1 Derivation: PPR and its derivative

Notation: $\mathbf{s}$ is seeds, M is transition matrix, $\mathbf{w}$ are parameters, and $\mathbf{p}^\infty$ is the PPR stationary distribution, $\mathbf{p}^t$ is a point in the power-rule iteration for computating PPR, and $\mathbf{d}^t$ is the partial derivative of $\mathbf{p}^t$ wrt the parameters $\mathbf{w}$.

$$\mathbf{p}^{t+1} \equiv \alpha\mathbf{s} + (1-\alpha)\mathrm{M}^\top \mathbf{p}^t \quad (1)$$
$$\mathbf{d}^t \equiv \frac{\partial}{\partial \mathbf{w}}\mathbf{p}^t \quad (2)$$

Note $\mathbf{p}^t$ and $\mathbf{d}^t$ have different dimensions: while $\mathbf{p}^t_u$ is a scalar score for u under PPR, $\mathbf{d}^t_u$ is a vector, giving the sensitivity of that score to each parameter in $\mathbf{w}$.

M is defined as follows. There is a weight vector $\mathbf{w}$, and for an edge $u \to v$, there is a feature vector $\vec{\phi}_{uv}$, which is used to define a basic score s_{uv} for the edge, which is passed thru a squashing function f, e.g., $f(x) \equiv e^x$, and then normalized to form M.

$$s_{uv} \equiv \vec{\phi}_{uv} \cdot \mathbf{w} \quad (3)$$
$$t_u \equiv \sum_{v'} f(s_{uv'}) \quad (4)$$
$$\mathrm{M}_{u,v} \equiv \frac{f(s_{uv})}{t_u} \quad (5)$$

We can define $\mathbf{d}^t$ recursively:

$$\mathbf{d}^{t+1} = \frac{\partial}{\partial \mathbf{w}}\mathbf{p}^{t+1} \quad (6)$$
$$= \frac{\partial}{\partial \mathbf{w}}\left(\alpha\mathbf{s} + (1-\alpha)\mathrm{M}^\top\mathbf{p}^t\right) \quad (7)$$
$$= (1-\alpha)\frac{\partial}{\partial \mathbf{w}}\mathrm{M}^\top\mathbf{p}^t \quad (8)$$
$$= (1-\alpha)\left((\frac{\partial}{\partial \mathbf{w}}\mathrm{M}^\top)\mathbf{p}^t + \mathrm{M}^\top\frac{\partial}{\partial \mathbf{w}}\mathbf{p}^t\right) \quad (9)$$
$$= (1-\alpha)\left((\frac{\partial}{\partial \mathbf{w}}\mathrm{M}^\top)\mathbf{p}^t + \mathrm{M}^\top\mathbf{d}^t\right) \quad (10)$$

Now let's look at $\frac{\partial}{\partial \mathbf{w}}\mathrm{M}$, which I'll denote dM below. Note that each dM_{uv} is a vector, again giving the sensitivity of the weight M_{uv} to each parameter in $\mathbf{w}$.

$$\mathrm{dM}_{uv} = \frac{\partial}{\partial \mathbf{w}}\frac{f(s_{uv})}{t_u} \quad (11)$$
$$= \frac{1}{t_u^2}\left(t_u\frac{\partial}{\partial \mathbf{w}}f(s_{uv}) - f(s_{uv})\frac{\partial}{\partial \mathbf{w}}t_u\right) \quad (12)$$

To continue this, define **df** and **dt** as the vectors

$$\mathbf{df}_{uv} \equiv \frac{\partial}{\partial \mathbf{w}} f(s_{uv}) = f'(s_{uv})\vec{\phi}_{uv} \tag{13}$$

$$\mathbf{dt}_{u} \equiv \frac{\partial}{\partial \mathbf{w}} t_u = \sum_{v'} \mathbf{df}_{uv'} \tag{14}$$

Note that $\mathbf{df}_{uv}$ has no more non-zero components than $\vec{\phi}_{uv}$, so it is sparse, and $\mathbf{dt}_u$ is also sparse, but somewhat less so.

We can continue the derivation as

$$\mathrm{dM}_{uv} = \frac{1}{t_u^2}\left(t_u \frac{\partial}{\partial \mathbf{w}} f(s_{uv}) - f(s_{uv}) \frac{\partial}{\partial \mathbf{w}} t_u\right) \tag{15}$$

$$= \frac{1}{t_u^2}\left(t_u \mathbf{df}_{uv} - f(s_{uv})\mathbf{dt}_u\right) \tag{16}$$

This is a little different from the old algorithm: there is always an (implicit) reset with probability α, and that reset probability can be increased by learning by weighting the features for the (explicit) reset links that are present already in the graph. No min α!

2 Computation

Since dM and M are reused many times, we should compute and store M and dM first. This will expand the size of the graph somewhat: in particular, we will need to store, not only the active edges u, v and their features $\vec{\phi}_{uv}$, but also dM_{uv}, which includes weights for all features of vertexes v' that are siblings of v (i.e., there is an edge u, v'). I'm not sure how bad this will be in practice: perhaps we should estimate it for some of our test cases. After this, operations should run in time linear in the size of the new, less sparse graph encoded in dM. I believe that this scheme also makes operations like the APR learning more modular.

Computing M and dM is one pass over the graph, shown in Table 1.

Then you can start with $\mathbf{p}^0 = \mathbf{d}^0 = \mathbf{0}$ and iterate

$$\mathbf{p}^{t+1} = \alpha \mathbf{s} + (1-\alpha)\mathrm{M}^\top \mathbf{p}^t \tag{17}$$

$$\mathbf{d}^{t+1} = (1-\alpha)\left(\mathrm{dM}^\top \mathbf{p}^t + \mathrm{M}^\top \mathbf{d}^t\right) \tag{18}$$

In more detail, the iteration for the updates on **p** are shown in Table 2.

3 Loss functions and lazy regularization

For SRW each example is a triple $(\mathbf{s}, P, N)$ where **s** is the seed distribution, $P = \{a^1, \ldots, a^I\}$ are the positive (a-ok?) examples, and $N = \{b^1, \ldots, b^J\}$ are the negative (bad?) examples. I use **p** for the PPR distribution starting at **s**, and write $\mathbf{p}[u]$ for $\mathbf{p}_u$ if I run out of space for subscripts.

1. For each node/row u
    (a) $t_u = 0$
    (b) $\mathbf{dt}_u = \mathbf{0}$, an all-zeros vector
    (c) For each neighbor v of u
        i. $s_{uv} = \mathbf{w} \cdot \vec{\phi}_{uv}$, a scalar.
        In detail: For $i \in \vec{\phi}_{uv}$: increment s_{uv} by $\mathbf{w}_i \vec{\phi}_i$
        ii. $t_u \mathrel{+}= f(s_{uv})$, a scalar
        iii. $\mathbf{df}_{uv} = f'(s_{uv})\vec{\phi}_{uv}$, a vector, as sparse as $\vec{\phi}_{uv}$
        In detail: For $i \in \vec{\phi}_{uv}$: set $\mathbf{df}_{uv,i} = \vec{\phi}_f * c$, where $c = f'(s_{uv})$
        iv. $\mathbf{dt}_u \mathrel{+}= \mathbf{df}_{uv}$, a vector, as sparse as $\sum_{v'} \vec{\phi}_{uv'}$
        In detail: For $i \in \mathbf{df}_{uv}$: increment $\mathbf{dt}_{u,i}$ by $\mathbf{df}_{uv,i}$

    Now $t_u = \sum_{v'} f(s_{uv'})$ and $\mathbf{dt}_u = \sum_{v'} \mathbf{df}_{uv'}$
    (d) For each neighbor v of u create the vector

    $$\mathrm{dM}_{uv} = \frac{1}{t_u^2}\left(t_u \mathbf{df}_{uv} - f(s_{uv})\mathbf{dt}_u\right)$$

    Or in detail: For $i \in \mathbf{dt}_u$: $\mathrm{dM}_{uv,i} = \frac{1}{t_u^2}\left(t_u \mathbf{df}_{uv,i} - f(s_{uv})\mathbf{dt}_u, i\right)$.
    There aren't any dimensions i that are present in $\mathbf{df}_{uv}$ but not $\mathbf{dt}_u$, since $\mathbf{dt}_u$ is a summation. Also create the scalar

    $$\mathrm{M}_{uv} = \frac{f(s_{uv})}{t_u}$$

    You can now discard the intermediate values like t_u, $\mathbf{dt}_u$, $\mathbf{df}_{uv}$.

Table 1: Computing M and dM

```
Updating $\mathbf{p}$:

  1. $\mathbf{p}^{t+1} = \mathbf{0}$

  2. For each node $u$

     (a) $\mathbf{p}_u^{t+1} \mathrel{+}= \alpha \mathbf{s}_u$
     (b) For each neighbor $v$ of $u$
           i. $\mathbf{p}_v^{t+1} \mathrel{+}= (1-\alpha)\mathbf{p}_u^t \mathrm{M}_{uv}$

Updating $\mathbf{d}$:

  1. $\mathbf{d}^{t+1} = \langle \mathbf{0}, \ldots, \mathbf{0} \rangle$ — i.e., for each node $u$ there is an all-zeros vector of weights.

  2. For each node $u$

     (a) For each neighbor $v$ of $u$
           i. For each $i$ in $\mathrm{dM}_{vu}$

                $\mathbf{d}_{v,i}^{t+1} \mathrel{+}= (1-\alpha)\mathbf{p}_u^t \mathrm{dM}_{uv,i}$

          ii. For each $i$ in $\mathbf{d}_v^t$

                $\mathbf{d}_{v,i}^{t+1} \mathrel{+}= (1-\alpha)\mathbf{d}_{u,i}^t \mathrm{M}_{uv}$
```

Table 2: Updates for **d** and **p**

The loss function is

$$L(\mathbf{w}) \equiv -\left(\sum_{k=1}^{I} \log \mathbf{p}[a^k] + \sum_{k=1}^{J} \log(1 - \mathbf{p}[b^k]) \right) + \mu R(\mathbf{w}) \tag{19}$$

where $R(\mathbf{w})$ is the regularization, eg $R(\mathbf{w}) \equiv ||\mathbf{w}||_2^2$. This means loss increases as the objective function decreases, where the objective function is proportional to the probability of either hitting a positive solution or not-hitting a negative solution. Once we have $\mathbf{d}$ it's easy to compute the gradient of this as

$$\frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}) \equiv -\left(\sum_{k=1}^{I} \frac{1}{\mathbf{p}[a^k]} \mathbf{d}[a^k] - \sum_{k=1}^{J} \frac{1}{1 - \mathbf{p}[b^k]} \mathbf{d}[b^k] \right) + \mu \frac{\partial}{\partial \mathbf{w}} R(\mathbf{w}) \tag{20}$$

We can split this into two parts: the empirical loss gradient, which is

$$-\left(\sum_{k=1}^{I} \frac{1}{\mathbf{p}[a^k]} \mathbf{d}[a^k] - \sum_{k=1}^{J} \frac{1}{1 - \mathbf{p}[b^k]} \mathbf{d}[b^k] \right)$$

and the regularization gradient,

$$\mu \frac{\partial}{\partial \mathbf{w}} R(\mathbf{w})$$

If an example doesn't contain all features then the empirical gradient will be a sparse vector, but the regularization gradient will be dense. So the following code might be more efficient for SGD than just computation of the full gradient and taking a step in that direction.

1. Maintain a "clock" counter m which is incremented when each example is processed. Also maintain a history $\mathbf{h}_i$ which says, for each feature i, the last time t an example containing i was processed.
2. When a new example $(\mathbf{s}, P, N)$ arrives at time t
   (a) For each feature active in the example, initialize it, if necessary, and them perform the regularization-loss gradient update $t - \mathbf{h}_i$ times.
   (b) Peform the empirical-loss update, using the new weights.
3. When you finish learning at final time T, consider every feature i, and perform the regularization-loss gradient update $T - \mathbf{h}_i$ times. Then write out the final parameters.

4 Inference: PPR and APR

Inference using requires only M, which at theorem-proving time is computed on-the-fly. That is, whenever you need a row of M (the set of values $\mathrm{M}[u, v]$ for all v near u) you simply compute the normalized outlinks of u.

The power-iteration version of PPR, which in the codebase is called the `PPRProver`, simply iterates this step until convergence (or for a fixed number of iterations), starting with $\mathbf{p}^0 = \mathbf{0}$.

$$\mathbf{p}^{t+1} \equiv \alpha\mathbf{s} + (1-\alpha)\mathrm{M}\mathbf{p}^{t} \tag{21}$$
$$\tag{22}$$

Breaking this down, the one-step update is the following.

1. $\mathbf{p}^{t+1} = \mathbf{0}$
2. For each key u with non-zero weight in $\mathbf{s}$:
   (a) $\mathbf{p}^{t+1}[u]$ += $\alpha\mathbf{s}[u]$
3. For each key u with non-zero weight in $\mathbf{p}^t$:
   (a) For each node v near u in M: $\mathbf{p}^{t+1}[v]$ += $(1-\alpha)\mathrm{M}[u,v]\mathbf{p}^t[u]$

The approximate PageRank is based on a more primitive `push` operation. The full approximate PageRank algorithm also uses $\mathbf{d}$, a vector of node degrees.

1. Let $\mathbf{p} = \mathbf{0}$ and $\mathbf{r} = \mathbf{s}$
2. While there is some vertex u such that $r(u) \geq \varepsilon\mathbf{d}(u)$:
   (a) Perform the `push`(u) operation:
      i. Save the current value of u in $\mathbf{r}$: $ru = \mathbf{r}[u]$
      ii. $\mathbf{p}[u]$ += $\alpha\mathbf{r}[u]$
      iii. $\mathbf{r}[u]$ *= $(1-\alpha)$
      iv. For each node v near u in M: $\mathbf{r}[v]$ += $(1-\alpha)\mathrm{M}[u,v]ru$

In the old implementation, $\mathbf{d}[u]$ was computed and cached each time a new key u was added to $\mathbf{r}$. In the new implementation of the prover, this caching is done by the proof graph, so there's no need to do it in the prover.

The old DPR prover implementation computed this approximation by finding u's with large $\mathbf{r}[u]$ using a variation of the standard depth-first prover: essentially, you traversed the tree depth-first, and whenever you hit a node with $\mathbf{r}[u]$ below threshold, you stopped. Factoring this in the algorithm used the same initial values of $\mathbf{p}$ and $\mathbf{r}$ but then called a recursive routine `proveState`(u_0), where u_0 is the initial query node (aka the initial state of the proof graph).

William Wang's got a new version which calls this iteratively with smaller and smaller ε's, which seems to work faster. That suggests that it's helpful to do pushes on the nodes with larger $\mathbf{r}$ values first. With that in mind, another thing we could consider would be storing u's in a heap/priority queue, ordered by $\mathbf{r}$ values. I know this idea has been used in the past but I don't know how much it helps. Obviously there's an overhead for the heap but my guess is that's not going to dominate.

Function `proveState`(u):

1. If $r(u) \geq \varepsilon\mathbf{d}(u)$:
   (a) Perform the `push`(u) operation
   (b) For each v near u (i.e., each subgoal of u): `proveState`(v)

5 Architectural Comments

[Katie's comments inline and signed with -k]
The routines that are suggested are:

- Loading: Compute M and dM from a graph and **w**. It seems necessary, altho a little non-modular, to initialize and create new features while the graph is read in. *[Do this here, or in SGD? -k]* We should be able to estimate performance pretty well after this step.

  Inputs: graph and current values of parameters **w**.

  Outputs: M and dM, and a list of all active features i.

  Parameters: f, f'; method to init new features i for f, clock time and lazy-regularization function.

- Extended inference: compute **p** and **d**.

  Inputs: **s**, **w**, M and dM.

  Outputs: **p** and **d**.

- SGD: Initialize new features i, use clock time and lazy-regularization function to update their weights, and update **w** using the empirical (non-gradient) loss.

  Inputs: **p**, **d**, and P, N.

  Outputs: Modified **w**.

A proposed data structure which would be very efficient is shown in Table 3 and 4. Some things that aren't clear now are:

- Where does the feature-index $\leftrightarrow$ feature-name symbol table go? Probably not in the graph as shown here.

- Is M and dM part of the graph or a different structure?

Also a clarification: `M` and `dM_lo,dM_hi`, are not dense 2-matrixes, of size quadratic in the number of nodes, n. Instead `M` is a length-n array of variable-size arrays, and `M[u]`, for node index `u`, is an array of size m_u, where m_u is the number of neighbors (edges away from) `u`. And `dM_lo,dM_hi` are parallel structures.

A final note: the squashing function f and its derivative f' are only used in computing **df** in Equation 13. One future extension to ProPPR might be to have facts with weights defined by parameters: e.g., facts like `sim(a,b)` which when used would lead to an edge with a computed weight s_{uv} based on an internal set of parameters $\vec{\lambda}$ (think of a learned similarity subroutine). A modular way of dealing with this might be worth thinking through. Conceptually, $\vec{\lambda}$ is a part of **w**: one could have an interface which given an edge u,v with weight s_{uv} returns $f'(s_{uv})\frac{\partial}{\partial \mathbf{w}}s_{uv}$ in terms of components of $\vec{\lambda}$. For training you would also need to pass in an update of the $\vec{\lambda}$ features to the learning subroutine....

```
class Graph {

  // names of features

  String[] featName;

  // space for labels for feature weights on edges, each is a
  // feature index and a weight

  int[] label_featIndex;
  double[] label_featWeight;

  // space for edges, each is a destination node and a
  // list of labels, which point into the label space

  int[] edge_dst;
  int[] edge_labels_lo;
  int[] edge_labels_hi;

  // space for list of neighbors of nodes, each
  // points into the edge space

  int [] node_near_lo;
  int [] node_near_hi;

  // list of nodes in graph
  int node_lo, node_hi;

  // space for weight derivatives, which are structurally
  // just like labels.  these maybe should be vectors since
  // we don't know the size of this space in advance...?

  int[] deriv_featIndex;
  double[] deriv_featDeriv;

  // list of derivative features for M[u][euv]
  int[][] dM_lo;
  int[][] dM_hi;
}
```

Table 3: Proposed data structure for graph

```
for (int u = g.node_lo; u < g.node_hi; u++) {
  // euv is edge from u to v
  for (euv = g.node_near_lo[u]; euv < g.node_near_hi[u]; euv++) {
    int v = g.edge_dst[euv];
    // luvk is k-th label on edge euv
    for (luvk = g.edge_labels_lo[euv]; luvk = g.edge_labels_hi[euv]; luvk++) {
      int i = g.label_featIndex[luvk];
      double w = g.label_featWeight[luvk];
      ...
    }
  }
}
for (int d = g.dM_lo[u][euv]; d < g.dM_hi[u][euv]; d++) {
  i = g.deriv_featIndex[d];
  double v = g.deriv_featDeriv[d];
  // v is partial/(partial feature i) if M[u][v]
  ...
}
```

Table 4: Accessing the proposed data structure for graph

A Linear Algebra proof of imperative procedure for p and d updates

A.1 Computing the p update

$$\begin{aligned}
\mathbf{p}^{t+1} =& \alpha\mathbf{s} + (1-\alpha)\mathbf{p}^t\mathrm{M} \\
\mathbf{p}^t\mathrm{M} =& \begin{bmatrix} p_0 & p_1 & p_2 & \cdots & p_n \end{bmatrix} \begin{bmatrix} M_{00} & \mathrm{M}_{01} & \mathrm{M}_{02} & \cdots \\ M_{10} & \mathrm{M}_{11} & \mathrm{M}_{12} & \cdots \\ \mathrm{M}_{20} & \mathrm{M}_{21} & \mathrm{M}_{22} & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix} \\
=& \begin{bmatrix} p_0^{t+1} & p_1^{t+1} & p_2^{t+1} & \cdots \end{bmatrix} \\
p_0^{t+1} =& p_0\mathrm{M}_{00} + p_1\mathrm{M}_{10} + p_2\mathrm{M}_{20} + \ldots \\
p_1^{t+1} =& p_0\mathrm{M}_{01} + p_1\mathrm{M}_{11} + p_2\mathrm{M}_{21} + \ldots
\end{aligned}$$

If we set p_u^{t+1}, we must iterate across, over all terms of the first element, then all terms of the second element, etc:

$$p_u^{t+1} += (1-\alpha)p_v\mathrm{M}_{vu}$$

Given our data structure for M however, constructing M_{vu} is difficult. To use M_{uv} instead, we can set p_v^{t+1} and iterate down, computing the first term of all elements, then the second term of all elements, etc:

$$p_v^{t+1} += (1-\alpha)p_u\mathrm{M}_{uv}$$

A.2 Computing the d update

Computing d proceeds in much the same way:

$$\mathbf{d}_i^{t+1} = (1-\alpha)(\mathbf{p}^t\mathrm{dM}_i + \mathbf{d}_i^t\mathrm{M})$$

We'll start by deconstructing the first term only:

$$
\begin{aligned}
\mathbf{p}^t \mathrm{dM}_i = &\left[\begin{array}{llll} p_0 & p_1 & p_2 & \dots \end{array}\right] \left[\begin{array}{llll} \mathrm{dM}_{00,i} & \mathrm{dM}_{01,i} & \mathrm{dM}_{02,i} & \dots \\ \mathrm{dM}_{10,i} & \mathrm{dM}_{11,i} & \mathrm{dM}_{12,i} & \dots \\ \mathrm{dM}_{20,i} & \mathrm{dM}_{21,i} & \mathrm{dM}_{22,i} & \dots \\ \vdots & \vdots & \vdots & \end{array}\right] \\
= &\left[\begin{array}{llll} d_{0,i}^{t+1} & d_{1,i}^{t+1} & d_{2,i}^{t+1} & \dots \end{array}\right] \\
d_{0,i}^{t+1} = &p_0 \mathrm{dM}_{00,i} + p_1 \mathrm{dM}_{10,i} + p_2 \mathrm{dM}_{20,i} + \dots \\
p_{1,i}^{t+1} = &p_0 \mathrm{dM}_{01,i} + p_1 \mathrm{dM}_{11,i} + p_2 \mathrm{dM}_{21,i} + \dots
\end{aligned}
$$

We have the same choice of setting $d_{u,i}^{t+1}$ or $d_{v,i}^{t+1}$, with the same dependence on using $\mathrm{dM}_{vu,i}$ or $\mathrm{dM}_{uv,i}$. Since we must use $\mathrm{dM}_{uv,i}$ due to the way our data structure is constructed, we must update $d_{v,i}^{t+1}$:

$$d_{v,i}^{t+1} += (1-\alpha)\mathbf{p}_u^t \mathrm{dM}_{uv,i}$$

The same principles hold for the second term:

$$d_{v,i}^{t+1} += (1-\alpha)\mathbf{d}_{u,i}^t \mathrm{M}_{uv}$$