*:

[34]:
```
def build_P_v2(chutes_and_ladder_locations):
    """Builds the transition probability matrix for the chutes and ladders game

    Args:
        chutes_and_ladder_locations: Dataframe containing the info from
        the chutes_and_ladder_locations.csv

    Returns:
        A numpy array of shape (101, 101)

    """
    # helper function
    def shift(array):
        return(np.concatenate((np.array([0]), array[:-1])))

    # creating the transition probability
    # matrix, P, assuming no chutes and ladders
    pdf = np.zeros(101)
    pdf[1:7] = 1/6
    P = pdf.copy()

    for _ in range(100):
        pdf = shift(pdf)
        P = np.concatenate((P, pdf))

    P = P.reshape(101, 101)

    # fixing P for last squares in game
    for i in [95, 96, 97, 98, 99, 100]:
        for j in range(1, i+6-99):
            P[i, j] = 1/6

    # fixing P for chutes and ladders
    for i, j in zip(chutes_and_ladder_locations.start,␣
↪chutes_and_ladder_locations.end):
        P[:,j] += P[:,i]
        P[:,i] = np.zeros(101)

    return P
```

[35]:
```
P = build_P_v2(chutes_and_ladder_locations)
```

[36]:
```
P[95,:]
```

[36]:
```
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.16666667, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.16666667, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.16666667, 0.16666667, 0.        , 0.16666667,
       0.16666667])
```

[37]:
```
P[98,:]
```

[37]:
```
array([0.        , 0.        , 0.16666667, 0.16666667, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.16666667,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.16666667, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.16666667,
       0.16666667])
```

[38]:
```
P[100, :]
```

[38]:
```
array([0.        , 0.        , 0.16666667, 0.16666667, 0.        ,
       0.16666667, 0.16666667, 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.16666667,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.16666667, 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        ])
```

Given the above, the changes look good.

[39]:
```
eigenvalues = np.linalg.eig(P)[0]
eigenvectors = np.linalg.eig(P)[1]
```

[40]:
```
eigenvalues
```

[40]:
```
array([ 0.        +0.j        ,  0.        +0.j        ,
        0.        +0.j        ,  0.        +0.j        ,
        0.        +0.j        ,  0.        +0.j        ,
        0.        +0.j        ,  0.        +0.j        ,
        0.        +0.j        ,  0.        +0.j        ,
        0.        +0.j        ,  0.        +0.j        ,
        0.        +0.j        ,  0.        +0.j        ,
        0.        +0.j        ,  0.        +0.j        ,
        1.        +0.j        ,  0.79967581+0.1871812j ,
        0.79967581-0.1871812j ,  0.38906409+0.667867j  ,
        0.38906409-0.667867j  ,  0.65804093+0.34373306j,
        0.65804093-0.34373306j,  0.56329098+0.38363599j,
        0.56329098-0.38363599j,  0.40188969+0.54194621j,
        0.40188969-0.54194621j,  0.17651432+0.62831241j,
        0.17651432-0.62831241j,  0.00558521+0.50705448j,
        0.00558521-0.50705448j, -0.09166505+0.49801901j,
```

```
-0.09166505-0.49801901j, -0.22235329+0.44368131j,
-0.22235329-0.44368131j, -0.25320134+0.36712995j,
-0.25320134-0.36712995j, -0.25025582+0.25147793j,
-0.25025582-0.25147793j, -0.2237589 +0.12384945j,
-0.2237589 -0.12384945j,  0.02011819+0.14581843j,
 0.02011819-0.14581843j, -0.09021372+0.17320746j,
-0.09021372-0.17320746j, -0.00188651+0.13778718j,
-0.00188651-0.13778718j, -0.16555567+0.13791522j,
-0.16555567-0.13791522j, -0.10392421+0.15713579j,
-0.10392421-0.15713579j,  0.03795025+0.0953242j ,
 0.03795025-0.0953242j , -0.21891259+0.04940595j,
-0.21891259-0.04940595j, -0.03831582+0.12270549j,
-0.03831582-0.12270549j, -0.14233151+0.09945197j,
-0.14233151-0.09945197j,  0.01263549+0.0435041j ,
 0.01263549-0.0435041j , -0.15712056+0.07009848j,
-0.15712056-0.07009848j,  0.01409874+0.02669607j,
 0.01409874-0.02669607j, -0.18385829+0.j        ,
-0.0129397 +0.0652976j , -0.0129397 -0.0652976j ,
-0.0745633 +0.08757022j, -0.0745633 -0.08757022j,
-0.172426  +0.02937612j, -0.172426  -0.02937612j,
-0.10951672+0.0786037j , -0.10951672-0.0786037j ,
-0.1288786 +0.06566206j, -0.1288786 -0.06566206j,
-0.15793737+0.j        , -0.15241888+0.j        ,
-0.1362053 +0.04325108j, -0.1362053 -0.04325108j,
-0.02093471+0.03363125j, -0.02093471-0.03363125j,
-0.0091084 +0.j        , -0.01044901+0.j        ,
-0.16666667+0.j        , -0.08434219+0.04618487j,
-0.08434219-0.04618487j, -0.04718054+0.03018104j,
-0.04718054-0.03018104j, -0.06169204+0.01084608j,
-0.06169204-0.01084608j, -0.121736  +0.j        ,
-0.0342983 +0.j        , -0.08930798+0.01548685j,
-0.08930798-0.01548685j, -0.09230602+0.j        ,
-0.10065097+0.j        ])
```

As can be seen from above, there is an eigenvalue of 1. Let's find corresponding eigenvector.

[41]:
```
# Abs gives you mod in python for complex numbers too
i = np.argmax(abs(eigenvalues))
eigenvector_1 = eigenvectors[i]
```

[42]:
```
eigenvector_1 = eigenvector_1.real
# values between 0 and 1
eigenvector_1 = (eigenvector_1-min(eigenvector_1)/
 ↪(max(eigenvector_1)-min(eigenvector_1)))
# probability vector
eigenvector_1 = eigenvector_1/sum(eigenvector_1)
eigenvector_1
```

[42]:
```
array([0.00980723, 0.00980723, 0.00980723, 0.00980723, 0.00980723,
       0.00980723, 0.00980723, 0.00980723, 0.00980723, 0.00980723,
       0.00980723, 0.00980723, 0.00980723, 0.00980723, 0.00980723,
       0.00980723, 0.00980723, 0.00980723, 0.00980723, 0.00980723,
       0.01171894, 0.0103375 , 0.0103375 , 0.01018868, 0.01018868,
       0.0107733 , 0.0107733 , 0.0076881 , 0.0076881 , 0.00978059,
       0.00978059, 0.00853084, 0.00853084, 0.00996376, 0.00996376,
       0.01099542, 0.01099542, 0.01155829, 0.01155829, 0.00915199,
       0.00915199, 0.00830023, 0.00830023, 0.01054372, 0.01054372,
       0.00966873, 0.00966873, 0.00864146, 0.00864146, 0.01172424,
       0.01172424, 0.00992807, 0.00992807, 0.00894075, 0.00894075,
       0.01124826, 0.01124826, 0.00862361, 0.00862361, 0.00999539,
       0.00999539, 0.0111016 , 0.0111016 , 0.01032294, 0.01032294,
       0.0066198 , 0.0066198 , 0.00995494, 0.00995494, 0.01286401,
       0.01027398, 0.01027398, 0.00985355, 0.00985355, 0.00811239,
       0.00811239, 0.01093649, 0.01093649, 0.01236931, 0.01236931,
       0.0102761 , 0.00874271, 0.01022354, 0.01022354, 0.0093562 ,
       0.0093562 , 0.00968466, 0.00995411, 0.01224721, 0.00949898,
       0.00949898, 0.00915955, 0.00915955, 0.00985306, 0.00985306,
       0.01103086, 0.00967902, 0.01063287, 0.01063287, 0.00926527,
       0.00868826])
```

The above is the stationary distribution. And below find the square a player's piece is most likely to be on after 1 billion moves.

[43]:
```
np.argmax(eigenvector_1)
```

[43]:
```
69
```

ii. Compute the relaxation time and explain why it makes sense in terms of your results for the expected time of a game in the previous homework.

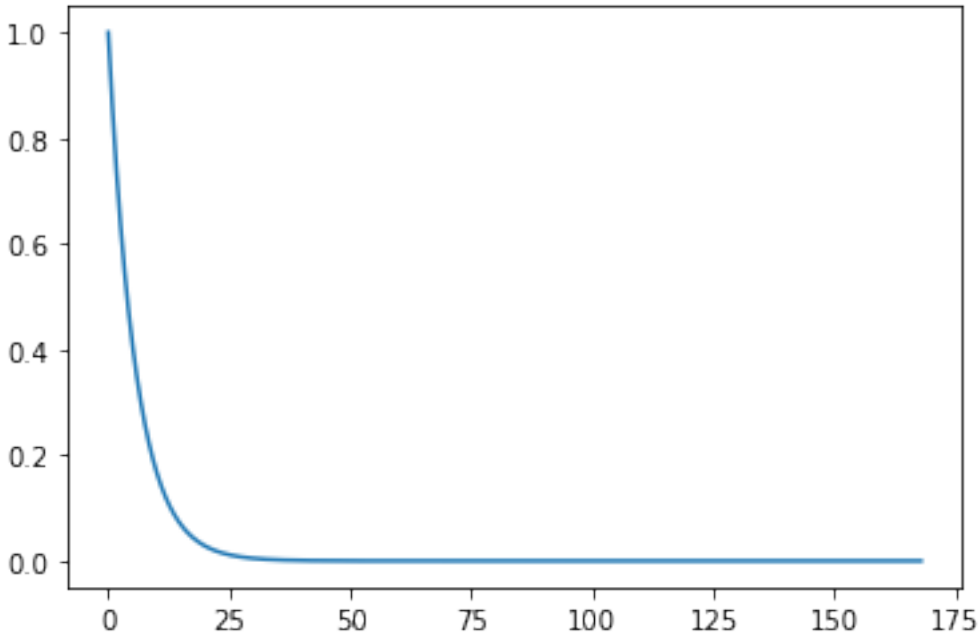
[44]:
```
eigenvalue_2 = sorted(abs(eigenvalues), reverse=True)[1]
```

The relaxation time is:

[45]:
```
rt = 1/(1-eigenvalue_2)
rt
```

[45]:
```
5.595675700579098
```

[46]:
```
x_points = np.arange(0.00001, 30, 0.01)*rt
y_points = np.exp(-x_points*1/rt)
plt.plot(x_points, y_points)
plt.show()
```



The relaxation to stationarity is exponentially fast for particular Markov chains (like the one we are considering). In other words, there is a typical time, the relaxation time, after which a stationary regime is established. Not sure how it relates to the expected value in the last HW, I think relaxation times have more to do with big O (time complexity) and being able to compare times to convergence among different particular types of markov chains (i.e., not sure if the paricular number 5 has a specific meaning and how it could relate to the expected value of the length of a game).

iii. Now suppose that we play the game as follows. Each round, with probability p the player doesn't move. With probability $1-p$ the player rolls the die and moves as usual. For $p = .9, .99, .999$, compute the relaxation time. Explain the pattern you see in the relaxation times.

[47]:
```
def TPM(p, P):
    return (1-p)*P + p*np.identity(101)
```

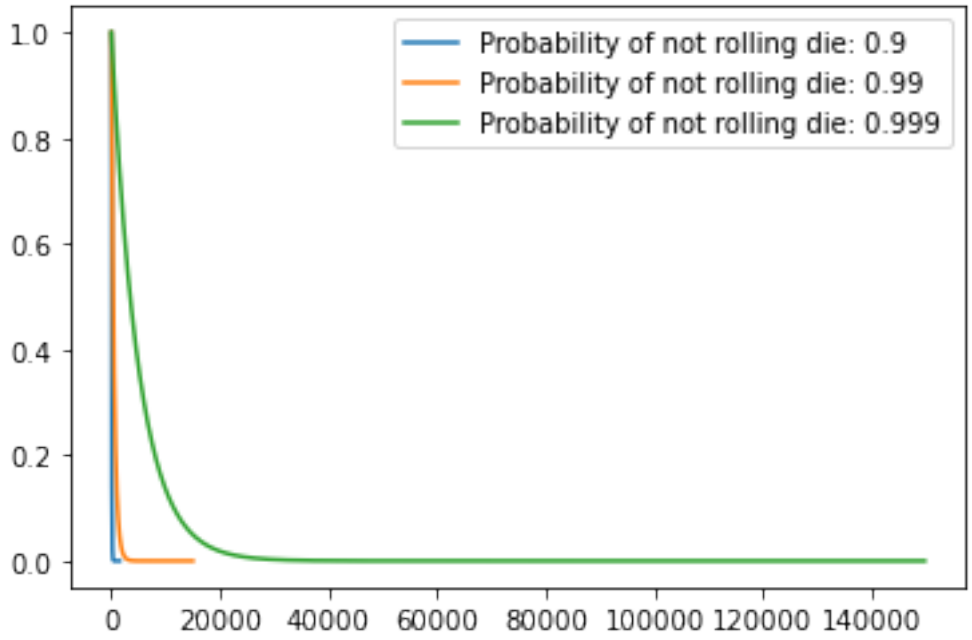
[48]:
```
def compute_relaxation_time(P):
    eigenvalues = np.linalg.eig(P)[0]
    eigenvalue_2 = sorted(abs(eigenvalues), reverse=True)[1]
    return 1/(1-eigenvalue_2)
```

[49]:
```
rt = []
for p in [.9,.99,.999]:
    tpm = TPM(p, P)
    relaxation_time = compute_relaxation_time(tpm)
    print(f"Probability of not rolling die: {p} \t Relaxation Time:␣
 ↪{relaxation_time}")
```

```
    rt.append(relaxation_time)
```

```
Probability of not rolling die: 0.9      Relaxation Time: 50.368521142963836
Probability of not rolling die: 0.99     Relaxation Time: 499.6286432833968
Probability of not rolling die: 0.999    Relaxation Time: 4992.345070657904
```

[50]:
```
x_points = np.arange(0.00001, 30, 0.01)*rt[0]
y_points = np.exp(-x_points*1/rt[0])
plt.plot(x_points, y_points, label="Probability of not rolling die: 0.9")
x_points = np.arange(0.00001, 30, 0.01)*rt[1]
y_points = np.exp(-x_points*1/rt[1])
plt.plot(x_points, y_points, label="Probability of not rolling die: 0.99")
x_points = np.arange(0.00001, 30, 0.01)*rt[2]
y_points = np.exp(-x_points*1/rt[2])
plt.plot(x_points, y_points, label="Probability of not rolling die: 0.999")
plt.legend()
plt.show()
```



As probability of p increases, relaxation time increases. This makes sense: the more likely we are to not move (not roll die), the longer it will take us to reach the stationary distribution. The matrix $(1-p) * P + p *$ np.identity(101), call it Q, should have the same stationary distribution as P, regardless of probability p. The only difference is that since Q now allows for the possibility of remaining in the same state it takes longer for Q than P to reach the stationary distribution.

[ ]: