Coursework 2: probabilistic ranking

Original by Carl Rasmussen and Manon Kok for CUED course 4f13. This version adapted by Damon Wischik.

In this assignment, you'll be using the (binary) results of the 2011 ATP men's tennis singles for 107 players in a total of 1801 games (which these players played against each other in the 2011 season), to compute probabilistic rankings of the skills of these players.

What to submit. Your answers should contain an explanation of what you do, and 2–4 central commands to achieve it. Complete listings are unnecessary. The focus of your answer should be *interpretation:* explain what the numerical values and graphs you produce *mean,* and why they are as they are. The text of your answer to each question should be no more than a paragraph or two.

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import scipy.io, scipy.stats, scipy.linalg
from numpy.linalg import solve
import requests, io
%matplotlib inline
```

Data import

The match data is provided as `https://teachingfiles.blob.core.windows.net/probml/tennis_data.mat`. It contains a vector W of length 107 whose ith entry is the name of player i; and an array G of dimension 1801×2, one row per game, where the first column contains the identity of the player who won and the second column contains the identify of the player who lost. Note that this convention means that the variable y_g (the game outcome) in lecture notes is always $+1$, and can consequently be ignored. Some rows will appear more than once (corresponding to two players having played each other several times with the same outcome).

```
r = requests.get('https://teachingfiles.blob.core.windows.net/probml/tennis_data.mat')
with io.BytesIO(r.content) as f:
    data = scipy.io.loadmat(f)
    W = np.concatenate(data['W'].squeeze())
    G = data['G'] - 1   # W[G[i,0]] is winner of game i, W[G[i,1]] is loser
    M = W.shape[0]      # number of players M = 107
    N = G.shape[0]      # number of games N = 1801
```

The appendix at the bottom of this document shows a crude ranking of the players, based on the fraction of their games that they win.

Question (a)

Complete the following code for Gibbs sampling, by adding the lines required to sample from the conditional distributions needed for Gibbs sampling for the ranking model discussed in lectures. Run the Gibbs sampler, e.g. for 1100 iterations. Plot some of the sampled player skills as a function of the Gibbs iteration (see the Section for some tips on multiple plotting).

Does it look as though the Gibbs sampler is able to move around the posterior distribution? What are the burn-in and mixing times for the Gibbs sampler? It may be helpful to look at autocorrelation plots; see the Section .

```
pv = 0.5 * np.ones(M)   # prior variance
w = np.zeros(M)         # skills, initialized to be the prior mean μ0 = 0

for _ in range(1100):
    # Sample performance differences (t) given skills (w) and outcomes (G)
    s = w[G[:,0]] - w[G[:,1]]
    σ = 1
    t = s + σ * scipy.stats.norm.ppf(1 - np.random.uniform(size=N)*(1-scipy.stats.norm.cdf(-s/σ)))

    # Sample skills given performance differences
    Σinv = ...
    Σ = np.linalg.inv(Σinv)
    μtilde = ...
    μ = Σ @ μtilde
    w = np.random.multivariate_normal(mean=μ, cov=Σ)
```

Question (b)

Do inference in the model instead, by running message passing and expectation propagation, using the code below. For the same players you plotted in part (a), show the mean and standard deviation of their skill levels as a function of the iteration. How long does it take to converge?

(See the Python programming note in the appendix, about the `yield` statement and how to use `gaussian_ep`.)

```
def gaussian_ep(G, M):
    def Ψ(x): return scipy.stats.norm.pdf(x) / scipy.stats.norm.cdf(x)
    def Λ(x): return Ψ(x) * (Ψ(x) + x)
    N = len(G)

    μ_s, p_s = np.empty(M), np.empty(M)
    μ_gs, p_gs = np.zeros((N,2)), np.zeros((N,2))
    μ_sg, p_sg = np.empty((N,2)), np.empty((N,2))

    while True:
        # 1. Compute marginal skills
        # Let skills be N(μ_s, 1/p_s)
        p_s = np.ones(M) * 1/0.5
        μ_s = np.zeros(M)
        for j,(winner,loser) in enumerate(G):
            p_s[winner] += p_gs[j,0]
            p_s[loser] += p_gs[j,1]
            μ_s[winner] += μ_gs[j,0] * p_gs[j,0]
            μ_s[loser] += μ_gs[j,1] * p_gs[j,1]
        μ_s = μ_s / p_s

        # 2. Compute skill -> game messages
        # winner's skill -> game: N(μ_sg[,0], 1/p_sg[,0])
        # loser's skill -> game: N(μ_sg[,1], 1/p_sg[,1])
        p_sg = p_s[G] - p_gs
        μ_sg = (p_s[G]*μ_s[G] - p_gs*μ_gs) / p_sg

        # 3. Compute game -> performance messages
        v_gt = 1 + np.sum(1/p_sg, 1)
        σ_gt = np.sqrt(v_gt)
        μ_gt = μ_sg[:,0] - μ_sg[:,1]

        # 4. Approximate the marginal on performance differences
        μ_t = μ_gt + σ_gt * Ψ(μ_gt/σ_gt)
        p_t = 1 / v_gt / (1-Λ(μ_gt/σ_gt))

        # 5. Compute performance -> game messages
        p_tg = p_t - 1/v_gt
        μ_tg = (μ_t*p_t - μ_gt/v_gt) / p_tg

        # 6. Compute game -> skills messages
        # game -> winner's skill: N(μ_gs[,0], 1/p_gs[,0])
        # game -> loser's skill: N(μ_gs[,1], 1/p_gs[,1])
        p_gs[:,0] = 1 / (1 + 1/p_tg + 1/p_sg[:,1])  # winners
        p_gs[:,1] = 1 / (1 + 1/p_tg + 1/p_sg[:,0])  # losers
        μ_gs[:,0] = μ_sg[:,1] + μ_tg
        μ_gs[:,1] = μ_sg[:,0] - μ_tg

        yield (μ_s, np.sqrt(1/p_s))
```

Question (c)

Explain the concept of *convergence* for both the Gibbs sampler and the message passing algorithms. What type of object are we converging to in the two cases, and how do you judge convergence? For each of the players you plotted in (a), find the mean and standard deviation of their skill as found by the converged algorithm in (b), and annotate the plot from part (a) to show those values. Explain what you see.

Question (d)

Who is better out of `W[0]` Rafael Nadal and `W[15]` Novak Djokovic? There are several ways we could answer this.

- Using the Gibbs sampler (and taking account of burn-in and mixing time), first compute the marginal distribution of skill for each of these players, then work out the probability that Nadal's skill is higher.
- Using the Gibbs sampler, use joint samples from the Gibbs sampler to directly estimate the probability.
- Using the message passing algorithm, read off the distribution of skills of these two players, and work out the probability that Nadal's skill is higher.
- Repeat these three methods, but this time to work out the probability that Nadal would beat Djokovic in a match.

Calculate all these six probabilities. Explain how they relate to each other. Which answer is best? Why? Support your explanation with appropriate plots.

Appendix

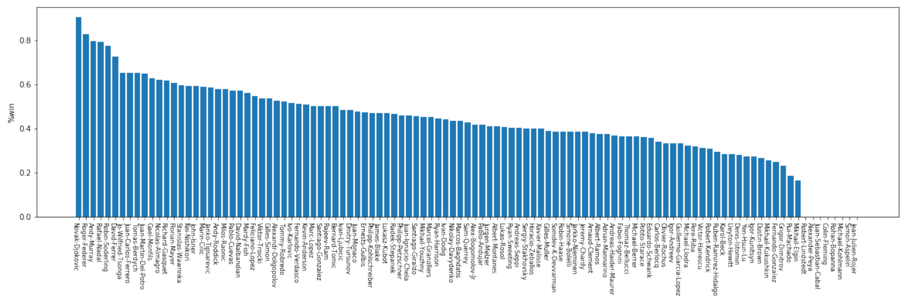
Data import

Here is a crude ranking of the players, based on a simple ratio

$$\frac{\text{number of wins for player } i}{\text{number of games played by player } i}.$$

```
wins = np.zeros(M, dtype=int)
losses = np.zeros(M, dtype=int)
w, x = np.unique(G[:,0], return_counts=True)
wins[w] = x
w, x = np.unique(G[:,1], return_counts=True)
losses[w] = x
score = wins / (wins + losses)
rank_order = np.argsort(score)[::-1]

with matplotlib.rc_context({'figure.figsize': [20,5], 'xtick.labelsize': 8}):
    x = np.arange(M)
    plt.bar(x, score[rank_order], align='center', width=.8)
    plt.xticks(x, W[rank_order], rotation=-90, ha='right')
    plt.ylabel('%win')
plt.show()
```



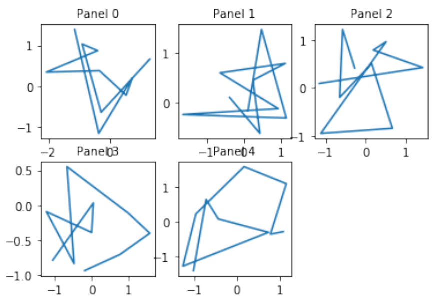
Question (a)

Some notes about the code in Question (a), and how it uses normally distributed random variables:

- If we just wanted to sample `t` given skill differences `s`, we could just use `t=np.random.normal(loc=s, scale=σ)`. But we actually want to generate `t` conditional on `t>0`. Using the inversion transformation method as mentioned in the notes, we end up with the code given here.
- We could also compute `μ` by defining `Rinv = scipy.linalg.cholesky(Σinv)` and then `μ=solve(Rinv, solve(Rinv.T, μtilde))`.
- We could also generate `w` by `w = μ + solve(Rinv, np.random.normal(size=M))`
- To generate a simple normal random variable with mean μ and variance σ^2, we can first generate $X \sim \text{Normal}(0,1)$ and then return $\mu + \sigma X$. To generate a multivariate normal random variable with mean vector μ and covariance matrix Σ, we first generate $X \sim \text{MVN}(0, I)$ and then return $\mu + RX$ where R is the matrix square root of Σ. It can be found from the Cholesky decomposition $RR^\top = \Sigma$.

To plot many panels in a single figure,

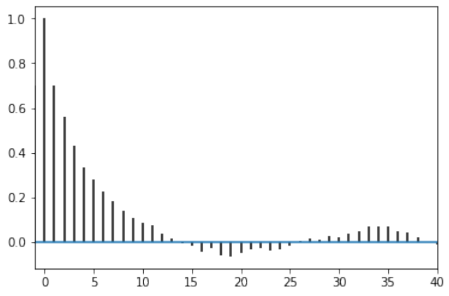
```
with matplotlib.rc_context({'figure.figsize': [6,4], 'axes.titlesize': 10}):
    fig = plt.figure()
    for i in range(5):
        ax = fig.add_subplot(2, 3, i+1)  # 2 rows of panels, 3 columns
        ax.plot(np.random.normal(size=10), np.random.normal(size=10))
        ax.set_title(f'Panel {i}')
```



To plot autocorrelations,

```
# Generate synthetic data with autocorrelations
x = [5, 5]
for _ in range(1000):
    lastx = x[-2:]
    nextx = np.mean(lastx) + 0.1 * np.random.normal() - 0.3 * (lastx[0] - 3)
    x.append(nextx)
x = np.array(x)

# Subtract the mean, then plot autocorrelation
plt.acorr(x - np.mean(x), maxlags=40)
plt.xlim(-1,40)
plt.show()
```



Question (b)

The `gaussian_ep` function includes a `yield` statement, which means it is a lazy list generator, and you can use it with e.g.

```
g = gaussian_ep(G, M)
for _ in range(10):
    print(next(g))
```

Question (d)

To evaluate $\mathbb{P}(\text{Normal}(\mu,\sigma^2) > x)$, you can use the appropriate scientific function directly,

```
scipy.stats.norm.sf(x, loc=μ, scale=σ)
```

or you can compute it using Monte Carlo integration,

```
np.mean(np.random.normal(size=10000, loc=μ, scale=σ) > x)
```

For this simple calculation there's no point in Monte Carlo integration, but the technique is useful in situations where there is no simple built-in scientific function.