Photo credit: Dave DiCello

17-803 Empirical Methods

Bogdan Vasilescu, Institute for Software Research

Literature Reviews

Thursday, February 11, 2021

Outline for Today

- The role of theory – leftovers from Tuesday
- Literature reviews

I. The Role of Theory

(Leftovers)

- Topic: using AI to generate programming source code from natural language
- 9 months into his PhD
- Has built a tool
- Needs an evaluation plan

Meet Stu Dent

Stu Dent idea by Steve Easterbrook

Stu's Evaluation Plan

- Controlled experiment using an IDE plugin
  - Independent variable: Stu's "NL2Code" vs. writing code "from scratch"
  - Dependent variables: correctness, speed, subjective assessment
  - Tasks: various Python
  - Subjects: CS grad students
  - Hypotheses:
    - H1: "Code written using NL2Code is more often correct than code written from scratch."
    - H2: "Subjects complete tasks faster when using NL2Code than when writing code from scratch."
    - H3: "Subjects prefer using the snippets from NL2Code over writing code from scratch."
- Results:
  - H1 & H2 & H3 rejected*
  - Subjects found NL2Code unintuitive

* True story: https://arxiv.org/abs/2101.11149

Why Build a Tool?

- Build a Tool to Test a Theory
  - Tool is part of the experimental materials needed to conduct your study
- Build a Tool to Develop a Theory
  - Theory emerges as you explore the tool
- Build a Tool to Explain your Theory
  - Theory as a concrete instantiation of (some aspect of) the theory

Slide by Steve Easterbrook

Stu's Theory

- Background assumptions
  - Tasks can be completed by piecing together code snippets involving popular libraries / APIs
  - Many such example code snippets are available in NL2Code's trained data
  - …

Stu's Theory

Basic theory (brief summary)

- Programmers decompose tasks into a sequence of (small) steps.
- At every step, they know conceptually what must be done next, but
  - (a) do not know how to create a concrete implementation of their idea, or
  - (b) would rather not have to type it in.
- NL2Code could help speed up task completion especially in the (b) scenario;
  - otherwise, with (a) users might not recognize which NL2Code search result to use, if multiple, or know how to integrate that snippet into their program.
- Possible speedups would occur primarily because users risk getting distracted when they switch context going outside of their IDEs;
  - not because of the time it would take to write down source code (because programmers mostly copy paste code from Stack Overflow anyway; they rarely write code from scratch).
- …

Stu's Theory

- Some possible derived hypotheses:
  - For tasks where programmers have extensive prior experience (i.e., they could have written solutions from scratch), using NL2Code should reduce task completion times.
  - The more steps (e.g., API calls) are involved in implementing a solution to a task, the more NL2Code should speed up task completion times.
  - …

Take-Home Messages

- Articulate the theory(s) underlying your work
- Be precise about your research questions
- Be deliberate (and ideally explicit) about your philosophical stance
- Use the theory to guide the study design

Test the theory, not the tool!

Slide by Steve Easterbrook

Summary

- In any empirical study, theories become a "lens" through which the world is observed and interpreted, whether or not they are explicitly acknowledged.
  - Real-world phenomena too rich / complex to study without that much filtering.

- Quantitative methods:
  - Theory to decide which variables to isolate and measure, and which to ignore or exclude.
- Qualitative methods:
  - Theory to focus data analysis / interpretation.

Summary

- Without the theory, we have no way of making sense of the accumulation of empirical results.
  - An individual study can never offer conclusive results.

- Theories support analytical generalization
  - Provide a deeper understanding of our empirical results
  - ...and hence how they apply more generally
  - Much more powerful than statistical generalization

All Methods Are Flawed

- E.g. Laboratory Experiments
  - Cannot study large scale software development in the lab!
  - Too many variables to control them all!
- E.g. Case Studies
  - How do we know what's true in one project generalizes to others?
  - Researcher chose what questions to ask, hence biased the study
- E.g. Surveys
  - Self-selection of respondents biases the study
  - Respondents tell you what they think they ought to do, not what they actually do
- ...etc...

Slide by Steve Easterbrook

Strategies To Overcome Weaknesses

- Theory-building
  - Testing a hypothesis is pointless (single flawed study!)…
  - …unless it builds evidence for a clearly stated theory
- Empirical induction
  - Series of studies over time…
  - Each designed to probe more aspects of the theory
  - …together build evidence for a clearly stated theory
- Mixed-methods research
  - Use multiple methods to investigate the same research question
  - Each method compensates for the flaws of the others
  - …together build evidence for a clearly stated theory

Slide by Steve Easterbrook

II. Literature Review

- Lingard, L. (2015). Joining a conversation: the problem/gap/hook heuristic. Perspectives on Medical Education, 4(5), 252-253.
- Lingard, L. (2018). Writing an effective literature review. Perspectives on Medical Education, 7(2), 133-135.
- Justin Zobel, Writing for Computer Science (3rd Edition). Springer, 2015

A Literature Review Helps You Choose a Research Topic

Can it be studied? vs *Should* it be studied?

- *Does it add to the body of knowledge?*
- *Who else besides you would care about results?*

A Literature Review Serves Multiple Purposes

- Report what is known about your topic.
  - Share with reader results from related studies
  - Benchmark for comparing results

- Main purpose: identify what remains *unknown*.
  - Relate your study to literature, filling in gaps
  - Direction for your research questions and hypotheses
  - Framework for establishing the importance of your study

Several Forms of Literature Review Are Possible

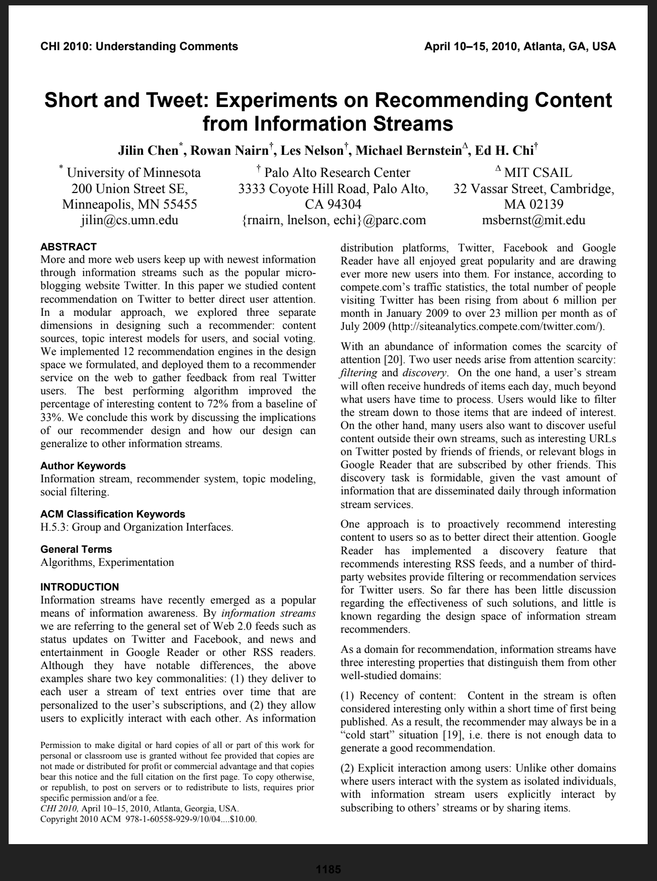
- Integrate what others have done and said
- Criticize prior work
- Build bridges between related topics
- Identify the central issues in a field

Side Effect: Lit Review Pushes You To Articulate Your Contributions

- Does your project:
  - Address a new topic?
  - Use a new data collection method?
  - Extend the discussion?
  - Refine / extend a theory?
  - Replicate a study in a new situation?
  - …

Activity: Read and Discuss the Lit Reviews in One of the Following Papers

Read up to “DESIGNING RECOMMENDERS FOR TWITTER” on p. 1187



Short and Tweet: Experiments on Recommending Content from Information Streams

Jilin Chen[*], Rowan Nairn[†], Les Nelson[†], Michael Bernstein[Δ], Ed H. Chi[†]

[*] University of Minnesota
200 Union Street SE,
Minneapolis, MN 55455
jilin@cs.umn.edu

[†] Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304
{rnairn, lnelson, echi}@parc.com

[Δ] MIT CSAIL
32 Vassar Street, Cambridge, MA 02139
msbernst@mit.edu

ABSTRACT
More and more web users keep up with newest information through information streams such as the popular micro-blogging website Twitter. In this paper we studied content recommendation on Twitter to better direct user attention. In a modular approach, we explored three separate dimensions in designing such a recommender: content sources, topic interest models for users, and social voting. We implemented 12 recommendation engines in the design space we formulated, and deployed them to a recommender service on the web to gather feedback from real Twitter users. The best performing algorithm improved the percentage of interesting content to 72% from a baseline of 33%. We conclude this work by discussing the implications of our recommender design and how our design can generalize to other information streams.

Author Keywords
Information stream, recommender system, topic modeling, social filtering.

ACM Classification Keywords
H.5.3: Group and Organization Interfaces.

General Terms
Algorithms, Experimentation

INTRODUCTION
Information streams have recently emerged as a popular means of information awareness. By *information streams* we are referring to the general set of Web 2.0 feeds such as status updates on Twitter and Facebook, and news and entertainment in Google Reader or other RSS readers. Although they have notable differences, the above examples share two key commonalities: (1) they deliver to each user a stream of text entries over time that are personalized to the user’s subscriptions, and (2) they allow users to explicitly interact with each other. As information distribution platforms, Twitter, Facebook and Google Reader have all enjoyed great popularity and are drawing ever more new users into them. For instance, according to compete.com’s traffic statistics, the total number of people visiting Twitter has been rising from about 6 million per month in January 2009 to over 23 million per month as of July 2009 (http://siteanalytics.compete.com/twitter.com/).

With an abundance of information comes the scarcity of attention [20]. Two user needs arise from attention scarcity: *filtering* and *discovery*. On the one hand, a user’s stream will often receive hundreds of items each day, much beyond what users have time to process. Users would like to filter the stream down to those items that are indeed of interest. On the other hand, many users also want to discover useful content outside their own streams, such as interesting URLs on Twitter posted by friends of friends, or relevant blogs in Google Reader that are subscribed by other friends. This discovery task is formidable, given the vast amount of information that are disseminated daily through information stream services.

One approach is to proactively recommend interesting content to users so as to better direct their attention. Google Reader has implemented a discovery feature that recommends interesting RSS feeds, and a number of third-party websites provide filtering or recommendation services for Twitter users. So far there has been little discussion regarding the effectiveness of such solutions, and little is known regarding the design space of information stream recommenders.

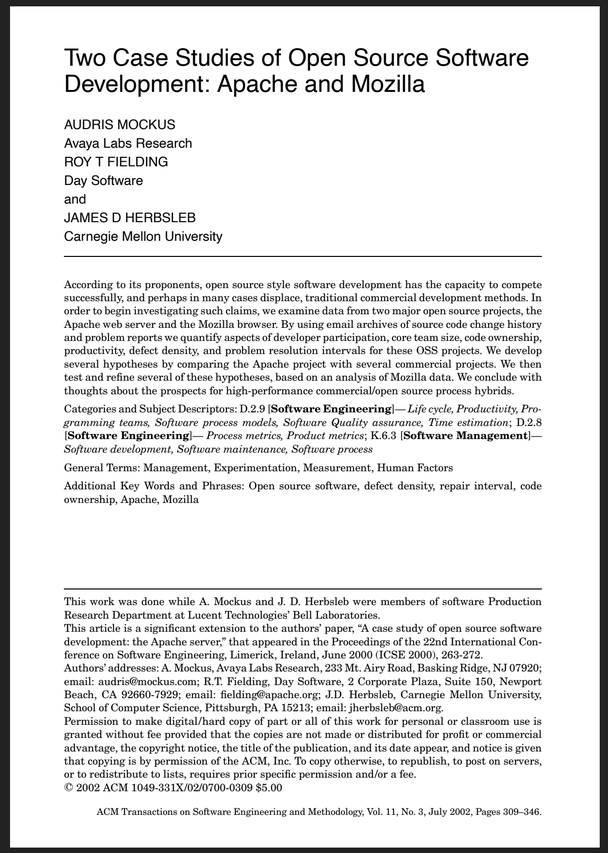
As a domain for recommendation, information streams have three interesting properties that distinguish them from other well-studied domains:

(1) Recency of content: Content in the stream is often considered interesting only within a short time of first being published. As a result, the recommender may always be in a “cold start” situation [19], i.e. there is not enough data to generate a good recommendation.

(2) Explicit interaction among users: Unlike other domains where users interact with the system as isolated individuals, with information stream users explicitly interact by subscribing to others’ streams or by sharing items.

CHI 2010, April 10–15, 2010, Atlanta, Georgia, USA.

Read up to “METHODOLOGY AND DATA SOURCES” on page 302



Two Case Studies of Open Source Software Development: Apache and Mozilla

AUDRIS MOCKUS
Avaya Labs Research
ROY T FIELDING
Day Software
and
JAMES D HERBSLEB
Carnegie Mellon University

According to its proponents, open source style software development has the capacity to compete successfully, and perhaps in many cases displace, traditional commercial development methods. In order to begin investigating such claims, we examine data from two major open source projects, the Apache web server and the Mozilla browser. By using email archives of source code change history and problem reports we quantify aspects of developer participation, core team size, code ownership, productivity, defect density, and problem resolution intervals for these OSS projects. We develop several hypotheses by comparing the Apache project with several commercial projects. We then test and refine several of these hypotheses, based on an analysis of Mozilla data. We conclude with thoughts about the prospects for high-performance commercial/open source process hybrids.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]— *Life cycle, Productivity, Programming teams, Software process models, Software Quality assurance, Time estimation*; D.2.8 [**Software Engineering**]— *Process metrics, Product metrics*; K.6.3 [**Software Management**]— *Software development, Software maintenance, Software process*

General Terms: Management, Experimentation, Measurement, Human Factors

Additional Key Words and Phrases: Open source software, defect density, repair interval, code ownership, Apache, Mozilla

This work was done while A. Mockus and J. D. Herbsleb were members of software Production Research Department at Lucent Technologies’ Bell Laboratories.
This article is a significant extension to the authors’ paper, “A case study of open source software development: the Apache server,” that appeared in the Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June 2000 (ICSE 2000), 263-272.
Authors’ addresses: A. Mockus, Avaya Labs Research, 233 Mt. Airy Road, Basking Ridge, NJ 07920; email: audris@mockus.com; R.T. Fielding, Day Software, 2 Corporate Plaza, Suite 150, Newport Beach, CA 92660-7929; email: fielding@apache.org; J.D. Herbsleb, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213; email: jherbsleb@acm.org.

ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 3, July 2002, Pages 309–346.

Discussion points:

- How much prior work was there?
- How is the literature organized?
- What kind of questions is the paper addressing?
- What is the knowledge gap being addressed?

Activity: Read Beginning of "Two Case Studies.." By Mockus Et Al

Two Case Studies of Open Source Software Development: Apache and Mozilla

AUDRIS MOCKUS
Avaya Labs Research
ROY T FIELDING
Day Software
and
JAMES D HERBSLEB
Carnegie Mellon University

According to its proponents, open source style software development has the capacity to compete successfully, and perhaps in many cases displace, traditional commercial development methods. In order to begin investigating such claims, we examine data from two major open source projects, the Apache web server and the Mozilla browser. By using email archives of source code change history and problem reports we quantify aspects of developer participation, core team size, code ownership, productivity, defect density, and problem resolution intervals for these OSS projects. We develop several hypotheses by comparing the Apache project with several commercial projects. We then test and refine several of these hypotheses, based on an analysis of Mozilla data. We conclude with thoughts about the prospects for high-performance commercial/open source process hybrids.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]— *Life cycle, Productivity, Programming teams, Software process models, Software Quality assurance, Time estimation*; D.2.8 [**Software Engineering**]— *Process metrics, Product metrics*; K.6.3 [**Software Management**]— *Software development, Software maintenance, Software process*

General Terms: Management, Experimentation, Measurement, Human Factors

Additional Key Words and Phrases: Open source software, defect density, repair interval, code ownership, Apache, Mozilla

This work was done while A. Mockus and J. D. Herbsleb were members of software Production Research Department at Lucent Technologies' Bell Laboratories.
This article is a significant extension to the authors' paper, "A case study of open source software development: the Apache server," that appeared in the Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June 2000 (ICSE 2000), 263-272.
Authors' addresses: A. Mockus, Avaya Labs Research, 233 Mt. Airy Road, Basking Ridge, NJ 07920; email: audris@mockus.com; R.T. Fielding, Day Software, 2 Corporate Plaza, Suite 150, Newport Beach, CA 92660-7929; email: fielding@apache.org; J.D. Herbsleb, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213; email: jherbsleb@acm.org.

ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 3, July 2002, Pages 309–346.

1. INTRODUCTION

The open source software "movement" has received enormous attention in the last several years. It is often characterized as a fundamentally new way to develop software [Dibona et al. 1999; Raymond 1999] that poses a serious challenge [Vixie 1999] to the commercial software businesses that dominate most software markets today. The challenge is not the sort posed by a new competitor that operates according to the same rules but threatens to do it faster, better, cheaper. The OSS challenge is often described as much more fundamental, and goes to the basic motivations, economics, market structure, and philosophy of the institutions that develop, market, and use software.

The basic tenets of OSS development are clear enough, although the details can certainly be difficult to pin down precisely (see Perens [1999]). OSS, most people would agree, has as its underpinning certain legal and pragmatic arrangements that ensure that the source code for an OSS development will be generally available. Open source developments typically have a central person or body that selects some subset of the developed code for the "official" releases and makes it widely available for distribution.

These basic arrangements to ensure freely available source code have led to a development process that is radically different, according to OSS proponents, from the usual industrial style of development. The main differences most often mentioned are the following.

- OSS systems are built by potentially large numbers (i.e., hundreds or even thousands) of volunteers. It is worth noting, however, that currently a number of OSS projects are supported by companies and some participants are not volunteers.
- Work is not assigned; people undertake the work they choose to undertake.
- There is no explicit system-level design, or even detailed design [Vixie 1999].
- There is no project plan, schedule, or list of deliverables.

Taken together, these differences suggest an extreme case of geographically distributed development, where developers work in arbitrary locations, rarely or never meet face to face, and coordinate their activity almost exclusively by means of email and bulletin boards. What is perhaps most surprising about the process is that it lacks many of the traditional mechanisms used to coordinate software development, such as plans, system-level design, schedules, and defined processes. These "coordination mechanisms" are generally considered to be even more important for geographically distributed development than for colocated development [Herbsleb and Grinter 1999], yet here is an extreme case of distributed development that appears to eschew them all.

Despite the very substantial weakening of traditional ways of coordinating work, the results from OSS development are often claimed to be equivalent, or even superior to software developed more traditionally. It is claimed, for example, that defects are found and fixed very quickly because there are "many eyeballs" looking for the problems (Eric Raymond [1999] calls this "Linus's Law"). Code is written with more care and creativity, because developers are working only on things for which they have a real passion [Raymond 1999].

It can no longer be doubted that OSS development has produced software of high quality and functionality. The Linux operating system has recently enjoyed major commercial success, and is regarded by many as a serious competitor to commercial operating systems such as Windows [Krochmal 1999]. Much of the software for the infrastructure of the Internet, including the well-known bind, Apache, and sendmail programs, were also developed in this fashion.

The Apache server (one of the OSS software projects under consideration in this case study) is, according to the Netcraft survey, the most widely deployed Web server at the time of this writing. It accounts for over half of the 7 million or so Web sites queried in the Netcraft data collection. In fact, the Apache server has grown in "market share" each year since it first appeared in the survey in 1996. By any standard, Apache is very successful.

Although this existence proof means that OSS processes can, beyond a doubt, produce high-quality and widely deployed software, the exact means by which this has happened, and the prospects for repeating OSS successes, are frequently debated (see, e.g., Bollinger et al. [1999] and McConnell [1999]). Proponents claim that OSS software stacks up well against commercially developed software both in quality and in the level of support that users receive, although we are not aware of any convincing empirical studies that bear on such claims. If OSS really does pose a major challenge to the economics and the methods of commercial development, it is vital to understand it and to evaluate it.

This article presents two case studies of the development and maintenance of major OSS projects: the Apache server and Mozilla. We address key questions about their development processes, and about the software that is the result of those processes. We first studied the Apache project, and based on our results, framed a number of hypotheses that we conjectured would be true generally of open source developments. In our second study, which we began after the analyses and hypothesis formation were completed, we examined comparable data from the Mozilla project. The data provide support for several of our original hypotheses.

In the remainder of this section, we present our specific research questions. In Section 2, we describe our research methodology for both the Apache and Mozilla projects. This is followed in Section 3 by the results of Study 1, the Apache project, and hypotheses derived from those results. Section 4 presents our results from Study 2, the Mozilla project, and a discussion of those results in light of our previous hypotheses. We conclude the article in Section 5.

1.1 Research Questions

Our questions focus on two key sets of properties of OSS development. It is remarkable that large numbers of people manage to work together successfully to create high-quality, widely used products. Our first set of questions (Q1 to Q4) is aimed at understanding basic parameters of the process by which Apache and Mozilla came to exist.

Q1: *What were the processes used to develop Apache and Mozilla?*

In answer to this question, we construct brief qualitative descriptions of Apache and Mozilla development processes.

Q2: *How many people wrote code for new functionality? How many people reported problems? How many people repaired defects?*

We want to see how large the development communities were, and identify how many people actually occupied each of these traditional development and support roles.

Q3: *Were these functions carried out by distinct groups of people, that is, did people primarily assume a single role? Did large numbers of people participate somewhat equally in these activities, or did a small number of people do most of the work?*

Within each development community, what division of labor resulted from the OSS "people choose the work they do" policy? We want to construct a profile of participation in the ongoing work.

Q4: *Where did the code contributors work in the code? Was strict code ownership enforced on a file or module level?*

One worry of the "chaotic" OSS style of development is that people will make uncoordinated changes, particularly to the same file or module, that interfere with one another. How does the development community avoid this?

Our second set of questions (Q5 to Q6) concerns the outcomes of these processes. We examine the software from a customer's point of view, with respect to the defect density of the released code, and the time to repair defects, especially those likely to significantly affect many customers.

Q5: *What is the defect density of Apache and Mozilla code?*

We compute defects per thousand lines of code, and defects per delta in order to compare different operationalizations of the defect density measure.

Q6: *How long did it take to resolve problems? Were high priority problems resolved faster than low priority problems? Has resolution interval decreased over time?*

We measured this interval because it is very important from a customer perspective to have problems resolved quickly.

2. METHODOLOGY AND DATA SOURCES

In order to produce an accurate description of the open source development processes, we wrote a draft of description of each process, then had it reviewed by members of the core OSS development teams. For the Apache project, one of the authors (RTF), who has been a member of the core development team from the beginning of the Apache project, wrote the draft description. We then circulated it among all other core members and incorporated the comments of one member who provided feedback. For Mozilla, we wrote a draft based on many published accounts of the Mozilla process.[1] We sent this draft to the Chief Lizard Wrangler who checked the draft for accuracy and provided comments. The descriptions in the next section are the final product of this process. The commercial development process is well known to two of the authors (AM, JDH) from years of experience in the organization, in addition to scores of interviews

[1]Please see Ang and Eich [2000], Baker [2000], Eich [2001], Hecker [1999], Howard [2000], Mozilla Project, Oeschger and Boswell [2000], Paquin and Tabb [1998], Yeh [1999], Williams [2000], and Zawinski [1999].

Mockus Et Al, "Two Case Studies"

- Open source is often characterized as a fundamentally new way to develop software
- The open source development process is radically different from the usual industrial style of development:
  - extreme case of geographically distributed development, where developers work in arbitrary locations, rarely or never meet face to face, and coordinate their activity almost exclusively by means of email and bulletin boards

The Gap + Hook

- What is perhaps most surprising about the process is that it lacks many of the traditional mechanisms used to coordinate software development, such as plans, system-level design, schedules, and defined processes.
- These “coordination mechanisms” are generally considered to be even more important for geographically distributed development than for colocated development [Herbsleb and Grinter 1999], yet here is an extreme case of distributed development that appears to eschew them all.
- Despite the very substantial weakening of traditional ways of coordinating work, the results from OSS development are often claimed to be equivalent, or even superior to software developed more traditionally.

The Gap + Hook (2)

- Although this existence proof (Apache, Linux) means that OSS processes can, beyond a doubt, produce high-quality and widely deployed software, the exact means by which this has happened, and the prospects for repeating OSS successes, are frequently debated
- If OSS really does pose a major challenge to the economics and the methods of commercial development, it is vital to understand it and to evaluate it.

Activity: Read Beginning of "Short and Tweet" by Chen Et Al

Short and Tweet: Experiments on Recommending Content from Information Streams

Jilin Chen[*], Rowan Nairn[†], Les Nelson[†], Michael Bernstein[Δ], Ed H. Chi[†]

[*] University of Minnesota
200 Union Street SE,
Minneapolis, MN 55455
jilin@cs.umn.edu

[†] Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto,
CA 94304
{rnairn, lnelson, echi}@parc.com

[Δ] MIT CSAIL
32 Vassar Street, Cambridge,
MA 02139
msbernst@mit.edu

ABSTRACT
More and more web users keep up with newest information through information streams such as the popular micro-blogging website Twitter. In this paper we studied content recommendation on Twitter to better direct user attention. In a modular approach, we explored three separate dimensions in designing such a recommender: content sources, topic interest models for users, and social voting. We implemented 12 recommendation engines in the design space we formulated, and deployed them to a recommender service on the web to gather feedback from real Twitter users. The best performing algorithm improved the percentage of interesting content to 72% from a baseline of 33%. We conclude this work by discussing the implications of our recommender design and how our design can generalize to other information streams.

Author Keywords
Information stream, recommender system, topic modeling, social filtering.

ACM Classification Keywords
H.5.3: Group and Organization Interfaces.

General Terms
Algorithms, Experimentation

INTRODUCTION
Information streams have recently emerged as a popular means of information awareness. By *information streams* we are referring to the general set of Web 2.0 feeds such as status updates on Twitter and Facebook, and news and entertainment in Google Reader or other RSS readers. Although they have notable differences, the above examples share two key commonalities: (1) they deliver to each user a stream of text entries over time that are personalized to the user's subscriptions, and (2) they allow users to explicitly interact with each other. As information distribution platforms, Twitter, Facebook and Google Reader have all enjoyed great popularity and are drawing ever more new users into them. For instance, according to compete.com's traffic statistics, the total number of people visiting Twitter has been rising from about 6 million per month in January 2009 to over 23 million per month as of July 2009 (http://siteanalytics.compete.com/twitter.com/).

With an abundance of information comes the scarcity of attention [20]. Two user needs arise from attention scarcity: *filtering* and *discovery*. On the one hand, a user's stream will often receive hundreds of items each day, much beyond what users have time to process. Users would like to filter the stream down to those items that are indeed of interest. On the other hand, many users also want to discover useful content outside their own streams, such as interesting URLs on Twitter posted by friends of friends, or relevant blogs in Google Reader that are subscribed by other friends. This discovery task is formidable, given the vast amount of information that are disseminated daily through information stream services.

One approach is to proactively recommend interesting content to users so as to better direct their attention. Google Reader has implemented a discovery feature that recommends interesting RSS feeds, and a number of third-party websites provide filtering or recommendation services for Twitter users. So far there has been little discussion regarding the effectiveness of such solutions, and little is known regarding the design space of information stream recommenders.

As a domain for recommendation, information streams have three interesting properties that distinguish them from other well-studied domains:

(1) Recency of content: Content in the stream is often considered interesting only within a short time of first being published. As a result, the recommender may always be in a "cold start" situation [19], i.e. there is not enough data to generate a good recommendation.

(2) Explicit interaction among users: Unlike other domains where users interact with the system as isolated individuals, with information stream users explicitly interact by subscribing to others' streams or by sharing items.

CHI 2010, April 10–15, 2010, Atlanta, Georgia, USA.

(3) User-generated content: Users are not passive consumers of content in information streams. People are often content producers as well as consumers. Micro-blogging software such as Twitter and Facebook status updates are prominent examples.

In this paper we describe our design and empirical studies of a recommender system built on top of Twitter, called zerozero88, which recommends URLs that a particular Twitter user might find interesting. The recommender we developed is publicly available at www.zerozero88.com.

We chose Twitter as our target platform for several reasons, most importantly because it shares all the common features of information streams described earlier. As a successful platform, Twitter also provides a chance to recruit real users and alleviate their real attention scarcity problems. Finally, Twitter provides a set of public APIs, enabling us to implement and deploy our recommender. We chose to focus on recommending URLs, because the URL represents a common 'unit' of information on the web, and previous research has identified sharing URLs and reporting news as common uses of Twitter [9].

We wish to investigate:

(a) Whether recommender systems can help users find interesting content on Twitter?

(b) What elements lead to an effective Twitter-based recommendation? How can this understanding inform recommender design for other information streams?

To achieve our research goals, we first conducted pilot interviews to elicit early qualitative feedback and refine our system design. After implementing the system, we conducted a controlled field study on our web service to gather quantitative results.

The rest of the paper is structured as follows. First, we discuss how existing research relates to our work. We then provide an overview of information production and information seeking practices on Twitter. We describe the design space of our recommender, and then detail our studies and the results. We conclude with discussions of our findings that may generalize to other information streams.

RELATED WORK
Recommenders as a solution to attention scarcity have been studied for years. Perhaps the most well-known approach is collaborative filtering (CF), which recommends items (such as news stories) using similarities of preferences among users [10]. This approach does not rely on the content of items, but instead requires users to rate items to indicate their preferences, and infers preference similarity from the overlap of rated items across users.

CF recommenders commonly suffer from little user rating overlap early on, known as the "cold-start" problem; a common solution is to use other information like the textual content of the items to be recommended [4, 19].

There is a wealth of research on recommenders that utilize the content of items. Such recommenders are often used in domains where extensive textual content is available for items, such as websites [14] and books [13]. For example, to recommend websites, Pazzani et al. first created bag-of-word profiles for individuals from their activities and then chose websites most relevant to the profile of the individual as recommendations [14]. Because activities of an individual are often insufficient for creating useful profiles, Balabanovic et al. proposed to create profiles not from an individual's activity but from a group of related individuals [4]. This work can be viewed as a hybrid of collaborative filtering and content-based approaches [12].

Recommendations can be generated from explicit social information and social processes as well. For example, Hill et al. described a social filtering recommender on Usenet newsgroups [8]. For each newsgroup, they recommended the most frequently mentioned URLs to that group. Andersen et al. proposed the concept of a trust-based recommender [2]. From a theoretical perspective they discussed ways to employ users' opinions toward other users to compute recommendations. Several other papers investigated the possibility of using social network structures for recommendation [5, 7]. For example, Chen et al. recommended friends-of-friends as potential friends to users of a social networking site, and showed that this scheme is accepted more often than recommending people sharing common keywords [5].

Prior research in developing scalable recommenders [6, 15, 18] is also relevant here because the Twitter ecosystem is so huge that many otherwise useful algorithms become intractable. For example, Sarwar et al. applied clustering algorithms to partition user population, built neighborhoods for users from the partition, and considered only those neighborhoods when computing recommendations [18]. Another relevant work integrated distributed computation techniques for recommendation in Google News [6]. These techniques recursively chop a full problem into sub-problems, so that in the end they can utilize all information in the system despite the large scale of the data.

Outside of academic research, several start-up companies provide information stream filtering or recommendation services, such as my6sense.com, feedafever.com, and MicroPlaza.com. Both my6sense and feedafever filter RSS feeds, including Twitter streams. MicroPlaza recommends personalized news for Twitter users. As start-ups, none of them disclose their approaches or benchmarks.

Because Twitter has both textual and social information available, key parts of the past work described above may be applicable for a Twitter recommender. However, most of them have not yet been implemented and evaluated on Twitter or information streams in general. As a result, it is unclear whether these techniques function well given the differences between their original domains and Twitter, or if some techniques need to be changed to fit the needs of

Twitter users. Our work not only depict the design space for a Twitter recommender, but also better inform designers of recommenders for other information streams.

INFORMATION PRODUCTION & SEEKING ON TWITTER
Twitter describes itself as a micro-blogging service. Users of the site can post short messages, each up to 140 characters, commonly known as tweets. As information producers, people post 'tweets' for a variety of purposes, including daily chatter, conversation, sharing information/URLs and reporting news [9]. Other information streams may have different dominating purposes for posting. For example, on Facebook most of status updates are daily chatter and conversation, while a majority of blog posts in Google Reader may be for information sharing.

As an information seeker, each Twitter user sees a tweet stream when visiting Twitter. A new account only includes tweets posted by one's self; one can include another user's tweets by *following* that user. Throughout this paper, whenever user A follows user B, we refer to A as B's *follower*, and B as A's *followee*.

While some might refer to their followees as their "friends", the following relationship on Twitter is not reciprocal, and does not necessarily imply friendship or even acquaintance between two users. For example, over two million users follow Barrack Obama, few of whom he follows back. Obviously, those people follow President Obama because they are interested in what he says, not because they are personal friends with him. This mechanism of following is different from friendship in other sites such as Facebook, where connections between people are always reciprocal and require confirmation from both sides.

A typical Twitter user picks a list of followees by hand and monitors her personal stream over time. People can also discover information outside their stream in a number of ways, including typing the username of an arbitrary user to see her stream, checking the most popular topics across the whole Twitter site, searching for tweets over the whole Twitter site by keywords, or using one of many third party services that support exploration on Twitter.

DESIGNING RECOMMENDERS FOR TWITTER
We form our design space into three dimensions: (1) how to select candidate URLs, (2) how to use content information, and (3) how to use social information. We illustrate the full design space in Table 1, where each cell is a possible design choice we can make in one of the three dimensions.

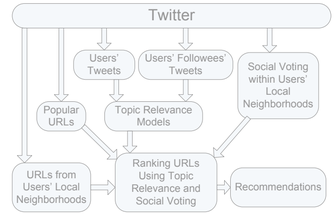


Figure 1. Conceptual Model of the Whole Recommender

We discuss each dimension in the following subsections. Then, we will elaborate on possible system designs and articulate design questions that we answer through empirical studies. The conceptual model of the system that we built is shown in Figure 1.

We did not consider collaborative filtering in our design, as this would require each URL to have feedback from several users to compute reliable recommendations. Moreover, the real-time value of URLs on Twitter requires recommenders to consider new URLs as soon as possible. Under those two constraints, in order to obtain enough feedback for URLs before they become too old to be valuable, the recommender needs a large volume of real-time usage data, as demonstrated in the Google News recommender [6]. However, since we do not have access to large amounts of usage data, this is not a viable option for us. As a result, in formulating our design space, we focused on using content of the tweets and information from social processes.

Selecting the Candidate Set
In building our Twitter based URL recommender, we must first select a limited candidate set of URLs for recommendations due to the high volume of tweets on Twitter. According to TweeSpeed.com, as of September 2009, the number of tweets sent per hour on Twitter ranges from 400,000 to 1,400,000. Scanning those tweets for URLs in real time is a technical challenge. Given limited access to tweets and processing capabilities, our first design question is how to select the most promising candidate set of URLs to consider for recommendations.

Our problem of selecting a candidate set of URLs bears similarities to prior work on scalable recommenders [15,

Design Dimension	Possible Design Choices		
***CandidateSet**: Selecting Candidate Set*	**FoF** (followee-of-followees)	**Popular**	
***Ranking-Topic:** Ranking Using Topic Relevance*	**Self-Topic** score	**Followee-Topic** score	**None**
***Ranking-Social**: Ranking Using Social Voting*	**Vote** score	**None**	

Table 1. The Design Space of the Recommender, Spanning 2x3x2=12 Possible Algorithm Designs

Chen Et Al, "Short and Tweet"

Introduction

- Information streams are increasingly popular.
- With an abundance of information comes the scarcity of attention.
- Need to filter the stream down.
- One approach is to recommend interesting content to users to better direct their attention.

Chen Et Al, "Short and Tweet"

- Recommenders as a solution to attention scarcity have been studied for years.
- Perhaps the most well-known approach is collaborative filtering (CF) - infers preference similarity from the overlap of rated items across users
- CF recommenders commonly suffer from little user rating overlap early on ("cold-start"); a common solution is to use other information
- There is a wealth of research on recommenders that utilize the textual content of items.
- Recommendations can be generated from explicit social information and social processes as well.

The Gap + Hook

- Because Twitter has both textual and social information available, key parts of the past work described above may be applicable for a Twitter recommender.
- However, most of them have not yet been implemented and evaluated on Twitter or information streams in general.
- As a result, it is unclear whether these techniques function well given the differences between their original domains and Twitter, or if some techniques need to be changed to fit the needs of Twitter users.

- Our work not only depict the design space for a Twitter recommender, but also better inform designers of recommenders for other information streams.

Summary: Three Components to a Literature Review

(1) Identify a problem in the world that people are talking about;

(2) Establish a gap in the current knowledge or thinking about the problem; and

(3) Articulate a hook that convinces readers that this gap is of consequence.

Homework (Due Thursday Next Week) : Write a Mini Critique Essay Blog Post

- Consider a top venue in your research area and browse papers recently published there.
- Find a paper with an empirical component that illustrates “best practices” discussed in class so far:
  - Well-structured lit review, with clear problem / gap / hook
  - Precise research questions
  - (Bonus) Articulated theory
- Write a blog post about that paper. Include:
  - Description of problem / gap / hook
  - Research questions
  - Overview of study design / methods
  - Your critique of all of the above

Aside: Citation dos and don’ts

Citation Dos and don'ts (1)

- Don't add citations just to pad the bibliography!
- Prefer:
  - Original paper over secondary source
  - Well-written material over bad
  - Peer-reviewed, top tier venue paper over unpublished / arXiv

Citation Dos and don'ts (2)

- Describe results from other papers fairly and accurately
  - Neither belittle papers, nor overstate their significance

Robinson's theory suggests that a cycle of handshaking can be eliminated, but he did not perform experiments to confirm his results [22].

Robinson's theory suggests that a cycle of handshaking can be eliminated [22], but as yet there is no experimental confirmation.

Citation Dos and don'ts (3)

- References that are discussed should not be anonymous.

Other work [16] has used an approach in which...

Marsden [16] has used an approach in which...
Other work (Marsden 1991) has used an approach in which...

Citation Dos and don'ts (4)

- Quoted material should be an exact transcription of the original text.
  - Permissible changes:

They describe the methodology as "a hideous mess [...] that somehow manages to work in the cases considered [but] shouldn't".

Credits

- Graphics:
  - Dave DiCello photography (cover)
- Content:
  - Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In Guide to advanced empirical software engineering (pp. 285-311). Springer, London.
  - Varpio, L., Paradis, E., Uijtdehaage, S., & Young, M. (2020). The distinctions between theory, theoretical framework, and conceptual framework. Academic Medicine, 95(7), 989-994.
  - Lingard, L. (2015). Joining a conversation: the problem/gap/hook heuristic. Perspectives on Medical education, 4(5), 252-253.
  - Lingard, L. (2018). Writing an effective literature review. Perspectives on medical education, 7(2), 133-135.
  - Justin Zobel, Writing for Computer Science (3rd Edition). Springer, 2015