Algorithms for Smart broadcasting
Recap: *When-to-post* problem setup

**Broadcasters’ posts as a counting process** $N(t)$

- $N_1(t)$
- $N_2(t)$
- $\vdots$
- $N_n(t)$

**Users’ feeds as sum of counting processes** $M(t)$

- $M_1(t) = A^T N(t)$
- $M_2(t)$
- $\vdots$
- $M_n(t)$
Recap: Measuring Visibility

Position of the highest ranked tweet by broadcaster i in follower j’s wall

\[ r_{ij}(t) = 0 \]

\[ r_{ij}(t') = 4 \]

\[ r_{ij}(t'') = 0 \]

- \[ M(t) \] represents the visibility measure over time.
- \[ t \] is the time axis.
- Older tweets are indicated by a down arrow.
- Posts by broadcaster u are highlighted in green.
- Posts by other broadcasters are highlighted in red.
Recap: Maximizing Visibility

Minimize (quadratic) loss:

$$\text{minimize} \int_0^T \left( r^2(t) + c \lambda^2(t) \right) dt$$

Maximize time spent at the top:

$$\text{maximize} \int_0^T \mathbb{I}(r(t) < 1) dt$$

s.t. $$\int_0^T \lambda(t) \leq C$$
Today: Evaluating broadcasting strategies

Task: Implementing strategies

Metrics:
- Average rank
- Time at the top

Simulated
Working of the simulator

Simulation_options = SimOpt(  
    src_id=0,  
    end_time=100,  
    other_sources=[  
        'Poisson',  
        {  
            'src_id': 1,  
            'seed': 10016,  
            'rate': 1.0  
        }  
    ],  
    sink_ids=[1000],  
    edge_list=[(0, 1000), (1, 1000)]  
)
How to implement a strategy?

class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
How to **implement** a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
```

```python
class Event:
    def __init__(self, event_id, time_delta, cur_time, src_id, sink_ids, metadata=None):
        self.event_id = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time = cur_time
        self.src_id = src_id
        self.sink_ids = sink_ids
        self.metadata = metadata
```

```python
event = Event(event_id=1, time_delta=5, cur_time=5, src_id=0, sink_ids=[1000])
```
How to implement a strategy?

class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()

class Event:
    def __init__(self, event_id, time_delta, cur_time, src_id, sink_ids, metadata=None):
        self.event_id = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time = cur_time
        self.src_id = src_id
        self.sink_ids = sink_ids
        self.metadata = metadata

event = Event(event_id=2, time_delta=2.5, cur_time=7.5, src_id=1, sink_ids=[1000])
How to implement a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()

class Event:
    def __init__(self, event_id, time_delta, cur_time,
                 src_id, sink_ids, metadata=None):
        self.event_id = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time = cur_time
        self.src_id = src_id
        self.sink_ids = sink_ids
        self.metadata = metadata
```

```
event = Event(event_id=2, time_delta=2.5, cur_time=7.5, src_id=1, sink_ids=[1000])
```

**Updated!**
How to **implement** a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()
```

event = Event(event_id=2, time_delta=2.5, cur_time=7.5, src_id=1, sink_ids=[1000])

```
class Event:
    def __init__(self, event_id, time_delta, cur_time,
                 src_id, sink_ids, metadata=None):
        self.event_id = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time = cur_time
        self.src_id = src_id
        self.sink_ids = sink_ids
        self.metadata = metadata
```

$\Delta t$ always measured from last self post.
How to implement a strategy?

```python
class Broadcaster:
    # ...
    def get_next_interval(self, event):
        raise NotImplementedError()

class Event:
    def __init__(self, event_id, time_delta, cur_time,
                 src_id, sink_ids, metadata=None):
        self.event_id = event_id
        self.time_delta = time_delta  # Since last event
        self.cur_time = cur_time
        self.src_id = src_id
        self.sink_ids = sink_ids
        self.metadata = metadata
```

```
event = Event(event_id=3, time_delta=7.5, cur_time=15,
              src_id=1, sink_ids=[1000])
```

Repeats to the end.
Broadcasting Strategies

1. **Poisson**
   \[ \lambda(t) = \mu \]

2. **Hawkes**
   \[ \lambda(t) = \mu + \alpha \sum_{t_i \in \mathcal{H}(t)} \exp(-\beta(t - t_i)) \]

3. **RedQueen**
   \[ \lambda(t) = c \, r(t) \]

4. **Smart Poisson**
   \[ \lambda(t) = \mu \, \mathbb{1}(r(t) > 0) \]

Already implemented.

This lecture.

To implement.
Poisson broadcaster: already implemented

\[ \lambda(t) = \mu \]

```python
class Poisson(Broadcaster):
    def __init__(self, src_id, seed, rate=1.0):
        super(Poisson, self).__init__(src_id, seed)
        self.rate = rate

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None or event.src_id == self.src_id:
            # Draw a new time, one event at a time
            scale = 1.0 / self.rate
            return RS.exponential(scale)
```
Poisson broadcaster: already implemented

\[ \lambda(t) = \mu \]

Use `self.random_state` for repeatable experiments and debugging.

```python
class Poisson(Broadcaster):
    def __init__(self, src_id, seed, rate=1.0):
        super(Poisson, self).__init__(src_id, seed)
        self.rate = rate

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None or event.src_id == self.src_id:
            # Draw a new time, one event at a time
            scale = 1.0 / self.rate
            return RS.exponential(rate)
```

If this is the beginning of the simulation or if the post was by this broadcaster, return a new sample.

No else branch: not returning a value means *do not* change old time.
Hawkes broadcaster: another example

\[ \lambda(t) = \mu + \alpha \sum_{t_i \in \mathcal{H}(t)} \exp(-\beta(t - t_i)) \]

```python
class Hawkes(Broadcaster):
    def __init__(self, src_id, seed, l_0=1.0, alpha=1.0, beta=1.0):
        super(Hawkes, self).__init__(src_id, seed)
        self.l_0 = l_0
        self.alpha = alpha
        self.beta = beta
        self.prev_excitations = []

    def get_rate(self, t):
        # Returns the rate of current Hawkes at time `t`.
        return self.l_0 + \n            self.alpha * sum(np.exp([-1.0 * (t - s) for s in self.prev_excitations if s <= t]))

    def get_next_interval(self, event):
        t = self.get_current_time(event)
        RS = self.random_state
        if event in None or event.src_id == self.src_id:
            rate_bound = self.get_rate(t)

            # Ogata sampling for one t-delta
            while True:
                t_delta = RS.exponential(scale=1.0 / rate_bound)

                # Rejection sampling
                if RS.rand() < rate_bound:
                    break
                else:
                    t += t_delta

            self.prev_excitations.append(t + t_delta)
            return t + t_delta - self.get_current_time(event)
```

- Initializing and saving parameters
- Calculating \( \lambda(t^+) \)
- Ignore unless it was our event or the 1st event
- Ogata’s thinning algorithm
- Return \( \Delta t \) from our own post
Recap: RedQueen broadcaster

\[ \lambda(t) = cr(t) \]

- Minimizes loss: \( \int_0^T (r^2(t) + c \lambda^2(t)) \, dt \)
- For the task: \( c = \sqrt{\frac{s}{q}} = 1 \)

- Sampling using Superposition:

\[
\begin{align*}
\Delta_i &\sim \exp(c) \\
&t_1 + \Delta_1 \\
&t_2 + \Delta_2 \\
&t_3 + \Delta_3 \\
&t_4 + \Delta_4 \\
&\min_i t_i + \Delta_i
\end{align*}
\]
RedQueen Broadcaster implementation

\[ \lambda(t) = cr(t) \]

```python
class Opt(Broadcaster):
    def __init__(self, src_id, seed, q=1.0, s=1.0):
        super(Opt, self).__init__(src_id, seed)
        # ...
        self.rank = 0

    def get_next_interval(self, event):
        if event is None:
            # Tweet immediately if this is the first event.
            self.rank = 0
            return 0

        elif event.src_id == self.src_id:
            # No need to tweet if we are on top of all walls
            self.rank = 0
            return np.inf

        else:
            # Calculate current rank
            self.rank = self.rank + 1
            # cur_time = self.get_current_time(event)

            # t_delta = ...
            # if ...:
            #     return t_delta
```
RedQueen Broadcaster implementation

\[ \lambda(t) = c r(t) \]

```python
class Opt(Broadcaster):
    def __init__(self, src_id, seed, q=1.0, s=1.0):
        super(Opt, self).__init__(src_id, seed)
        #
        self.rank = 0

    def get_next_interval(self, event):
        if event is None:
            # Tweet immediately if this is the first event.
            self.rank = 0
            return 0
        elif event.src_id == self.src_id:
            # No need to tweet if we are on top of all walls
            self.rank = 0
            return np.inf
        else:
            # Calculate current rank
            self.rank = self.rank + 1
            # cur_time = self.get_current_time(event)

            # t_delta = ...
            # if ...:
            #     return t_delta
```

This is how rank evolves.
RedQueen Broadcaster implementation

\[ \lambda(t) = cr(t) \]

class Opt(Broadcaster):
    def __init__(self, src_id, seed, q=1.0, s=1.0):
        super(Opt, self).__init__(src_id, seed)
        # ...
        self.rank = 0

    def get_next_interval(self, event):
        if event is None:
            # Tweet immediately if this is the first event.
            self.rank = 0
            return 0

        elif event.src_id == self.src_id:
            # No need to tweet if we are on top of all walls
            self.rank = 0
            return np.inf

        else:
            # Calculate current rank
            self.rank = self.rank + 1
            # cur_time = self.get_current_time(event)

            # t_delta = ...
            # if ...:
            #     return t_delta

Return infinite if we do not plan to post.
\[ \lambda(t) = c r(t) \]

class Opt(Broadcaster):
    def __init__(self, src_id, seed, q=1.0, s=1.0):
        super(Opt, self).__init__(src_id, seed)
        # ...
        self.rank = 0

    def get_next_interval(self, event):
        if event is None:
            # Tweet immediately if this is the first event.
            self.rank = 0
            return 0
        elif event.src_id == self.src_id:
            # No need to tweet if we are on top of all walls
            self.rank = 0
            return np.inf
        else:
            # Calculate current rank
            self.rank = self.rank + 1
            # cur_time = self.get_current_time(event)

            # t_delta = ...
            # if ...
            #     return t_delta

\[ \Delta_i \sim \exp(1.0) \]
\[ \min_i t_i + \Delta_i \]
Smarter than Poisson Broadcaster

\[ \lambda(t) = \mu \mathbb{1}(r(t) > 0) \]

**Class SmartPoisson(Broadcaster):**

```
"""Like the Poisson Broadcaster, but does not post if already on top."""

def __init__(self, src_id, seed, rate=1.0):
    super(SmartPoisson, self).__init__(src_id, seed)
    self.is_dynamic = True
    self.rate = rate
    self.on_top = False

def get_next_interval(self, event):
    RS = self.random_state
    if event is None:
        return NotImplemented
    elif event.src_id == self.src_id:
        self.on_top = True
        return np.inf
    elif self.on_top:
        # If we are no longer on top, schedule a post.
        self.on_top = False
        return NotImplemented
```

**Heuristic to improve time at top:**

- Do not post if already on top.
- If not on top, then post at a steady pace to let bursts of others’ posts pass (e.g., breaking news).
- Contrast: always maintaining low rank.
Smarter than Poisson Broadcaster

\[ \lambda(t) = \mu \mathbb{I}(r(t) > 0) \]

```python
class SmartPoisson(Broadcaster):
    """Like the Poisson Broadcaster,
    but does not post if already on top."""
    def __init__(self, src_id, seed, rate=1.0):
        super(SmartPoisson, self).__init__(src_id, seed)
        self.is_dynamic = True
        self.rate = rate
        self.on_top = False
    def get_next_interval(self, event):
        RS = self.random_state
        if event is None:
            return NotImplemented
        elif event.src_id == self.src_id:
            self.on_top = True
            return np.inf
        elif self.on_top:
            # If we are no longer on top, schedule a post.
            self.on_top = False
            return NotImplemented
```

Using a flag to figure out if on top or not.
Smarter than Poisson Broadcaster

\[ \lambda(t) = \mu \mathbb{I}(r(t) > 0) \]

```python
class SmartPoisson(Broadcaster):
    """Like the Poisson Broadcaster,
    but does not post if already on top."""
    def __init__(self, src_id, seed, rate=1.0):
        super(SmartPoisson, self).__init__(src_id, seed)
        self.is_dynamic = True
        self.rate = rate
        self.on_top = False

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None:
            return NotImplementedError
        elif event.src_id == self.src_id:
            self.on_top = True
            return np.inf
        elif self.on_top:
            # If we are no longer on top, schedule a post.
            self.on_top = False
            return NotImplementedError

    def broadcast(self, message):
        if self.is_dynamic:
            interval = self.get_next_interval(None)
            # ... broadcast logic ...
```

Return infinite if we do not plan to post.
Smarter than Poisson Broadcaster

\[ \lambda(t) = \mu \mathbb{1}(r(t) > 0) \]

```python
class SmartPoisson(Broadcaster):
    """Like the Poisson Broadcaster,
    but does not post if already on top."""

    def __init__(self, src_id, seed, rate=1.0):
        super(SmartPoisson, self).__init__(src_id, seed)
        self.is_dynamic = True
        self.rate = rate
        self.on_top = False

    def get_next_interval(self, event):
        RS = self.random_state
        if event is None:
            return NotImplemented
        elif event.src_id == self.src_id:
            self.on_top = True
            return np.inf
        elif self.on_top:
            # If we are no longer on top, schedule a post.
            self.on_top = False
            return NotImplemented
```

To be implemented.
Live Coding

- Show execution of simulation
- Diagnostic plots
- Evaluation metrics
## Evaluation

<table>
<thead>
<tr>
<th>Metric</th>
<th>RedQueen</th>
<th>Smart Poisson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-1</td>
<td>57 ± 3</td>
<td>58 ± 4</td>
</tr>
<tr>
<td>Average rank</td>
<td>59 ± 6</td>
<td>67 ± 10</td>
</tr>
<tr>
<td>Number of posts</td>
<td>61 ± 3</td>
<td>62 ± 4</td>
</tr>
</tbody>
</table>
Happy coding!

Questions?

- Drop me an e-mail at utkarshu@mpi-sws.org
- Skype: utkarsh.upadhyay