An Investigation Into The Pinterest Recommendation System

Written entirely by
Sreetam Ganguly
sreetamneverrests@gmail.com

Dated: October 10, 2020

Pinterest is a social media website which is made up of three building blocks: the users, the boards and the pins. Pins are individual posts made or uploaded by users. Boards are collections of pins, which may be compiled manually by individual users. A user can be defined as a human operating or making the use of the website or application. A user can “like” or “view” pins and boards. They are also permitted to “follow” boards, which essentially means that the concerned user will see more pins in the board in the future. The entire Pinterest system can be viewed as a bipartite graph

$$G = (P, B, E)^2$$

where,

P is the set of pins
B is the set of boards
E is the set of bidirectional edges that connect pins with boards.

An edge E_{ij} connects pin P_i and board B_j only if the board B_j has the pin P_i in its collection. Furthermore, there are further connections between users and boards and pins. Since a connection between a user and a board is essentially a connection between a user and many pins, these connections can be broken down to connections between users and pins. This can be represented as

$$Q_d = \{(p_i, w_i) \mid i \in N,\ i < k\}$$

Where Q_d is the set of ordered pairs of pins and their “weights”, also called the set of “query pins”. The weight of a pin represents how much the user likes the pin. It can be calculated based on the actions the user takes when he or she sees the pin (likes, shares or reports). The weights decay with a half-life of λ.

When a user starts using the app, he or she starts following a few boards and liking a few pins. From those actions, this set is compiled. Q_d is the set for a user *d*. In general, for the rest of this document, this set will be denoted by Q. This is because the underlying function of the recommendation system (discussed next paragraph onwards) works irrespective of user.

The recommendation system of Pinterest works by taking into account the previous likes and dislikes of the user. The system accomplishes this by re-evaluating the set Q at a fixed frequency. After taking into account the set Q, the system recommends the user pins which the user might like. The system is non-deterministic, fast, works in constant time complexity and is scalable, which makes it ideal for a pool of a large number of users and content. The system can recommend an assorted set of items to each of its 10^{11} users from a set of 10^{12} items, in real-time.

The recommendation system is a modified Random Walk algorithm. The system iterates through the set Q. For each pin p in Q, it determines which pins might be most relevant to the user. The system does this by "traversing" to a board which is linked to the pin and is most likely to be related to the user's interests, and then again "traversing" to another pin connected to the board, having those same qualities. The algorithm, named "*Pixie*" is explained below.

A1. PixieRandomWalk (q: Query pin, E: Set of edges, U : User personalization features, α : Real, N : Int, n_p : Int, n_v : Int):

1. totSteps = 0, $V = 0$
2. nHighVisited = 0
3. **Repeat**
4. currPin = q
5. currSteps = **SampleWalkLength**(α)
6. **for** i = [1 : currSteps] **do**
7. currBoard = E(currPin)[**PersonalizedNeighbor**(E,U)]
8. currPin = E(currBoard)[**PersonalizedNeighbor**(E,U)]
9. V [currPin]++
10. **if** V [currPin] == n_v **then**
11. nHighVisited++
12. totSteps += currSteps
13. **until** $totSteps \geq N$ or $nHighvisited > n_p$
14. **return** V

A2. PixieRandomWalkMultiple(Q: Query pins, W: Set of weights for query pins, E: Set of edges, U: User personalization features, α : Real, N: Int)

1. **for all** $q \in Q$ **do**
2. $s_q = |E(q)| .(C - log|E(q)|)$ //E(q) is the degree if the pin q
3. $N_q = w_q N \frac{s_q}{\sum_{r \in Q} s_r}$ // w_q is the weight of the pin q in Q
4. V_q = **PixieRandomWalk**(q, E, U, α, N_q)
5. **for all** $p \in G$ **do**
6. $V[p] = \sum_{q \in Q} (\sqrt{V_q[p]})^2$
7. **return** V

The algorithm shortens the random path taken by the algorithm by scaling it sublinearly, taking into account the degree of a node (a pin in this case) (line 2 and 3 in A2). This has the advantage of reducing the number of steps taken for the random walk, thus decreasing the time and memory complexity. The sublinear scaling also prevents high degree nodes from getting recommended too often, while making it sure that nodes with smaller degrees get recommended. Nodes with higher degrees are generally extremely popular or “viral” pins. Nodes with smaller degrees are generally pins with low to medium popularity.

The system then does a random walk by calling the PixieRandomWalk algorithm (line 4 in A2). The PixieRandomWalk algorithm is described in A1. It moves a certain distance in its random walk. The distance is derived as described in the previous paragraph. During its random walk, the number of times each node gets visited is recorded (line 9 in A1). After a node reaches a certain upper limit, it increases a counter nHighVisited (line 10 and 11 in A1). When the counter nHighVisited reaches a certain limit, the loop is terminated. This aids in the convergence of the algorithm. When the number of recommended pins reaches a certain limit, the process stops. This is necessary since the efficiency of this algorithm is critically dependent on the assumption that a stable recommendation is reached in the smallest possible number of steps.

The PersonalizedNeighbor function (line 7 and line 8 in A1) returns the pin that is most relevant to the user. This results in more customizable and more personalized recommendations.

The array V (line 1, 9 and 10 in A1) is a visit counter, that stores the number of visits by the random walk for each pin or node. It is an array with a key-value pair.

The system then records the number of hits each pin gets and then scales it by considering the number of hits received from unique pins (line 6 in A2). Multiple visits from multiple query pins are thus preferred, rather than multiple visits from the same query pin.

The two most important features that must be implemented for this algorithm to run is the bipartite graph G and the visit counter V. The bipartite graph G can be implemented as a sparse adjacency matrix. Every node is first assigned a unique ID between 1 and |P U B|. Each node is associated with a list of its neighbours. Allocating such a large amount of memory dynamically is slow and might lead to data leakage. So, the adjacency matrix is concatenated into a continuous vector. The visit counter, which is a key-value pair, can be implemented as an open addressing hash table with linear probing. The values are visit counts and the keys are unique IDs of pins described in the adjacency matrix. The graph edges must be pruned in such a way that boards with several diverse pins are not considered, or considered with less weightage. This will decrease the time and memory complexity of the algorithm. This will also prevent boards with too many different pins from monopolizing the recommendations of users. This can be achieved by calculating the entropy of each board using topic vectors. Boards with high entropy may be removed from the graph.

This approach can give balanced and personalized recommendations to each user, in very less time. The process is scalable. However, the machines running the algorithm must have large

amounts of RAM and cache. If that criterion is met, this algorithm is a novel and simple approach. The algorithm must be implemented in Object-Oriented Programming (OOP) Languages like C++, Python or Java to keep scalability and maintenance practical and simple. C++ is the preferred choice in the implementation of this algorithm since it provides a balance of being fast and providing OOP support.

References:

Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, Jure Leskovec