Data Mining Assignment 8

Author: Lukas Gust

Due: April 24th

1 Finding q_*

Description:

We will consider four ways to find $q_* = M^t q_0$ as $t \to \infty$.

Matrix Power: Choose some large enough value t, and create M^t. Then apply $q_* = (M^t)q_0$. There are two ways to create M^t, first we can just let $M^{i+1} = M^i * M$, repeating this process $t-1$ times. Alternatively, (for simplicity assume t is a power of 2), then in $\log_2 t$ steps create $M^{2i} = M^i * M^i$.

State Propagation: Iterate $q_{i+1} = M * q_i$ for some large enough number t iterations.

Random Walk: Starting with a fixed state $q_0 = [0, 0, \ldots, 1, \ldots, 0, 0]^T$ where there is only a 1 at the ith entry, and then transition to a new state with only a 1 in the jth entry by choosing a new location proportional to the values in the ith column of M. Iterate this some large number t_0 steps to get state q_0'. (This is the burn in period.) Now make t new step starting at q_0' and record the location after each step. Keep track of how many times you have recorded each location and estimate q_* as the normalized version (recall $||q_*||_1 = 1$) of the vector of these counts.

Eigen-Analysis: Compute `eig(M)` and take the first eigenvector after it has been L_1 normalized.

A: Run each method (with $t = 1024$, $q_0 = [1, 0, 0, \ldots, 0]^T$ and $t_0 = 100$ when needed) and report the answers.

B: Rerun the *Matrix Power* and *State Propagation* techniques with $q_0 = [0.1, 0.1, \ldots, 0.1]^T$. For what value of t is required to get as close to the true answer as the older initial state?

C: Explain at least one *Pro* and one *Con* of each approach. The *Pro* should explain a situation when it is the best option to use. The *Con* should explain why another approach may be better for some situation.

D: Is the Markov chain *ergodic*? Explain why or why not.

E: Each matrix M row and column represents a node of the graph, label these from 1 to 10 starting from the top and from the left. What nodes can be reached from node 4 in one step, and with what probabilities?

Solution:

A:

Matrix Power:

```
def mat_pow(P, t, recursive=False):
    if recursive:
        if t == 0:
            return P
        else:
            Pi = mat_pow(P, t//2, recursive=recursive)
```

```
            return Pi @ Pi

    else:
        Pi = P
        for i in range(t):
            Pi = Pi @ P

        return Pi
```

Output:

```
[[ 0.05739691],
 [ 0.06696306],
 [ 0.08219028],
 [ 0.09539943],
 [ 0.13392613],
 [ 0.09173022],
 [ 0.15827633],
 [ 0.13392613],
 [ 0.10227086],
 [ 0.07792065]]
```

State Propagation:

```
def state_prop(P, t, q0):
    q_star = q0
    for i in range(t):
        q_star = P @ q_star

    return q_star
```

Output:

```
[[ 0.05739691],
 [ 0.06696306],
 [ 0.08219028],
 [ 0.09539943],
 [ 0.13392613],
 [ 0.09173022],
 [ 0.15827633],
 [ 0.13392613],
 [ 0.10227086],
 [ 0.07792065]]
```

Random Walk:

```
def rand_walk(P, t, t0, q0):
    # burn in
    q_star = q0.astype(int)
    for i in range(t0):
        W = np.sum(P[:,np.flatnonzero(q_star)])
```

```
        w = np.cumsum(P[:,np.flatnonzero(q_star)]/W)
        p = rand.uniform(0,1)
        k = np.argmax(w>p)

        q_star = np.zeros(q_star.shape, dtype=int)
        q_star[k] = 1

    # walk
    counts = np.zeros(q_star.shape)
    for j in range(t):
        W = np.sum(P[:,np.flatnonzero(q_star)])
        w = np.cumsum(P[:,np.flatnonzero(q_star)]/W)
        p = rand.uniform(0,1)
        k = np.argmax(w>p)
        counts[k] += 1

        q_star = np.zeros(q_star.shape, dtype=int)
        q_star[k] = 1


    return counts/np.linalg.norm(counts, ord=1)
```

Output:

```
[[ 0.06152344],
 [ 0.07128906],
 [ 0.06445312],
 [ 0.08300781],
 [ 0.11816406],
 [ 0.08691406],
 [ 0.17480469],
 [ 0.1484375 ],
 [ 0.10839844],
 [ 0.08300781]]
```

Eigen-Analysis:

```
w, v = np.linalg.eig(M)
qq = (v[:,0]/sum(v[:,0])).reshape((10,1)).astype(float)
qq
```

Output:

```
[[ 0.05739691],
 [ 0.06696306],
 [ 0.08219028],
 [ 0.09539943],
 [ 0.13392613],
 [ 0.09173022],
 [ 0.15827633],
 [ 0.13392613],
 [ 0.10227086],
 [ 0.07792065]]
```

B: We simply range t in powers of 2 and observe the difference between the result with the new q_* and the old q_*. For matrix power $t = 512$ is plenty large with a difference on the order of 10^{-15}. This is similar for state propagation but with a larger difference on the order of 10^{-12}. This is arguably close enough to the original while still being conservative.

C:

Matrix Power: Pro, fast and simple when used on a small graph like we did above. Con, expensive for large graphs, instead use the random walk or state propagation.

State Propagation: Pro, fast for large graphs since a matrix-vector multiplication is fast. Con, May need large number of iterations to converge, where as eigen-analysis would result in the converged state vector.

Random Walk: Pro, a state is chosen at each step through the random walk, so you can find out what the state might be for some number of steps even though it isn't the limiting state. Con, may not converge as fast and is random. Instead use state propagation.

Eigen-Analysis: Pro, we can evaluate how fast it will converge using the magnitude of the second eigenvalue. Con, expensive for large graphs, use state propagation or random walk instead.

D: The delicate balance property holds and this is a consequence of and ergodic graph. Also the probability transition matrix at every step t is positive. If it were not ergodic we might have zero entries in q_* or we might converge to a different state on a separate iteration.

E: We can simply run a few lines of code to find this or observe the probability transition matrix. We will do the latter. Note the 4th column of M in row form: `[0, 0, 0.3, 0, 0.3, 0.4, 0, 0, 0, 0]`. This tells us what nodes we can reach in one step starting at node 4. That is, we can reach Node 3 with $p = .3$, Node 5 with $p = .3$, and Node 6 with $p = .4$.

End