SNSRouter – A Framework for Intelligent Message Routing on Heterogeneous SNS *

Pili Hu †
Information Engineering
The Chinese University of Hongkong
Shatin, NT, Hongkong
hupili@ie.cuhk.edu.hk

ABSTRACT

Online Social Networks (OSN) has become an essential part of human life today. They also share a very large portion of Internet traffic in terms of Page Visits. Successful OSNs are like Facebook, Twitter, Renren, SinaWeibo, etc. Two of the basic functions are: 1) maintain connections with people; 2) read/write updated information in a networked fashion.

In this paper, we focus on the function of **information acquisition and dissemination**. We build a middleware for heterogeneous Social Network Services (SNS), thus enabling very easy cross-platform message manipulation. Besides those traditional OSNs, we can also abstract RSS, Email, SQLite and more other platforms in the same way. With this middleware, SNSAPI [6], many novel applications can be built and they are ready to be extended to other platforms.

We observe that message forwarding is a popular operation on SNS and there are many manual cross-platform forwarding behaviours. SNSRouter [5] project aims at developing an easier way to perform cross-platform message forwarding (routing). We propose to leverage combined human and machine intelligence. More specifically, we propose a Rank Preserving Regression (RPR) formulation to personalize incoming messages' ordering. Users with certain programming knowledge can easily add other personalization features. An appropriate ordering saves user's time in reviewing those messages. He/she can then make final forwarding decisions efficiently.

Evaluation on real traces collected by SNSRouter shows that RPR outputs very good weighting even with elementary feature extractions. The author of the report, who is not an expert in NLP, can construct those features in one day. This shows the flexibility of our system and algorithm framework.

Categories and Subject Descriptors

H.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software; M.5 [**KNOWLEDGE PERSONALIZATION AND CUSTOMIZATION**]: Framework

General Terms

System, Algorithm

Keywords

Social Network Services, Middleware, Personalization, Rank Preserving Regression

1. INTRODUCTION

Social Network Services mimic the real life social networks using modern technology. They come in many different forms. In this section, we first introduce different Social Network Services and provide a unified view of them. Then we introduce cross-platform message forwarding (routing), which motivates the SNSRouter project.

1.1 Heterogeneous Social Network Services

There are many Social Networking Services (SNS) nowadays. **Fig 1** illustrates some of them. If we focus on information acquisition and dissemination in a networked fashion, they can all be casted in the same way:

- Online Social Networks (OSN). Examples are like Facebook, Twitter, Renren, SinaWeibo, TecentWeibo in the figure. People form links (either directed or undirected) first and information can flow on those links.
- Communication Networks. Examples are like telephone network, SMS network and Email network. On those network, links do not need to be established beforehand. Sender can specify receiver(s) explicitly on the fly.
- Information Sources. Examples are like RSS feeds of blogs and keywords on search engines. When one subscribe to RSS feeds or write a script to monitor the response of a certain keyword search engine, a unidirectional link is established from the information sources to him/her.

*Source codes of the paper and project can be found in the following Github repository. This repository is continuously evolving. `https://github.com/hupili/sns-router`

†`http://personal.ie.cuhk.edu.hk/~hpl011/`

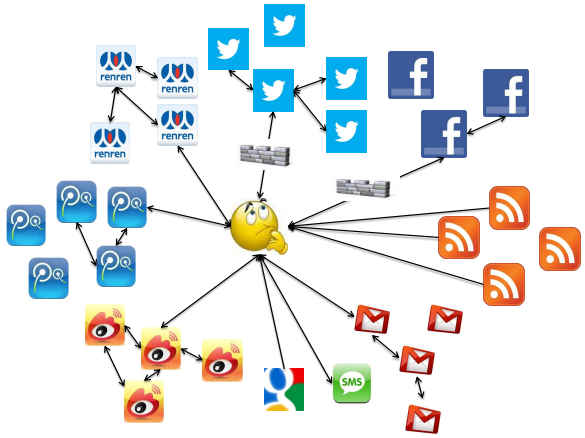


Figure 1: Illustration of Heterogeneous SNS

As one can see, the only differences are: 1) whether links are formed beforehand; 2) whether links are directed or undirected; 3) whether read/write are allowed or only read (or write) is allowed; 4) whether user authentication/authorization are required. Currently, there is no efficient way to abstract all these platforms, so we see people visiting several SNS's everyday.

1.2 Cross Platform Message Forwarding

Information dissemination in a single domain is natural. Many platforms have built-in "forward" functions, e.g. "retweet" on twitter, "forward" in email, etc. Many researches have also been done for information dissemination on a single platform. However, cross-platform behavior is a seldom discussed topic. We believe this is largely due to lack of mature engineering models and large collection of data sets. Nevertheless, we observe many manual cross-platform behaviours. For example:

- You subscribe to a blog RSS and read a gossip on it. The gossip is interesting so you copy and paste it to Renren.
- You receive Google Scholar updates on a certain topic by email. Once there are some eye-catching new papers, you forward to your colleagues by email.
- Now iPhone 5 is a big event and you want to get latest news about it. Instead of following "inbridge" at "twitter", you can follow "iphone+5" at "baidu" (by automatically poll the link: `http://www.baidu.com/s?ie=utf-8&wd=iphone+5`).

Note the roles people play in the above examples. We know Internet routers connect different IP subnet. Analogously, those people are routers across different SNSs. The routing job is done manually at preset. How about making it automated or semi-automated? Can we learn one's preference and use machine intelligence to assist the routing?

1.3 Organization

SNSRouter [6] is built on a prior work – SNSAPI [6]. In order to make this document self-contained, we will also include SNSAPI in the system design section.

In **Section 2** , we briefly survey related work. In **Section 3** , we discuss design philosophy of SNSAPI and SNSRouter. In **Section 4** , we present the system framework. In **Section 5** , we formulate the personalization problem as Ranking Preserving Regression. In **Section 6** , we transform the problem and propose to use Stochastic Gradient Descent (SGD) as the training algorithm. In **Section 7** we evaluate all aspects of our algorithm framework: efficiency, accuracy, robustness, and flexibility. Last, we conclude this paper in **Section 8** and share our visions on future works in **Section 9** .

Users who want to get hands on quickly can skip the those system and algorithm discussions and consult the executive summary in Appendix C. When time is permitted, we encourage you to come back and read those discussions to get a better picture so that you can full utilize the power of SNSRouter.

2. RELATED WORK

From the system aspect, some Internet resource aggregation services are related. From the algorithm aspect, some personalization services are related. In this section, we briefly survey those services.

2.1 Aggregation Services

There are many aggregation services on the Internet. Some of them focus on dealing with a certain type of sources, e.g. Google Reader for RSS feeds. Some of them can aggregate heterogeneous sources, like ifttt [7] and Yahoo Pipes [8].

ifttt abstracts services on the Internet by "channel" (we borrow this term in SNSAPI), e.g. Facebook, Twitter, RSS, SMS, etc. Users can define forwarding rules (called "recipe" on ifttt) in this manner: IF This happens, Then do That (IFTTT). This service is very handy for normal users who simply want to bridge several services, e.g. publish status on Facebook and the message is automatically forwarded to Twitter. ifttt supports a wide spectrum of platforms, but there is little flexibility for user to build advanced rules.

Yahoo Pipes (YP), on the contrary, supports less platforms but allows somewhat advanced rules. YP focus on the traditional web, so operations like fetching webpage, RSS feeds and CVS table are supported. In the pipe construction UI, users can drag sources and logic modules (e.g. condition, loop, etc) to the design board and connect them using "pipe". In the Web1.0 world, YP could be a good aggregator. However in the web2.0 world, User Generated Content (UGC) is dominating and it is full of noise. Simple constructions provided by YP is far from enough. e.g. forward a certain topic RSS entries to somewhere else.

2.2 Personalization with Machine Intelligence

One major problem of those SNS's is that there are too many messages flowing throughout the network. Even the data rate within one's personal view is too large for human to process efficiently. Besides filtering out noise, one also want to prioritize the display of messages so that messages can be processed by human in descending order of their importance. Different users have different preference so personalization is needed. Towards this end, many service providers also develop their own machine learning algorithms to help users personalize the incoming messages.

Personalized incoming message ranking is of its own research interest even in the case of single platform. For ex-

ample, SinaWeibo by default presents incoming messages in reverse chronological order. Users can enable ranked view and obtain a probably more informative timeline. For another example, GMail classifies incoming messages into "important" and "ordinary" types so that users can process "important" messages first.

Existing approaches focus more on developing better algorithms. Training data is obtained mostly by implicit feedback, e.g. how long a user stays on one message. User engagement in the training cycle is very low. Thus the degree of personalization is also low. Besides, those service providers target mass audience and provide no flexibility for people who understands how to program. For example, one may suffer spontaneously from someone's overwhelming messages. The common practice is to block his/her messages (not unlink the user). However, service providers usually set an upperbound on the number users blocked, e.g. 5 for Renren and 15 for SinaWeibo. One needs to upgrade his/her account to paid member in order to grow the upperbound. Anyone who knows programming agrees that blocking specified users is a very simple task given proper programming interface. To avoid blocking spontaneous noise sources permanently, one can also set some timeout logic or probabilistic blocking logic. However, all of those simple operations are not possible in current services.

On one hand, existing personalization approaches try to solve harder than necessary problems with little user side knowledge. On the other hand, some simple personalization demand can not be implemented by the users.

3. PHILOSOPHY

In this section, we discuss the design philosophy of our system framework and algorithm framework. In a word, we pursue simple but effective approaches which leverage human and machine intelligence simultaneously.

- **Focus on solving 80% problems**. The Pareto Principle [9] tells us that (qualitatively) one only needs 20% effort to solve 80% problems; In order to solve the rest 20% problems, he/she needs to spend the other 80% time. Example: conceptually, we only abstract three methods for OSN, namely **read**, **write** and **auth**. The three methods can cover more than 80% functions one need on an OSN. They are also universal across many other SNS. However, in order to abstract many platform specific functions, we need much more time, but those functions may be seldom used in our cross-platform settings.

- **Keep It Simple and Stupid (KISS)** [10]. Whenever possible, we prefer simpler solutions. Common and hard cut rules will be implemented directly with configurable interface. e.g. one can configure some channels to be read-only and others to be write-only directly. We do not need to "learn" such preference through user interaction. This principle also applies to our algorithm framework design, i.e. When simpler features are effective enough, we do not consider advanced features; When linear combination is good enough, we do not bother to look at non-linear combinations.

- **Combine human and machine intelligence**. In **Section 2** we discussed the current approaches adopted by both industry and research communities. Machine learning algorithms are asked to do much harder tasks than necessary. For one extreme example, one can put all bits of a message (sender, text, time, etc) into one very long vector as feature. If we have a very powerful machine learning algorithm, it may be able to learn what factor is important to personalize one's messages and rank them reasonably. Such algorithm, even exists, will be very costly. However, the user speaks louder than any statistical clues. If one user always prefer longer messages than shorter message, he/she just put the length of message into the feature vector and that is enough. We will discuss more in the formulation section.

- **Stay open with existing services**. We never want to solve a full stack of problem. Many problems are already solved (probably in a degraded way). We encourage users to take advantage of existing services rather than reinventing wheels. Example: I want to forward important messages from OSN to my cellphone through SMS. I will not try hard to enable "SMS platform" on SNSAPI. Instead, I will build the intelligence which filers out "important" messages using SNSAPI and then output those messages to an RSS file. Later on ifttt, I just add a straight forwarding recipe from the RSS file to my cellphone through SMS.

- **Trade execution efficiency for developing efficiency**. Many of the codes in this project are not optimized. There is no incentive to optimize them unless the execution time is out of human tolerable range. We prefer to make the codes extensible rather than running fast.

4. SYSTEM FRAMEWORK

The system is divided into two parts:

- SNSAPI is the middleware to interface with heterogeneous SNS's. Other developers can write "plugin" to enable new platform. Applications built on SNSAPI is readily available for future new platforms.

- SNSRouter Front End (SRFE) is the web-based UI for SNSRouter. Users can accomplish authorization, view incoming messages, post messages, and forward messages using SRFE. If ranking algorithm is enabled in the configuration, ranked timeline is also presented in SRFE. SRFE can be run wherever Python is supported.

4.1 SNSAPI Architecture

Fig 2 illustrates the system framework of SNSAPI. There are conceptually three layers:

- Interface Layer. SNSBase is the base class for all kinds of SNS. One can derive the base class to implement real logic that interfaces with those platforms. In SNSAPI terminology, the modules containing derived classes are called "plugin"; the derived classes are called "platform"; the instance of the classes are called "channel". Message and message list types are also defined in this layer. We make all the interfacing types json-serializable and pickle-able, so the communications inside and outside SNSAPI are very easy.

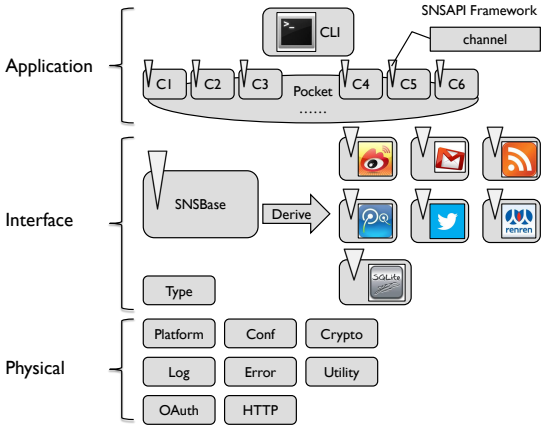


Figure 2: SNSAPI Architecture

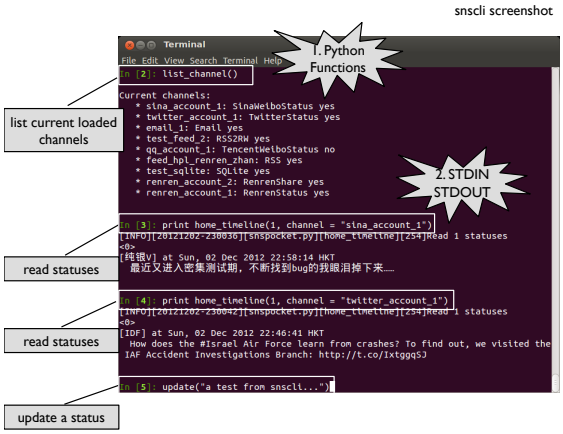


Figure 3: Command Line Interface of SNSAPI

- Physical Layer. There are many common operations when one interfaces with different SNS's. We implement them in the Physical layer so that plugin authors do not need to build all the logics from scratch. Examples are like, HTTP request/response, OAuth, Error definition, etc. Many third party modules are used in this layer. To keep flexibilty, we provide wrapper class for most third party modules, so that one can subsitute (part of) them with better ones easily.

- Application Layer. Application authors can directly use classes in the interface layer. e.g. write an auto-reply script by using RenrenStatus (derived from SNSBase); only a dozen of lines are needed. However, when one wants to operate a large collection of channels, this is not efficient. To avoid frequently repeated work, we develop a container class called "SNSPocket". Pocket can hold a collection of channels and perform batch operation with some simple configurable rules. For most applications, Pocket should be the Service Access Point (SAP) to SNSAPI. When there are new channels, users can enable them by configuration and no active involvement from the App developer is needed.

In the Application Layer, we also include one Command Line Interface (CLI) – "SNSCLI". The reason comes in three folds. First, SNSCLI is more appropriate for initial trials. It is run in a Python interactive shell and usage is also natural to non-Python users. Second, for App developer, SNSCLI can be used for debugging purpose. Third, if non-Python developers want to use SNSAPI, SNSCLI can serve as the SAP. Commands are piped in SNSCLI using STDIN and results are obtained from STDOUT. **Fig 3** is the screenshot of a running SNSCLI.

We remark that the above is just conceptual illustration of the SNSAPI framework. File level details are avoided in this document as they are distraction from our main topic. Interested readers can go to our project repository [6] and consult the developer's wiki.

4.2 SNSRouter Architecture

In SNSAPI terminology, SNSRouter is an Application built upon it. As we are Python developers, we choose to use Pocket class as the SAP to SNSAPI. The system part, i.e. SRFE, can be divided into three parts:

- Web UI. At the start time of SRFE, we face different choices of UI: desktop software or web services. We choose web UI because we want SRFE able to be accessed from any platforms. Another consideration is that the web UI is easy to be extended so that it provides RESTful API to downflow applications. To build SRFE, we need HTTP request handler, URL routing, and simple template engine. There are many micro frameworks available in Python like "cherrypy", "flask", and "bottle". We choose bottle mainly because of its single file property. This makes it easy to distribute along with SNSRouter so that minimum dependencies are required.

- Message Queue. If Web UI interfaces with SNSAPI directly, the response cycle will be very long. This is because SNSAPI's application programming model is synchronous, e.g. pocket invokes certain method of all channels and wait for response one by one. We believe this it's the App developers job to choose multi-threading or multi-processing solutions. In SRFE, we implement an asynchronous message queue, which is running in the backend. It polls each channel periodically and store messages in a local SQLite database. Request from Web UI is routed to the Message Queue so that response cycle is shortened.

- Ranking. Ranking can be enabled or disabled by configuration. Ranking module has interface with Queue. When enabled, incoming messages get their scores. From the Web UI, users can reach ranked timeline by asking the Message Queue to sort messages in descending order of score. Feature extraction and auto weighting (learner) logic is implemented in Ranking module.

Fig 4(a) shows a screen shot of ordinary timeline (reverse chronological order). Users can mark a message as "seen" so that it won't appear later in the timeline. "seen" flag will be an important part in the training data later. Below each message is a set of user defined tags. Different users can have different criterion and interpretation of those tags. Below tags is the forwarding panel. Users can add his/her own comments and forward the new message to all other platforms. This is further configurable (e.g. which tag to forward to which platform). **Fig 8(c)** shows the screen shot

of ranked timeline at the same time. The author's opinion on those messages are depicted in those screenshots. From this pair of screenshots, we can see that ranked version is much cleaner so that user can make forwarding decisions more efficiently (Quantitative measures are presented in the evaluation section).

5. FORMULATION

In this section, we discuss how we formulate a personalized ranking problem. Bear in mind that our ultimate goal is to make cross-platform message routing efficient. Now the engineering problem has been solved by SNSAPI and SRFE. Next is to design proper machine learning algorithms to make this done automatically. In this section, we first discuss the impossibility of complete auto forwarding algorithms. Then we discuss the problems arise from classification formulation. Last, we propose a regression model with user preference constraints.

5.1 Impossibility of Auto Forwarding

The most aggressive goal of SNSRouter is to make forwarding decisions automatically. However, after one month trial, we argue that this is impossible **in general**. By analyzing the real traces (167 forwarded messages out of 7.5K "seen" messages), we can qualitatively conclude that the forward decision is made by considering the following factors:

- Quality. This usually applies to knowledge sharing and discussion.
- Novelty (freshness). This usually applies to news.
- Authority. This usually applies to news.
- Degree of wisdom. This usually applies to twisdom (tweet-wisdom: refer to those short but philosophy enlightening pieces)

Although people have different tastes, the former three factors may be captured by the machine. However, when one looks into the traces, there are more messages of the same (even higher) quality, novelty, and authority. Why doesn't the user forward them? The user's consideration is beyond those factors, like "I have already forwarded 5 messages today; I don't want to flood my friend; This message is good but I'm not going to forward it". Other examples are like temporary shift of interest. Those factors are hard to capture for machines.

Another issue is that forward operation is sparse among all other operations. The author is devoted to this project and mark training data every day, so 167 out of 7.5K forwarded messages are collected after one month. This rate is much higher than ordinary usage of SNS's. Towards this end, it is very easy for one algorithm to overfit the training samples. For example, one algorithm will find the following piece of knowledge statistically significant: "Only messages originated during daytime is valuable enough for Pili to forward". Obviously, this is not the case. I will miss many good messages posted on Twitter by someone at the other side of the earth! The reason why this rule is statistically significant is that I do not have a ranking engine to bring night messages to the front in the initial several weeks.

We argue that complete automatic forwarding decision is impossible in general. Human users must review the messages and decide whether to forward. Nevertheless, we can develop some assistant tools to help human make decision more efficiently.

5.2 The Problem of Classification Formulation

Suppose we have N messages $m_1, m_2, \ldots, m_N$. We can extract some features of the messages somehow: (sample features are discussed in evaluation section)

$$X^{\mathrm{T}} = [x_1, x_2, \ldots, x_N] \tag{1}$$

where $x_i \in \mathbb{R}^K$. Besides, from the Web UI, users can assign tags to each message. The relation between messages and tags is multiple-to-multiple. For simplicity of discussion, let's assume each message has at most one tag (technically, we can make this true by creating auxiliary tags no matter what is the user's interpretation of the main tags). For those messages seen by the user by not tagged, we assign a "null" tag for it. Then every seen message has exactly one tag. We denote the tag of message i by t_i. Our first formulation is a classification problem. That is, given X, try to predict t_i using certain classifiers. If this can be done reasonably, users can make decisions based on predicted tags and the efficiency is improved.

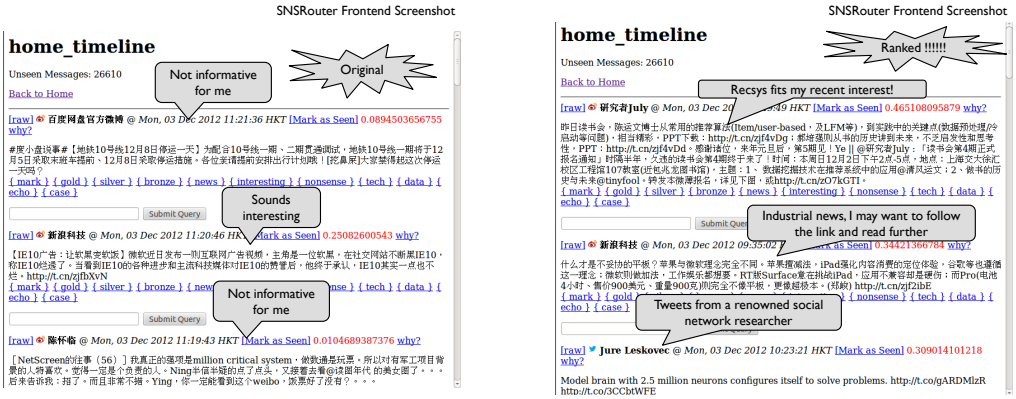
We dumped data into arff format and tried two commonly used classifiers using Weka [11]: Logit Classifier and J48 decision tree. In the experiments, we use Weka's default parameters and avoid any tuning. This is because we don't want our ranking algorithm to be the game of machine learning researchers. The user's knowledge in machine learning only affects what features he can extract but not the quality a learner is trained.

The accuracy of Logit classifier is only 78.5526%, which is not competitive. The overall accuracy of J48 is 87.3684% and is comparable to our later proposals. Although the accuracy is high, the rules generate by J48 is very complex, and the user can assert that they are "incorrect" by a glance. We present more data in **Appendix B**, e.g. the confusion matrix, sample branch of the tree, etc.

Besides the above algorithmic concerns, classification formulation has other problems:

- Hard cut rule. Classifiers set hard cut rule for each class. They usually do not output a likelihood or any equivalence. This cause problem when there is one message lie close to "news" and "interesting" in the feature space, but not close enough to any one of them. This message will be treated as "null" by nature. However it could be both new and interesting. Although intermediate stage of those classifiers may calculate such values (e.g. posterior in Naive Bayes, "likelihood" in Logit, etc), tapping into the already packaged algorithm is needed.
- Human can only process sequentially. In the last subsection, we argued that human has to be the last judge no matter what algorithm is used. If human can only process sequentially, the combined machine+human decision process is also sequential. There is no need to perform very accurate classification. According to our philosophy, we should avoid solving this more than necessary problem. On the contrary, if we can order the messages reasonably, this is already good enough for human decision stage.

Based on these observations, we proceed to propose a regression formulation.



(a) Original Home Timeline on SRFE

(b) Ranked Home Timeline on SRFE

Figure 4: Original v.s. Ranked Timeline. Screen shot taken at around Dec 3, 11:30am

5.3 Regression Formulation

Now we have extracted features for messages, X. Our goal is to rank them properly. Bear in mind that this is not a pure offline research experiment. New messages are coming in continuously and users can request timeline from Web UI at any time at any frequency (subject to human's Actions Per Minute). We don't want to rerank all the messages every time when new ones come in and return the top messages to the user. We want to make the query stage efficient. One simple solution is to assign a score to a message when it comes in, and store the score as a column in database. In the query stage, one only need to add an "ORDER BY" clause to enable ranking. This is standard in database practice and can be done very efficiently.

How to map features to a single score? Our first thought is of course linear combination, i.e. Xw. For one thing, non linear combination can be casted to linear combination by extending the dimension of features. For another thing, output of linear combination is highly interpretable, e.g. "+10 if the message comes from Pili", "+5 if the message is about social computing", etc. When talking about "enough" features, we do not mean the more the better. We indeed mean to incorporate more domain knowledge, which is possessed by the user (not the most clever machine learning researcher). Then we get the following regression formulation:

$$\underset{w}{\text{minimize}} \qquad ||y - Xw||_2^2 \tag{2}$$

where y_i is the score for message i. If we can find a set of weights w that combines to y very well on training data, it is reasonable to expect they perform similarly well on future incoming messages. Then our ranking flow is: 1) New message comes in; 2) Extract features; 3) Use learned w to generate a score, $\hat{y}_i = x_i^{\mathrm{T}} w$; 4) Store $\hat{y}_i$ in DB; 5) In query stage, sort messages by $\hat{y}_i$.

One direct difficulty is that y is unknown and it is very hard for users to generate training y's consistently. Another problem is that when user's interest shifts, he/she has to regrade the messages and train the weights again. This regrading process can be tedious for a user. Step back a little, we think it is reasonable to ask users to tell their preference between two messages. This leads to the following formulation:

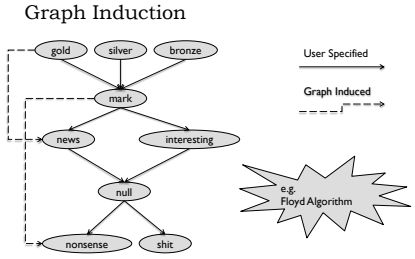


Figure 5: Graph Induction of Preference Constraint

$$\underset{y,w}{\text{minimize}} \qquad ||y - Xw||_2^2 \tag{3}$$

$$\text{s.t.} \qquad y_i > y_j,\ \forall (m_i, m_j) \in E \tag{4}$$

where E is a set of tuples (i, j) meaning message i is preferred than j. Note that y in this formulation is unknown variable. This makes it different from ordinary regression. Besides, E is only a partial ordering, so this is not ordinal regression problem. After some literature survey, we did not find similar standard problems. Towards this end, we will temporarily term this formulation as **Rank Preserving Regression (RPR)**. Details on how to obtain E is described in **Section 6.1**.

6. ALGORITHM DESIGN

In this section, we design the algorithm to train the RPR model. We first discuss how the preference constraints E are obtained. Then we discuss how to tackle with the optimization problem. Besides algorithmic considerations, there are also some practical considerations. These observations lead to the final choice – Stochastic Gradient Descent.

6.1 Induce Preference Relations on Graph

Fig 5 illustrates how preference constraints E are derived. Ovals stand for user defined tags. Solid arrows are user specified preference (through a json configuration file). Note

that no matter what is the meaning of the tags for a user, he/she should be able to tell which tag is preferred. One can just ask himself/herself some questions. If I only have 1 minute today, what type of messages do I want to view? If I have 1 more minute, what other types do I want to view next? In this way, users can specify some partial ordering of the tags.

We denote $T_u \succeq T_v$ if tag T_u is preferred than T_v. We construct the tag graph $G_T =< T, P >$, where T is the set of tags and $P = \{(T_u, T_v) | T_u \succeq T_v\}$. On this directed graph, we can induce more partial orderings. More concretely, if there exists a path from T_x to T_y, we know $T_x \succeq T_y$. Since the tag graph is so small, we simply invoke Floyd algorithm [2] to induce those path preferences without customizing on our own.

Now we collect all path preferences in one set P_I (meaning Induced Preference). Note that edge can be treated as length-1 path, so $P \subset P_I$. Then we can define message graph $G_M =< M, E >$, where $M = \{m_1, m_2, \ldots, m_N\}$ is the set of all messages and $E = \{(m_i, m_j) | (t_i, t_j) \in P_I\}$ encode a partial ordering of the messages.

We remark that this tag based preference induction is flexible. It provides different level of granularities. If one only needs coarse-grained ordering, he only needs to add two tags like "good" and "bad". One can also provide a set of fine-grained tags in a nested manner. e.g. for those tagged as "good", he can further tag "machine learning", "social computing", etc. At first, when there is only a little training samples, he can configure only one user-specified preference, i.e. "good" $\succeq$ "bad". Later when more training samples are available, he can add another line of configuration: "machine learning" $\succeq$ "social computing". If his interest shifts one day, he simply change this line to: "social computing" $\succeq$ "machine learning".

6.2 Straight Optimization Solver

The RPR model is in essence a Quadratic Programming (QP). There are many existing solvers and we can find some Python implementations (e.g. cvxpy which solves a superset of problems). However, the QP approach has the following problems:

- QP solver introduces more dependencies in our project, making it more difficult to port to other platforms.
- Solving QP is costly, if one want to run the algorithm on mobile devices. This is true for both deploying dependent libraries and executing the algorithm.
- Plain QP implementation only supports batch optimization. In our problem, m_i comes continuously and E is also evolving. If we want to capture time variant user interest, we must solve QP in an incremental fashion.

With those practical considerations, we seek for better training approaches by proper problem transformation.

6.3 Problem Transformation

We first convert the inequality preference constraints into equality constraints using indicator function:

$$\underset{y,w}{\text{minimize}} \quad ||y - Xw||_2^2 \tag{5}$$

$$\text{s.t.} \quad I[y_i > y_j] = 1,\ \forall (m_i, m_j) \in E \tag{6}$$

where $I[.]$ is the indicator function:

$$I[x] = \begin{cases} 1 & \text{x is True} \\ 0 & \text{x is False} \end{cases} \tag{7}$$

Since $I[.]$ only takes 0 or 1, the previous optimization is equivalent to the following form:

$$\underset{y,w}{\text{minimize}} \quad ||y - Xw||_2^2 \tag{8}$$

$$\text{s.t.} \quad 1 - I[y_i > y_j] \leq 0,\ \forall (m_i, m_j) \in E \tag{9}$$

The Lagrangian of this problem is:

$$L(y, w, \mu) = ||y - Xw||_2^2 + \sum_{k=1}^{|E|} \mu_k (1 - I[y_i > y_j]) \tag{10}$$

We can lowerbound the optimal value of the original problem by solving the Lagrangian dual problem:

$$v_d^* = \sup_{\mu \geq 0} \inf_{y,w} L(y, w, \mu) \tag{11}$$

The point $(\bar{y}, \bar{w}, \bar{\mu})$ which attains the v_d^* MAY be approximately an optimizer for the original problem. Note that this point does not necessarily be primal optimizer because primal feasibility may be violated and Lagrangian duality gap is not zero in general.

To solve the Lagrangian dual problem is still hard. Instead of exploring the whole space of (y, w, μ), we restrict ourselves to a family of (y, w, μ), which is easier to optimize. We impose the following additional constraints:

$$y = Xw \tag{12}$$

$$\mu_i = \lambda, \forall i = 1, 2, \ldots, |E| \tag{13}$$

$$\lambda \geq 0 \tag{14}$$

Then we have:

$$\tilde{L}(y, w, \mu) = \lambda \sum_{k=1}^{|E|} (1 - I[y_i > y_j]) \tag{15}$$

The supinf problem of **Eq 11** becomes the following optimization:

$$\underset{y,w}{\text{minimize}} \quad \sum_{(m_i, m_j) \in E} 1 - I[y_i > y_j] \tag{16}$$

$$\text{s.t.} \quad y = Xw \tag{17}$$

Note that the constraint can be plugged back to objective, so the objective is essentially a function in w and the whole problem is an unconstrained programming. To make the notation less cluttered, we keep the current form. To this point, we reduce variables from $(N + K)$ to K, which is significant in practice. Then it is possible to leverage first-order optimization methods, which are usually easy to implement.

6.4 Gradient Descent

The standard practice to deal with indicator function is approximation by Sigmoid function:

$$S(x) = \frac{1}{1 + e^{-\beta x}} \tag{18}$$

where we add a scaling factor β to control the approximation rate. As α goes larger and larger, $S(x)$ approximates $I(x)$ better. Now we define

$$f(w) \equiv \sum_{(i,j) \in E} 1 - S(y_i - y_j) \tag{19}$$

where $y_i = (Xw)_i$ and $(i,j) \in E$ is a shorthand notation for $(m_i, m_j) \in E$.

The gradient of $f(w)$ is:

$$\nabla f(w) = \sum_{(i,j)\in E} \nabla f_{ij}(w) \tag{20}$$

where

$$\nabla f_{ij}(w) = \beta(1 - S(y_i - y_j))S(y_i - y_j)(x_j - x_i) \tag{21}$$

Then we can use standard Gradient Descent (GD) to minimize $f(w)$.

6.5 Stochastic Gradient Descent

One key observation is that the full gradient $\nabla f(w)$ is the summation of per pair partial gradients $\nabla f_{ij}(w)$. To perform one step of GD, one needs to sum over $|E| = O(N^2)$ terms. This is very costly when N goes large. Luckily, this summation structure fits Stochastic Gradient Descent (SGD) [12] very well. Then we can sample a set of preference relations $E_s \subset E$ with $|E_s| \ll |E|$ and compute a "stochastic" gradient:

$$\nabla f_{E_s}(w) = \sum_{(i,j)\in E_s} \nabla f_{ij}(w) \tag{22}$$

We walk along $-\nabla f_{E_s}(w)$ for certain step size α. Probabilistically, SGD has similar effect to GD. This method is quite popular in the recent years' development of recommender systems. From our experience, it is much easier to set step size in SGD than GD.

In one extreme, we can implement SGD with $|E_s| = 1$. Our implementation of SGD can be summarized in **Alg 1**.

Algorithm 1 RPR Training Using SGD

Input: features X, initial weights w^0,
step size α, number of rounds R
Output: final weights: w^R

```
k = 1
while k ≤ R do
    Sample a pair of preference relation (i, j)
    g_ij ← ∇f_ij(w^{k−1})
    w^k ← w^{k−1} − α g_ij
    k ← k + 1
end while
```

7. EVALUATION

In this section, we evaluate our RPR-SGD proposal. We first introduce the data set collected in a time span slightly more than one month. Then we propose to use Kendall's tau correlation coefficient as a metric. Performance v.s. complexity of the algorithm is evaluated with a set of preliminary features. We also use real feedback to show that user efficiency is significantly improved. Next, we inject noise feature to show the robustness of RPR-SGD. Last, we use a case study to demonstrate how easy it is to incorporate a new feature and show the adaptive learning behaviour of RPR-SGD.

7.1 Pili's Data Set

Pili has been running SRFE for over one month to collect real traces. **Tbl 1** summarizes some basic statistics. We derive training and testing preference relations in the following way: 1) Select out all tagged messages (about 900); 2) Sample same number of untagged messages (about 900) and assign them "null" tag; 3) Divide the candidate messages (about 1.8K) into training and testing sets randomly; 4) Use graph induction to form $G_{M_{\text{train}}}$ and $G_{M_{\text{test}}}$, where M_{train} is training set and M_{test} is testing set. Altogether, there are more than 200K relations in both training and testing sets.

Table 1: Basic Statistics of Pili's Data Set

Item	Value
# of total messages	32533
# of seen messages	7553
# of tagged messages	924
# of forwarded messages	167
# of derived pairs (training)	231540
# of derived pairs (testing)	229009
# of features (+1 noise)	15

7.2 Criterion

Kendall's tau correlation coefficient [3] is a good measure of rank. The modified version for our problem is defined as:

$$K = \frac{\sum_{(i,j)\in E_{\text{test}}} I[\hat{y}_i > \hat{y}_j] - \sum_{(i,j)\in E_{\text{test}}} I[\hat{y}_j > \hat{y}_i]}{|E_{\text{test}}|} \tag{23}$$

where $\hat{y}_i$ is the predicted score of message i using the learned weights w and E_{test} is the set of edges in $G_{M_{\text{test}}}$. The first summation in numerator counts the number pairs whose relation is preserved. The second summation in numerator counts the number pairs whose relation is violated. The denominator is the total number of relations in our concern. Given this definition, we know that $K \in [-1, 1]$ and the larger the better. This is in accordance with the notion of correlation. When $\hat{y}_i$'s are assigned randomly, we can expect $K = 0$. In the following discussions, we also call this measure Kendall's score for short.

7.3 Sample Features

We argue that our RPR-SGD framework is easy to use, so we do not try very hard to extract advanced features (although they are apparently future work). **Tbl 2** summarizes our sample features extracted by our first user (Pili). He is not an expert in text analysis like topic mining, sentiment analysis, etc. Writing the logic to extract those features adds up to approximately (maybe slightly more than) one day's work.

Most of the features should be self-explanatory. We only elaborate the "topic_X" and "user_X" features a bit. We first discuss how we extract "topic_news" from messages. TF stands for Term Frequency and IDF stands for Inverse Document Frequency. Those two terms are borrowed from Information Retrieval field. We perform word segmentation (a vital step for Chinese messages) on all incoming messages and get the frequency that each term appears in those messages (DF). Similarly, we can first select all messages tagged as "news" by the Pili and count the term frequency (TF). The intuition is that the more frequent a term appears in "news" category, the more representative it is for this category; Also, the more frequent a term appears in all messages, the less representative it is for this category. The simplest way to implement this notion is TF times IDF. We

Table 2: Features

Name	Description
noise	Random variable in [0,1]
echo	{0, 1}: Whether the message is from myself
contain_link	Whether the message contains text link
topic_interesting	TF*IDF for "interesting"
topic_tech	TF*IDF for "mark" "gold" "silver" "bronze"
topic_news	TF*IDF for "news"
topic_nonsense	TF*IDF for "nonsense" "shit"
user_interesting	As above; Treat "user" as "term"
user_tech	As above
user_news	As above
user_nonsense	As above
text_len	Length of all message (original + retweet)
text_len_clean	Length without emoji, "@xxx" and puctuation
text_orig_len	Length of original message

can extract other topics in the same way. With a strong domain knowledge, the users can group several tags and extract topic together. When one new message comes in, we sum up all the TF*IDF values of its terms. Likewise, if we treat the sender of each message as a "term", we can compute a TF*IDF value for users, namely how likely a user will post certain topic.

We remark that the topic mining part is very elementary. One can use more advanced tools like WordNet, Ontology, etc. As an additional note, the word segmentation we used is Maximum Matching Segmentation (MMSeg), and our dictionary is simply a merge of pymmseg-cpp and SogoW (Sogo dict dating back to 2006). Many new terms, especially cyber phrases, can not be correctly segmented.

7.4 Performance and Complexity

We implemented SGD in SNSRouter project using Python straightforwardly. The code has not been optimized. **Tbl 3** shows the performance v.s. complexity. Step size is chosen as $\alpha = 10^{-2}$. When more rounds of SGD is performed, the Kendall's score becomes higher. In this setting, testing K becomes larger than 0.7 after several dozens of thousand iterations. This is substantial improvement compared to unranked version, meaning 85% pairs are in correct order (i.e. $K = (0.85|E| - 0.15|E|)/|E|$). From the data, we can see that SGD scales well. We also implemented GD. Under the same settings, GD costs about 30s in one step and needs about 15 steps to walk a point with $K \approx 0.75$. The author is confident that the SGD algorithm can be an order of magnitude faster by only code level optimization. For the time being, we leave it to the future work.

We also remark that:

- Larger α make SGD go faster but the oscillation around optimal point is larger. Small α is conservative so that it goes slower but is more stable. Users can configure different step size according to his/her demand and ability of device.
- Exactly optimizing the weights is not needed, as we argued in the formulation section. A set of coarsely tuned weights is already good enough to boost the user's efficiency. Users who are conscious of their domain knowledge can set the weights manually (or use it as initial point w^0). The first user (Pili) tried to manually craft a set of weights before any learner algorithm (e.g. SGD) is available. He spent 1 minute and came up with a weighting scheme which reaches a Kendall's score of 0.68. As the data set varies, this score may not be comparative to the above scores. Nevertheless, he can tell that the ranked timeline is already significantly better than raw timeline.

Table 3: Training with SGD

Item	1.	2.	3.
# of rounds of SGD	200,000	400,000	1,000,000
Wall clock time	32.63s	60.81s	159.57s
Kendall's score (training)	0.8178	0.8349	0.8414
Kendall's score (testing)	0.7598	0.7758	0.7865

7.5 Improved User Efficiency

We present the user feedback in **Tbl 4** to show RPR-SGD improves user efficiency significantly. We divide time period by weeks to absorb natural variance such as workday v.s. weekend, day v.s. night, etc. This deployment connects to Renren, SinaWeibo, TencentWeibo, SQLite, Gmail, and several other RSS feeds. "All" is the total number of messages fetched during the experiment time. "Seen" is the total number of messages seen by the user (flagged as "seen" from SRFE). "Value" means useful information to the user. As a rough estimation, we regard the following tags as "Value": "gold", "silver", "bronze", "mark", "news", "interesting" (As is shown in **Fig 5**, they have "$\succeq$" relation to "null"). The last column "V/S" counts the ratio between "Value" and "Seen".

Table 4: User Efficiency Evaluation

No.	Period	All	Seen	Value	V/S
1	Nov 13 - Nov 20	6578	1489	114	7.6561%
2	Nov 20 - Nov 27	9040	1544	138	8.9378%
3	Nov 27 - Dec 04	8472	846	184	21.749%
4	Dec 04 - Dec 11	8243	225	56	24.889%

It is obvious that RPR-SGD improves user efficiency. To help the readers truthfully interpret the presented result, we should note the following facts:

- Preliminary reranking test was brought online at Nov 26. One can see the V/S ratio is already slightly higher in week2.
- In week1, the SRFE is not running continuously day and night, because it is in the intensive debug period. Number of total messages is significantly smaller. This is only the number of messages collected during Pili's working time.
- Compared to week2, week3 and week4 have fewer number of total messages. This is because connection from CUHK campus to TencentWeibo API server failed since about Dec 1. We observed the unstability issue of TecentWeibo API server several times prior to SNSRouter project. This is out of our debugging ability.

- In week3 and week4, the user was not fully exposed to ranked timeline. He went back to ordinary timeline from time to time. This is to serve more unbiased samples for the algorithm. It will lower "V/S" somehow, but bring positive effect in the long run.

- When viewing the ranked timeline, user's criterion will be subconsciously higher. This is natural psychological effect. If the baseline is better, people tend to be more strict. Even with perfect ranking, people are unlikely to tag most of the top entries as valuable.

- We observe that duplication is a major cause to make "V/S" value look poorer than Kendall's score. Many times, the same information was tweeted by different sources using different wording. If it is valuable (according to our trained weighting scheme), all those messages are ranked high. However, the user may only "mark" one of them for future retrieval. We will discuss more on this aspect in the future work section.

In total, we find RPR-SGD can improve user efficiency substantially. It performs good in screening off noisy messages. As to duplicated high quality messages, more work has to be done to improve user efficiency to a higher level.

7.6 Robustness

In this experiment, we test how robust our algorithm is to noise. This is important measurement because we expect user to extract features for themselves. If their feature is not informative under RPR model or is nearly to noise, our weighting should not deviate too much. Being robust is the first step. In the future work, we can add automatic feature selection techniques.

We start with features listed in **Tbl 2** without "noise", and run enough steps of SGD to get a point with $K > 0.8$. The absolute values of trained weights for the rest features are all less than 10. Then we inject 1 noise feature, which is a random variable $\sim U[0, 1]$. We initialize the weight of "noise" by 10. **Tbl 5** shows the training result.

Table 5: Robustness Test

	Init	Round=200K	Round=400K
Kendall	0.0772	0.5435	0.8060
w(noise)	10.0	1.3407	-0.0132

One can see that the Kendall's score is close to 0 initially. This is because we initialized "noise" with very large weight and it becomes the dominating factor. Random ordering of the messages should result in $K \approx 0$. With more steps of SGD, the absolute value of the weight of "noise" becomes smaller and corresponding Kendall's score also improves.

We have the following remarks:

- We can add regularization terms to force non-informative features' weights approach zero. This is left to future work.

- Although the original purpose of "noise" feature is just for experimenting, we later find that "noise" is a very good feature. With small but not fully negligible weight, it introduces reasonable disturbance into the system, thus providing less biased samples. For example, if the user make posting time a feature, the algorithm may conclude that "larger timestamp is preferred". This is purely because messages are presented in reverse chronological order on the original timeline (where the users mark for training samples). When two messages are similar, only the first one "seen" by the user are marked. Such self-validation effect can be alleviated by a noise feature.

Figure 6: Ranked Timeline with Echo

7.7 Case Study – Echo Cancellation

In this section, we provide abundant screenshots with detailed steps to show how to add a new feature. We also present animation of training process to give the reader an impression of how RPR-SGD is adapting to new changes.

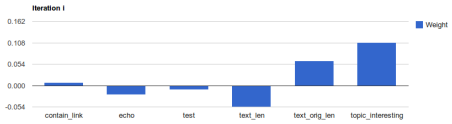
We choose Echo Cancellation as an example. SNS users should have noticed that they see their own messages on the timeline among others' posts. This is true for most platforms. We call those messages "echo". **Fig 6** shows the ranked timeline before echo cancellation. Echo will be more significant when the timeline is ranked: 1) A rational user is most likely to forward valuable (at least to him/her) messages; 2) The ranking algorithm learns the user's preference and rank those messages higher; 3) When forwarding the message, the user adds some comments, making it more valuable (from the algorithms point of view); 4) After fetching new messages, echos are more likely to rank higher.

We have many ways to cancel echos. For example, a programmer will find it very easy to use several lines of "IF" to achieve this goal. This way fits original notion of SNSAPI. We just enable the flexibility and let App developer to craft their own logic. In SNSRouter project, we are more serious about ranking. If one only needs to cancel echos, the straight coding is a good solution. However, we can imagine that users have many similar demand, e.g. "rank messages from some people higher", "rank messages containing music links higher", etc. When the "IF" tree grows large, it will be hard to manage it efficiently. RPR provides the framework so that users only need to tell the machine what factor may be important. The machine will automatically adjust between different factors.

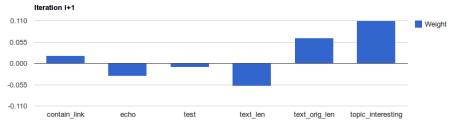
Here is the way to to **softly** cancel echos:

1. Add tag "echo" on the config panel of SRFE, as is shown in **Fig 7(a)**.

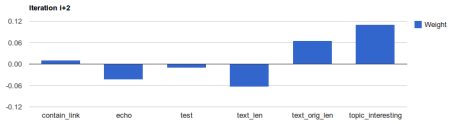
2. Users go to SRFE and tag some messages as "echo". The user can do this gradually for some time. Or, the user can go to our SQL frontend on SRFE to inten-



(a) Iteration i



(b) Iteration i+1



(c) Iteration i+2

Figure 8: Training Echo Feature

tionally select out some messages. In the latter way, users are leveraging their own domain knowledge.

3. Specify preference through json config file, as is shown in **Fig 7(b)**. Here we only add one preference relation: "null" $\succeq$ "echo".

4. Add the feature extraction logic as is shown in **Fig 7(c)**. The user simply derives a feature extractor from base class "FeatureBase". Except for structural codes, this example feature extractor only costs about 10 lines of codes. Many of the 15 feature extractors used in previous evaluation are also only several dozens of lines.

5. After the above preparation, the weights can be trained automatically. **Fig 8** gives an animation of how weights are adapted to the new scenario. To make it less cluttered, we selectively depicted some features for comparison. One can consult the notes in SNSRouter repository for detailed data. The figures show weight of "echo" (2nd column) goes negatively with more iterations and the other features remain roughly the same.

We remark that the weights adaption shown in **Fig 8** is highly interpretable for the user. The "echo" feature is either 1 or 0 and it recognizes echos by comparing sender name with channel configurations. Since its weight is negative, messages with "echo=1" will be ranked lower. This agrees with the "IF" logic in our first thought. The nice property for RPR-SGD is that this weight is trained automatically.

8. CONCLUSION

We observe that message forwarding is a major operation on SNS. Cross-platform forwarding behaviour is seldom discussed in both industry and academia. One main reason is that there is no full solution stack to make it automatic or semi-automatic, in order for researcher to collect enough data. There are also some business considerations discussed in Appendix A. In this paper, we take an initial step to tackle with the problem by leveraging system, algorithm and human in the following steps:

- Develop a middleware for heterogeneous SNS's.
- Develop a portable web frontend which supports easy cross-platform message forwarding and personal tagging.
- Collect real data in more than one month time span.
- Propose a flexible algorithm framework (RPR-SGD) to personalize incoming message ordering, which boosts the user efficiency.
- Develop some sample feature extraction modules.

Note that the usage of SNSRouter is not restricted to forwarding assistant. It can be used as a personal information center (with more forthcoming SNSAPI plugins). For example, monitor your course webpage and send emails to your friends when new slides are posted. For another example, follow celebrities on SinaWeibo so that their messages (including later deleted ones) are all automatically archived. This provides a distributed witness force, which help to improve their awareness of social responsibility! [1]

In total, SNSAPI and SNSRouter are open by design. Your imagination combined with our system infrastructure can result in many novel applications.

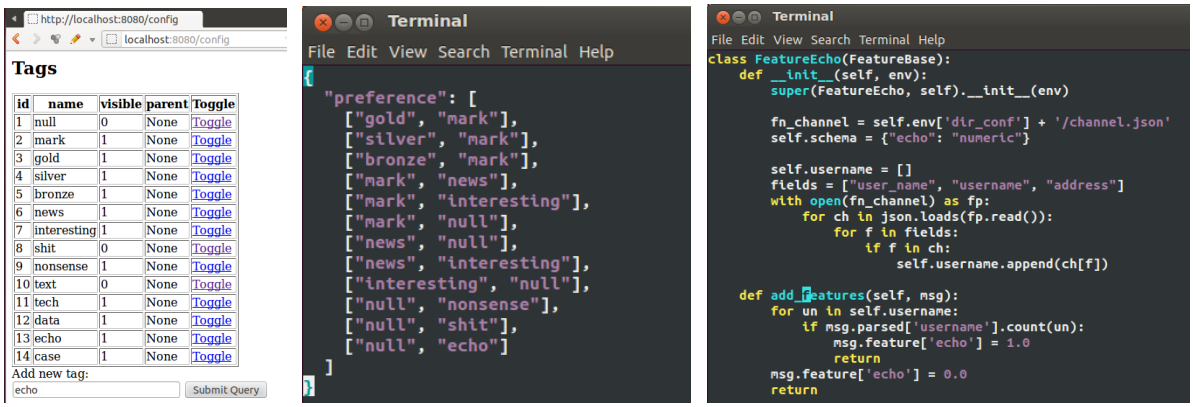
9. FUTURE WORK

In this section, we discuss promising future works. Some are directly followed from the discussions in previous sections. Some are our visions. We discuss from three aspects: system, algorithm and evaluation.

9.1 System

- Develop RESTful interface for all components of SNSRouter. Then one can outsource computationally intensive training to other servers.
- Add a new platform – SNSRouter to SNSAPI. One can then fetch SNSRouter timeline in the same way as others. In this way, users can use SNSRouter to aggregate multiple channels, which is useful to deal with large amount of RSS feeds.
- Build user community. We can provide online "FeatureStore" and "LearnerStore". In this way, users do not have to be constrained to our sample features or try to extract anything themselves. They can upload their extractors and download others' extractors. Those components are pluggable by configuration so that better personalization level is reached.

[1]Many celebrities on SinaWeibo act like media. They pursue the speed of dissemination information and the exposure rate. Many of them do not spend a minute to validate the messages they forward. After people find out it is rumor, they just delete the original message quietly without one word of apology. Late comers will only see a sentence in original position: "This message is deleted, please consult the customer service for more information" (in Chinese). However, the influence is already far spread on the network. I believe, the OSN's will be more reliable if many people are running SNSRouter. Under this scenario, those celebrities will think twice before they act. When mistakes are made, they will prefer to fix them rather than hide them.



(a) Add New Tag (b) Specify Preference for Echo (c) Add Feature Extraction Module

Figure 7: Three Steps to Add New Features

9.2 Algorithm

- Add regularization terms. First is to alleviate overfitting and standard L2 norm regularization suffices. Second, we can perform feature selection using L1 regularization first and only train selected features.
- Build advanced feature extractors. In the topic mining part, we used TF*IDF. There is large room to improve. For example, at the startup time, the system do not collect enough terms to do TF*IDF. If we plug a WordNet into SNSRouter, we can enlarge the term set by diffusion techniques. This also provides a way to smooth term importance and mine hidden (semantical not literal) topic.
- Code level optimization of SGD. As is mentioned in evaluation section, code level optimization for SGD can be done. We can change to "numpy" objects to avoid time consuming operations like list traversal. We can also implement a dedicated SGD library in C with Python module encapsulation. To make the code easy to migrate, the current stupid implementation will be kept as default portability consideration. After the users verify everything is going well, they can configure to optimized modules.
- SGD can do online training. e.g. one sample in, derive some pairs, do SGD on those pairs. In this way, it provides natural time sliding. Current SGD implementation in SNSRouter is batch training, which can be triggered by the user. One recent target is to make it online. The difficulty is not at algorithm side. Instead, sliding memory caching is needed to make it possible. Original plain SQLite interface will cause very large overhead.
- Deduplication. As we discussed in evaluation section, deduplication is a major cause to make overall efficiency gain for the users less significant than we expect from Kendall's score. Algorithmic wise, this is a traditional problem in IR. As to our application scenario, we also want to improve the frontend at the same time, so that users can process duplicated messages quickly. For example, we can present similar messages in tree structure so that users can"mark as seen" in a batch.

9.3 Evaluation

More subjective tests are to be done:

- We argue the algorithm framework is easy to personalize. In the future work, we will invite more developers who are not machine learning experts to test our system. In this way, we can see how efficient people are after they use our system. More sample extractors can be output in this process.

Acknowledgements

SNSAPI is the base of SNSRouter project. We would like to thank Junbo Li from BUPT, who is the co-founder of SNSAPI. We thank other contributors from Github: Chunliang Lu. We also highly appreciate people who answered our technical questions on StackOverflow. We thank Xiaoying Tang for her review of this report.

10. REFERENCES

[1] P. Hu. Tutorial collection. GitHub, https://github.com/hupili/tutorial, 3 2012. HU, Pili's tutorial collection.
[2] Wikipedia, Floyd-Warshall Algorithm, http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
[3] Wikipedia, Kendall's tau Rank Correlation Coefficient, http://en.wikipedia.org/wiki/Kendall_tau_rank_correlation_coefficient
[4] Pili Hu, Junbo Li, SNSAPI, https://snsapi.ie.cuhk.edu.hk/
[5] Pili Hu, SNSRouter, https://github.com/hupili/sns-router
[6] Pili Hu, Junbo Li, SNSAPI Github Repository, https://github.com/hupili/snsapi
[7] IFTTT, https://ifttt.com/
[8] Yahoo Pipies, http://pipes.yahoo.com/pipes/
[9] Wikipedia, Pareto Principple, http://en.wikipedia.org/wiki/Pareto_principle
[10] Wikipedia, Keep It Simple and Stupid, http://en.wikipedia.org/wiki/KISS_principle
[11] Weka, http://www.cs.waikato.ac.nz/ml/weka/

[12] Wikipedia, Stochastic Gradient Descent, http://en.wikipedia.org/wiki/Stochastic_gradient_descent

APPENDIX

A. BUSINESS

We tried to upgrade one sample app key on SinaWeibo but failed. [2] The reason is that they forbid cross-platform forwarding applications. From a few survey, we summarize some of their business considerations:

- Competition. They don't want users just to go to other OSN but possess the ability to synchronize content back to SinaWeibo. They want to lock users in because advertisements are presented on user timeline if they access it from web. This is very similar to the old days when you can not transfer cellphone number from one service provider to another. Considering the links established within one platform, users find it difficult to migrate to another platform.
- Censorship. SinaWeibo monitors user content and delete / block / hide messages if they are not appropriate. If multi-platform synchronization is allowed, those messages forwarded to another platform are out of control. In this case, they forbid any form of synchronization (even with application and audition).

The first one is out of our consideration. The second one is just placing a thin film to slow down the information dissemination process. You can not stop users' copy and paste anyway.

The way they stop such applications is by killing their app keys. This is effective if the App developer resort to a closed system and hide all details from the user. On another hand, we agree that this lowers the barrier for ordinary users to use the App.

Given those observations, we position SNSAPI and SNSRouter as developer applications. If a user want to use SNSRouter, he/she may have to spend several minutes to apply his/her own app key. Then the use of SNSRouter from service provider's point of view is just someone testing a new App. This is the freedom of developers and is in general hard to block.

B. CLASSIFICATION

As we mentioned in the formulation section, classification is one candidate formulation initially. We dumped data to arff format and ran some preliminary experiments.

B.1 Logit Classifier

The accuracy that Logit Classifier can reach is 78.5526%. Comparing with the data we presented in evaluation section, this performance is not competitive. A 0.8 Kendall's score maps back to 90% correctly ranked pairs. Incorrectly classified instances will amplify to larger number of incorrect pairs. The confusion matrix is presented in **Fig 9**. We can see that the most confused part is "news". From the user's point, this is natural. When I tagged messages as news, I already realized that there were not enough features for "news". First, they come from a wide spectrum of topics: politics, military, science, etc. It is hard to use topic to judge the "news"-ness, letting alone judge it with our elementary topic miner.

[2]See the issue tracker for more details, https://github.com/hupili/snsapi/issues/11

B.2 J48 Decision Tree

J48 decision tree can reach an overall accuracy of 87.3684% (testing). This value is very good. The confusion matrix is presented in **Fig 10**. One can see that it performs equally well for all classes. This is because decision tree can generate arbitrary (axis perpendicular) decision boundary. We also remark that the training and testing speed for J48 very high. All the process is completed in less than 5 seconds.

One drawback is that overfitting is obvious in J48. We present a sample branch for "mark" as follows:

```
   topic_news <= 0.00603
&& topic_tech <= 0.041455
&& topic_interesting <= 0.042225
&& topic_nonsense <= 0.010593
&& text_len > 0.12
&& id <= 30634
&& user_tech <= 0.010894
&& text_len_clean <= 0.0575
&& user_tech > 0.001621
==> mark (3.0/1.0)
```

The "3.0/1.0" in bracket means that 3 instances are correctly classified along this branch, and 1 instance is incorrectly classified. This small number is one sign of overfitting. Another sign is the "id" branch. It is just the rowid in SQLite DB and should not play any role in classification. Proper tuning may make it work better but the process will be less natural for anyone who is not machine learning researcher or practitioner.

C. EXECUTIVE SUMMARY OF SNSROUTER USAGE

This section summarizes what is done with SNSRouter. It is not a file level or instruction level guide. It is a higher level view so that users are clear about what to do and what not to do.

- Download SNSRouter from project repository.
- Install dependencies according to our wiki.
- Launch initial test with our sample configuration files, where several RSS channels are configured by default.
- Go to Web UI and verify everything is functioning.
- Configure more RSS channels and SQLite channels and verify they work. RSS and SQLite do not need authorization or authentication, so that are easier.
- If the user is not clear about channel configuration, please go to SNSAPI project and use SNSCLI, in which we embedded an online tutorial.
- Apply app keys from OSN's and configure them accordingly. As for app key application, we have dedicated wiki page for troubleshooting. As for channel configuration, one can also turn to SNSCLI.
- Go to Web UI and verify the rich configuration also works (without ranking).
- Add tags from config panel, and tag incoming messages accordingly. Remember to "mark as seen" so that one message will not show up again (by default). We do not impose any constraints on tags. However, users should

```
a   b   c   d   e   f   g   h   i   j   k   l   m   n   o   <-- classified as
5   0   0   0   0   0   0   0   0   0   0   2   0   0   0 |   a = echo
0   2   0   0   0   0   0   0   0   0   0   0   0   0   0 |   b = case
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   c = gold
0   2   0  16   0   0   0   0   0   0   0   2   0   0   0 |   d = interesting
0   0   0   2  71   0   0   0   7   0   0  15   0   0   0 |   e = nonsense
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   f = __fake__
0   0   0   0   0   0  90   0   0   0   1  22   0   0   0 |   g = mark
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   h = tech
0   0   0   3   1   0   0   0  10   0   0   0   0   0   0 |   i = shit
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   j = text
0   0   0   0   0   0   1   0   0   0  40  76   0   0   0 |   k = news
2   1   0   4   4   0   4   0   0   0  14 363   0   0   0 |   l = null
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   m = data
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   n = silver
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   o = bronze
```

Figure 9: Confusion Matrix, Logit Classifier

```
a   b   c   d   e   f   g   h   i   j   k   l   m   n   o   <-- classified as
6   0   0   0   0   0   1   0   0   0   0   0   0   0   0 |   a = echo
0   2   0   0   0   0   0   0   0   0   0   0   0   0   0 |   b = case
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   c = gold
0   0   0  15   0   0   0   0   0   0   0   5   0   0   0 |   d = interesting
0   0   0   0  70   0   2   0   6   0   1  16   0   0   0 |   e = nonsense
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   f = __fake__
0   0   0   0   0   0  94   0   0   0   3  16   0   0   0 |   g = mark
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   h = tech
0   0   0   0   1   0   0   0  13   0   0   0   0   0   0 |   i = shit
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   j = text
0   0   0   0   0   0   1   0   0   0 104  12   0   0   0 |   k = news
2   0   0   0  10   0  14   0   0   0   6 360   0   0   0 |   l = null
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   m = data
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   n = silver
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |   o = bronze
```

Figure 10: Confusion Matrix, J48

try their best to keep consistency and define them in mind clearly.

- Collect enough samples (several days; thousands of "seen" messages; hundreds of tagged messages).
- Install dependencies of ranking module. Enable ranking with default configuration for testing.
- Enable more feature extractors based on the users' observation. Different extractors has different dependencies. Users only need to worry about them when one extractor is to be activated.
- Write personalized extractor to meet a specific demand.
- Share feature extractors with others.