Emerging Themes in Software Engineering Research

Avinash Kak

Purdue University

July 20, 2010

Presented at SETLabs, Infosys, Bangalore, India

NOTE: This is a cleaned-up and slightly modified version of the talk I gave at Infosys. Most of the changes I have made were in response to the feedback I received during and after the talk.

Abstract

Much of the vocabulary of software engineering today has become outdated as a result of the following technological changes of recent years: (1) Many applications today run on platforms that parallelize the execution of software. This parallelism may be synchronous, as for the case of multi-core machines, or asynchronous, as for the case of SoA and other similar implementations. (2) Many applications, especially those that are intended for mobile devices, are rich in human interactivity that can give unanticipated values to program variables. And (3) The communication revolution has made it possible to create and maintain software in a geographically distributed manner. This talk will go into the nature of difficulties created by each of these developments and make the claim that software engineering, as it is practiced today, is mostly silent regarding these difficulties. In particular, the talk will point out that today's software engineering does not have much to say regarding the testability and maintainability characterization of a software library in the space spanned by the variables required for synchronization and the variables instantiated by human interactivity. With regard to the global and distributed production and maintenance of software, this talk will point to the need for "semantic" search and retrieval tools that can be used to simulate a team of programmers working together in close physical proxity for synergistic production of code. This talk will also briefly review the progress made at Purdue in the development of such tools.

Outline

Outline

A Bit About Purdue University

- One of the Big Ten universities in the US
- Located about 2.5 hours by road from Chicago and about an hour's drive from Indianapolis
- Operating budget: Roughly $2.2 Billion per year
- Research expenditures per year: around $400 Million

A Bit About Purdue University (contd.)

- One of the largest graduate schools in engineering
- One of the largest producers of Bachelor's degree holders in Electrical and Computer Engineering
- Around 40,000 students at its main campus in West Lafayette

Outline

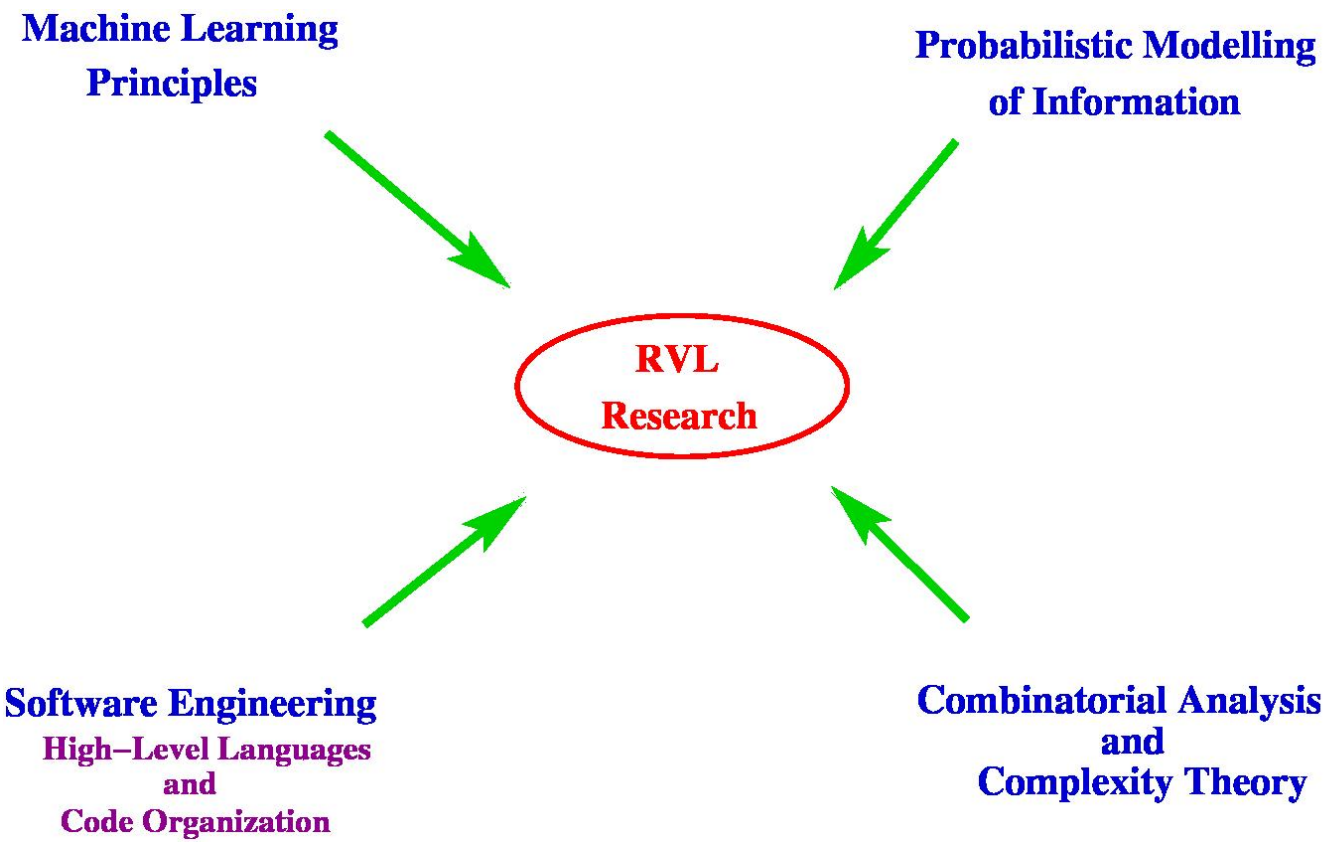
A Bit About the School of Electrical and Computer Engineering

- About 900 undergraduate students
- About 650 graduate students
- One of the largest producers of Ph.D.s in Electrical and Computer Engineering (about 60 Ph.D's a year)
- Annual budget including research expenditures: around $40 Million

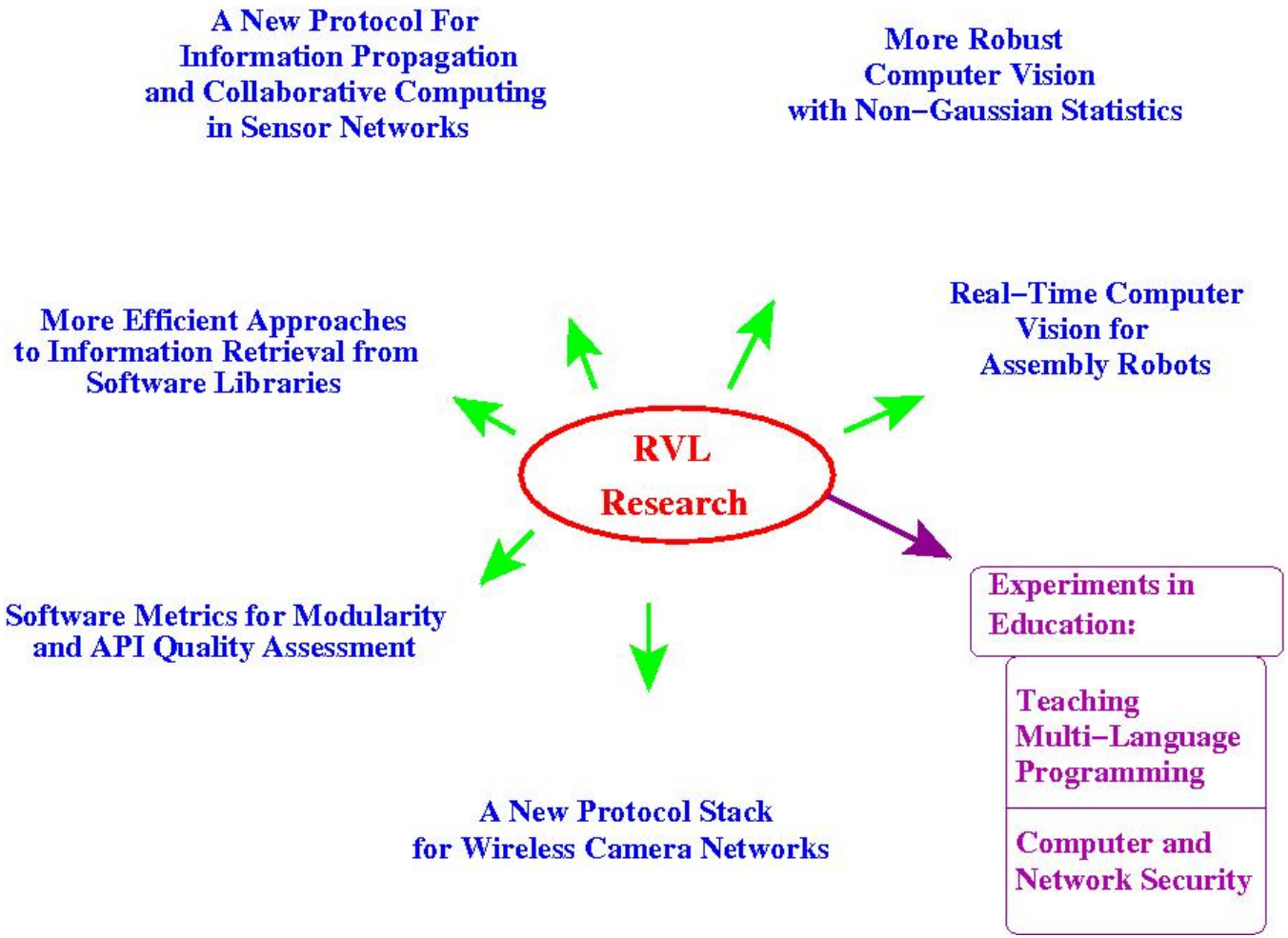
Outline

The RVL Ideas Factory

RVL (Purdue Robot Vision Lab) has existed for over 25 years and has produced 42 Ph.D.'s.



What New Ideas Have Come Out of the RVL Ideas Factory During the Last Five Years?



Outline

Software Engineering — The Traditional Model

Requirements Analysis

→ Specification

→ Design and architecture

→ Coding

→ Testing

→ Documentation

→ Maintenance

Software Engineering — As Alternatively Practiced

- Because of the problems with the traditional model, an alternative model — the agile model — has come into fairly widespread use
- In the agile model, software is created by a group of developers working together as a close-knit team through all phases of the development
- The informal nature of the collaboration in the agile model allows for the evolving software to be changed quickly in response to any changes in the requirements

Software Engineering — As Alternatively Practiced (contd.)

- However, the quality of the software produced depends much on the personalities of the people involved in designing and creating the software
- Usually, software starts out as a quick-and-dirty implementation by the team and then takes off from there through interactions with the clients
- **The quality of the software produced in this model is often a function of the deadline constraints imposed by the clients**

Software Engineering — As Practiced Today (Reality Check)

- All software is based on certain assumptions regarding its usage by the clients
- The usage assumptions commonly refer to:
  - the reasonableness of the values entered by humans in GUI based application
  - the availability of certain data files that contain data in certain formats
  - in networked applications, the assumptions commonly refer to the network latencies regarding data availability
  - in applications with database server backends, the assumptions refer to the latencies and the format of the data made available by the server
  - .... and many more such assumptions

Software Engineering — Reality Check (contd.)

- Regarding the values entered by humans in GUI based applications, it is now fairly common to provide a data-checking back-end to the GUI that makes sure that data-entry assumptions are valid
- However, the same cannot be said for all the other usage assumptions that a software library is implicitly based on
- In general, it would add too much of a cost burden to a software project if the source code included a test for ascertaining the validity of every usage assumption and if the source code included the appropriate remedial actions when the tests fail

Software Engineering — Reality Check (contd.)

- In the current models of software design and production, any violation of the implicit assumptions on which the software is based **is detected by the program crashing**
- The remedial action when a program crashes consists of the client making an urgent call to the tech support at an organization like Infosys
- In and of itself, there is nothing wrong with this approach to attaining cost-effectiveness in software projects — the important thing is to acknowledge the reality of this approach and its inherent limitations

Software Engineering — Reality Check (contd.)

- All software is based on certain assumptions regarding the hardware it will be run on
- With hardware becoming increasingly multi-core, software developed without attention to parallel processing capabilities of the hardware will typically under-utilize the hardware
- When software is designed to operate in a multi-threaded or multi-processed manner, how the threads are mapped to the cores becomes an important issue

Perils of Ignoring Performance Enhancements Made Possible by Multi-Core Machines

At Purdue when we replaced the following sort of a C++ call

```
std::for_each( a_list.begin(),
               a_list.end(),
               some_function )
```

by the following call

```
__gnu_parallel::for_each( a_list.begin(),
                          a_list.end(),
                          some_function )
```

in an application that was inherently amenable to parallelization, we were able to speed up the application by a factor of 2 on a dual-core machine

Software Engineering — Reality Check (contd.)

- Software that is critically dependent on multi-threading is always based on assumptions regarding the thread-safety of the function calls
- Multi-threading of software is generally very important to applications that involve human interactivity
- Thread safety is compromised when different threads of execution are allowed to access variables that are global with respect to the scope of each thread

Software Engineering — Reality Check (contd.)

SUMMARY

- Even the best software is bug-free only to the extent the client-related usage assumptions, network and database latency assumptions, file-formatting and file-data-availability assumptions, the concurrency-related assumptions, etc., hold
- In general, it is not cost-effective to check for the validity of all such assumptions and to provide code for the remedial measures to be taken when the assumptions fail
- **Therefore, in general, the notion of completely bug-free software is just a myth**

Software Engineering — A Reality Check (contd.)

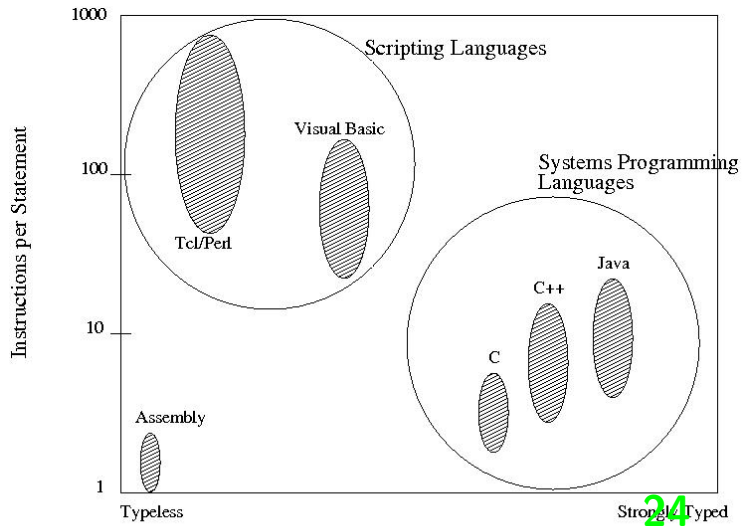
SUMMARY

- Astute developers are usually aware of all the assumptions under which the software can be considered to be bug-free, but will generally not state them in the documentation
- To admit that the software being shipped out is only conditionally bug-free obviously makes no business sense

As a Small Case Study in the Implicit Assumptions, Consider My Open-Source Perl Module for Automatic Data Clustering

- The name of the module file is Algorithm-KMeans (Version 1.20)
- Posted at the open-source site `http://www.cpan.org`
- It is over 1000 lines of Perl code (roughly equivalent to 10,000 lines of C code — See the conversion figure below by Dr. Ousterhout)
- The module is based on certain assumptions regarding how the data files will be configured and what call syntax the clients will use

If I had to put in a check for every possible way a data file could create problems for the module and every possible manner of calling the various functions in the module, that would have doubled the module size



Outline

What Are the Needs of Tomorrow in Software Engineering?

The Software Engineering needs of tomorrow are being driven by several on-going technological developments:

- Ever increasing utilization of concurrency and parallelism in general purpose software systems (not just GUI software) (synchronous parallelism)
- Greater dependence on asynchronous coordination between computing devices (Example: SoA)
- Greater number of applications that involve human interactivity (especially true for mobile applications)
- Multi-language software systems

And Then There Are Needs That Are Specific to Large Software

It is now common for the software libraries meant for enterprise applications to run into millions of lines of code. The large size creates its own set of the needs of tomorrow:

- Needs related to the organization of software from the standpoint of better testability, maintainability, etc. (the modularization problem)
- Giving a hierarchical organization to the modules, again for better testability and maintenance (the problem of creating super-modules)
- Developing semantic search engines so that the developers can have more productive interactions with large software systems (a critical need when large software is developed at geographically distributed locations)

Complications That Arise from Parallelism

Parallelism (both synchronous and asynchronous) creates the following sort of issues with regard to **testing and maintenance**:

- Using parallelism for non-SIMD applications can introduce too many synchronization points in large software
- Often the synchronization is subject to the participating processes fulfilling certain conditions
- A synchronization condition will usually consist of a predicate involving global program variables
- A synchronization condition may also involve estimates of the network and database latencies, as illustrated on the next slide

Complications That Arise from Parallelism (contd.)

- To elaborate, you may have one thread of execution processing the credit-card information supplied by a user and, at the same time, another thread of execution checking on the credit history of the user
- The synchronization here could be a simple predicate that succeeds when the evaluations in both threads succeed
- Or, in more complex scenarios, the synchronization predicate could also take into account the network latencies if both threads are making network calls
- In a static analysis of the code, the variables in such predicates and the constraints placed on them would need to be subject to testing and maintenance. Obviously, any synchronization predicates that depend on estimates of network latencies would need to adjusted should the network traffic conditions change.

Complications That Arise from Parallelism (contd.)

- The upshot is that each synchronization condition could potentially involve variables for testability and maintenance
- Software engineering of today is not equipped to account for such variables in its testing and maintenance protocols

Complications That Arise from Human Interactivity

Human interactivity creates the following sort of issues that are outside the purview of Software Engineering as it is understood and practiced today:

- Human interactivity can give previously unanticipated values to program variables
- Such unanticipated values can be checked (and discarded) for simple modes of data acquisition (such as form-filling, ATM-style sequenced entries, etc.)
- Catching unanticipated values is much more difficult for the newer interactive modes made possible by the multi-touch screen technology that allows for multi-finger taps, flicks, drags, pinches, etc.

Complications That Arise from Human Interactivity (contd.)

- The unanticipated values acquired for program variables during human interaction can place the runtime in a state that the software was not designed to handle

- For complex software, the large number of such program variables create combinatorial issues for software testing and maintenance that Software Engineering does not know how to deal with

Much Needed: New Software Engineering Vocabulary for Quantifying End-User Experience

The vocabulary of software engineering today is inadequate with regard to testing the software from the standpoint of end-user experience:

- The systems-oriented thinking of the bygone era still dominates the methods for testing software
  - Software is still thought of as a black box that must produce certain specified responses for certain specified inputs
- It is easy for software developers to believe that black-box testing is sufficient unto itself in order to establish that software is good
- In human-interactive software, the developers/tester/maintainers often have difficulty accepting the fact that humans are capable of exercising the software in ways not anticipated at the design time

Also Much Needed: New Software Engineering Vocabulary to Deal with the Large State Spaces Created by Concurrency and Human-Interactivity Variables

- For large applications, the space spanned by all of the synchronization variables required for parallel execution and the program variables that can be instantiated by human interactivity can be very large

- **Software engineering today is silent about how to create statistical testing protocols for coping with the combinatorial issues related to testing in the vary large space mentioned above**

Also Much Needed: Decision Supports Systems for Software Designers

- Software Engineering needs to borrow from Decision Support Systems in the Management Sciences so that the designers can assess the consequences of bad decisions
- Every stage of software development, all the way from high-level conceptualization to low-level implementation, involves choice
- Often this choice is simply a personal choice made by the designer, sometimes made arbitrarily, sometimes made on the basis of code-reuse considerations, sometimes on the basis of maintainability and testability considerations, etc.
- We need decision support tools that would allow the consequences of the wrong decisions to be assessed

Outline

Modeling the Software Libraries with Regard to the Largely Hidden Program Variables

- As mentioned previously, concurrency, parallelism, and human interactivity can create a large state space for the characterization of a software library **with regard to its testability and maintainability**

- **That leads us to the following question: What is the best way to model a software library so that we can easily assess the testability and the maintainability of the library through static analysis of the code?**

Modeling Large Software Libraries .... (contd.)

- In any deterministic approach to tackling the problem for large and complex software, we may have to deal with the combinatorial issues caused by the large number of the variables related to synchronization and human interactivity
- Unfortunately, any deterministic approach to modeling large software in the large state space is bound to fail because of exponential complexity
- **That is, in general, a deterministic approach to software testing and maintenance with regard to the variables required for the proper functioning of parallel execution and human interactivity will not be computationally feasible**

Modeling Large Software Libraries .... (contd.)

In my opinion, probability theory offers the best hope of modeling a software library with regard to its testability and maintainability in the large state space spanned by the variables

- Intractability caused by combinatorial explosions is best handled with probabilistic modeling
- If we agree that probability theory is the way to go for modeling software behavior, that would have a huge implication with regard to the protocols for the testing and maintenance of such software systems.
- That would imply that the testing and maintenance protocols of the future must adhere to the principles of **random input testing**

Regarding Random Input Testing

- In principle, the notion of random input testing goes all the way back to Norbert Weiner (and is described in his classic books)
- However, its formulation as used today is not appropriate for the very large state spaces needed for large and complex software
- Random input testing as used today has worked well for testing hardware systems where the inputs and the outputs generally form small state spaces

Some Additional New Research Directions

- Testability and maintainability characterization of large software through modularization and other metrics
  - The cost of maintaining and upgrading a code base that runs into millions of lines of code grows exponentially as the modularization quality of the code deteriorates (this speaks to the need for continued research in modularization metrics)
- Usability characterization of large software through API metrics
  - The larger the number of modules and the more complex their functionality, the greater the need for high-quality APIs (this speaks to the need for continued research in API metrics)

Some Additional New Research Directions (contd.)

- Research in methods for task disassembly and code re-assembly in large software projects
- Research in decision support tools for designers and architects so that they can better see the consequences of wrong decisions
- Development of platforms that facilitate collaborative production of code when developers work together across time zones

Outline

Global Software Delivery/Maintenance Needs More Powerful Tools for Interacting with Code

- In the global software production and maintenance models, it is not possible for software developers based in different geographies and in different time zones to interact with one another for synergistic production and maintenance of a large code base in the same way as when the developers work out of the same room or the same building

- So the challenge of global production/maintenance of large software involves creating information-retrieval tools that a developer can use to interact with an evolving code base to elicit the same kinds of answers that he/she used to get when working in close proximity to the other developers

With Semantic Search Tools, a Developer Would be Able to Pose Questions Like:

- "I have written a new piece of code. Is someone else already implementing the same functionality?"
- "What are all the classes that implement a certain functionality?"
- "I have written this piece of code and want to update the documentation. Where in the documentation should I make the changes, or give pointers?"
- "I have modified this piece of code. Where else should I update so that the changes are consistent?"
- "Who else is working on the code that is semantically related to the code segment I am working on?"

Productivity Enhancement with Semantic Search Tools .... (contd.)

- Questions of the sort shown on the previous slide also arise in automated or semi-automatic bug localization
- These sort of questions go way beyond what can be accomplished with string search, even when you include the power of regular expressions in string search.

Productivity Enhancement with Semantic Search Tools .... (contd.)

Applied to software libraries, the extent to which semantic search can be expected to succeed depends on how you model a software library:

- Such models may be deterministic (VSM, LSA), as in the traditional IR based retrieval engines
- Or, such models may be probabilistic, as in the more modern IR retrieval engines and the recently developed semantic search tools for text corpora (LDA)

Probabilistic Modeling for Information Retrieval

- One of the advantages of the probabilistic models is that the model size increases sub-linearly with the corpus size
- The probabilistic models fit a multinomial distribution (or a mixture of multinomials) to the corpus
- If a single multinomial is fit to the corpus, the model is called the unigram model
- When a mixture of multinomials is fit to the corpus, it is done through what is referred to as Latent Dirichlet Allocation (LDA)

Evaluation of Semantic Search Tools — A Big Headache for the Case of Software Libraries

- To successfully evaluate a search and retrieval algorithm, one must have access to what is known as a ground-truthed corpus
- That is, in order for a computer to measure the precision and recall stats for the retrievals carried out in response to different queries, the computer must have prior knowledge of what documents are relevant to which queries
- There exist a large number of such ground-truthed corpora for text, **but not yet for software libraries**

Evaluation of Semantic Search Tools for Software (contd.)

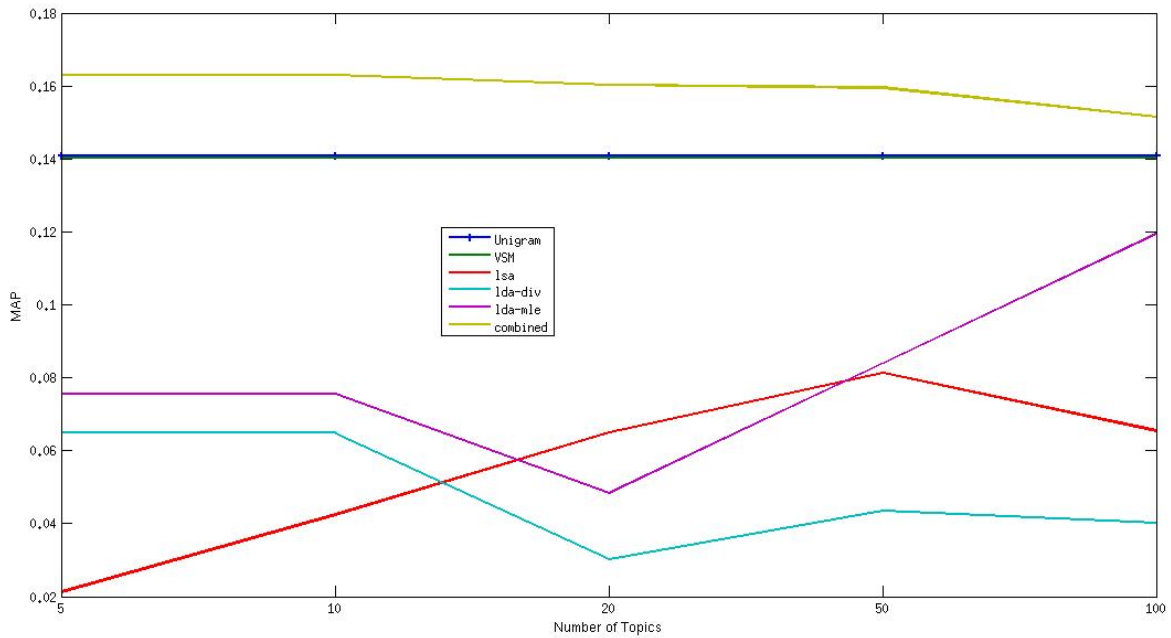
- In the absence of large ground-truthed software libraries, the best we can do at the moment is to evaluate our semantic search tools on the relatively small libraries that are available
- The JEdit software system is one such library
- JEdit is an open-source library that is used by thousands of programmers world-wide for developing Java applications
- For JEdit, the queries are the bug reports and the relevant documents the files mentioned in the bug fixes.
- Both the bug reports and the bug fixes are publicly available at the JEdit development website

Comparing Four Models for Semantic Retrieval

Using the JEdit libraries and the associated bug reports, at Purdue we have compared the performance of the following four models with regard to their retrieval performance:

- The Vector Space Model (VSM)
- The Smoothed Unigram Model
- The Latent Semantic Analysis Model (LSA)
- The Latent Dirichlet Allocation Model (LDA) using:
  - KL-Divergence between the topic distribution in a query and a document to measure the similarity between the two
  - Measure the similarity of a query to a document by using the LDA model to estimate the likelihood of the query given the document
  - Do the same as the above but use a combination of the LDA model and the unigram model

A Comparison of Mean Average Precision (MAP) Values for Four Different Models for Semantic Retrieval for JEdit Software Libraries

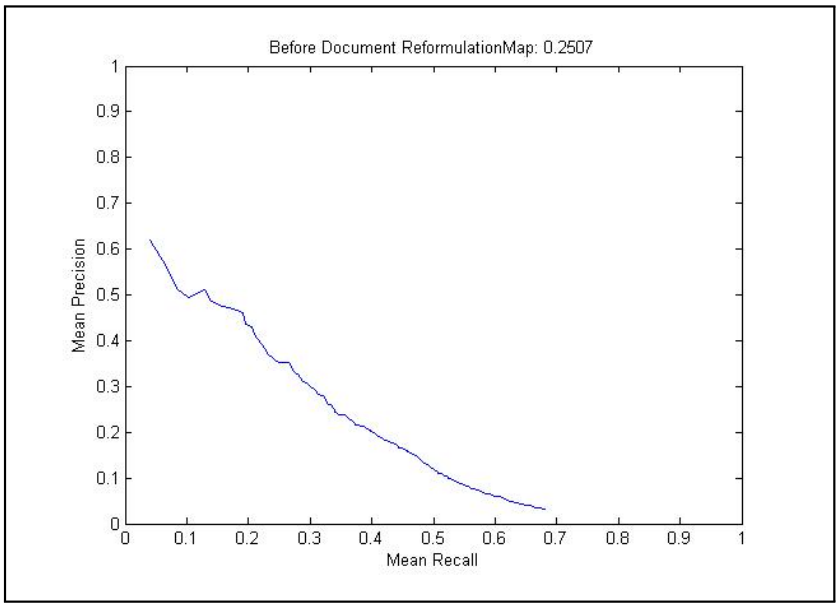


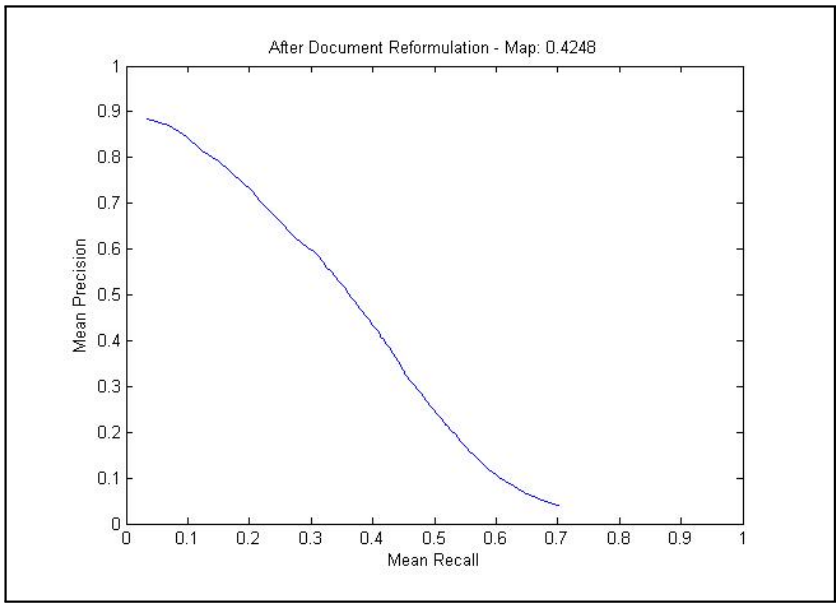
A Probabilistic Model Can be Fine-Tuned Through Relevance Feedback

- Relevance feedback means that, upon seeing the retrieval results for a query, the human marks the returned items as relevant or irrelevant
- A domain expert interacts with the system by supplying queries to it and, subsequently, by marking the retrieved documents as relevant and/or irrelevant vis-a-vis the queries
- The system makes modifications to the underlying statistical model of the software library
- The improvements to a model are achieved through methods that are referred to as document reformulation

Improvements in Semantic Retrieval from the JEdit Software Library

Shown at left is the Precision vs. Recall curve before relevance feedback, and at right the same after relevance feedback:





Outline

In Summary (About What is Missing in SE Today)

- I expect new research directions to emerge in Software Engineering as we come to grips with the design and architecture complexities introduced by concurrency and parallelism
- I expect the current interest in the use of probabilistic models for semantic search and retrieval to become even bigger as the software development, delivery, and maintenance become even more global and even more distributed
- I expect the research to continue in the development of algorithms for code modularization and for the measurement of the quality of a given modularization

In Summary (contd.)

- I expect the issues related to API design to become even more prominent as software modules acquire greater complexity on account of concurrency and parallelism
- I expect that there will be greater focus on the development of methods for software task decomposition and code re-assembly
- I also expect to see the emergence of decision support systems for designers and architects so that they can better see the risks associated with a bad design choice

Outline

Epilogue: Should Indian Software Houses Emulate Western Models of Software Production?

- Western models of code production came into existence during a period when the mystique of computing was still new and working with computers was considered to be cool
- But that is no longer the case
- In most of the world's prosperous countries (US, Europe, Japan, South Korea), programming is now considered to be one of the least desirable jobs. In all such societies, programmers are frequently referred to as "nerds" at best and "code monkeys" at worst

Epilogue (contd.)

- These pejorative characterizations are a consequence of the Western models of code production that require programmers to stay chained to their computer screens for the better part of each day, day after day
- This mode of work is dehumanizing for obvious reasons and not sustainable over the long haul
- The agile model of code production (also a Western model) alleviates some of the problems with the more traditional models by injecting a strong human component into the process

Epilogue (contd.)

- However, there is a certain "challenge and response" aspect to the agile model of code production, which may inject into the process momentary thrill and excitement, but this can also result in excessive programmer fatigue over the long haul
- The agile model cannot be scaled up for very large software projects that involve millions of lines of code
- Ignoring the lack of scalability, I believe that India can transcend the agile model and other models of software production that came out of the Western mindset

Epilogue (contd.)

- I believe that whereas the Western mindset begins and ends its day in the pursuit of individualism, the Indian mindset can transcend individualism and also see the virtues of harmony
- I do not mean to say that the ratio of self-centered people in India who are only looking out for themselves is less than it is in the West
- **What I mean to say is that, on the average, a person of Indian sensibilities is more likely to see the benefits of harmony over rank individualism**

Epilogue (contd.)

It is therefore up to the Indian software houses to experiment with new collaborative approaches to software production that value the human spirit in the best traditions of the Indian values and the Indian ethos

Thank you all!