The Importance of Machine Learning to the SCUM of Large Software

Avinash Kak

Purdue University

January 2, 2014

Presented at Infosys, Bangalore, India

Abstract

Software engineering concerns itself, on the one hand, with the production of code that meets the requirements of a given application, and, on the other, with ensuring that the code is searchable, changeable, usable, and maintainable. It's the second major concern of software engineering — Searchability, Changeability, Usability, and Maintainability (SCUM) — that will be impacted significantly by machine learning in the years to come. Machine Learning allows us to create compact deterministic and probabilistic models of software libraries that can subsequently be used for interacting with the libraries in several different ways that are important from the standpoint of SCUM. For example, with regard to Searchability and Maintainability, we can now create models (ranging in complexity from simple bag-of-words based models to more sophisticated models based on Markov Random Fields) that can be used for high-precision bug-localization, concern location, duplicate incident-report detection, bug and incident-report triaging, and so on. With regard to Changeability and Usability, we know that the former depends on the quality of modularization of the software, and the latter on the usability of the APIs of the modules. Recent research has demonstrated that both — meaning, the quality of modularization and the usability of the APIs — can be subject to quantitative characterization. Applying classification and prediction algorithms to these characterizations can help us distinguish between different software packages on the basis of their Changeability and Usability properties.

Outline

Outline

Software Engineering — Two Main Concerns

Primary Focus of SE: Producing software that meets the requirements specifications.

The last twenty years have produced several methodologies for software production. These include the **Waterfall methodology**, the **Agile methodology**, and so on.

A Focus Critical to the Lifetime Success of the Above: Making sure that the software that is produced scores high on SCUM. **But what's SCUM?**

What's SCUM?

S: Searchability

C: Changeability

U: Usability

M: Maintainability

Let's First Talk About the 'S' in SCUM — Searchability

- If we were asked to identify one element that is common to a wide range of our everyday human activities, that would be "**on-line search**."
- As a result, instantaneous **on-line search** has now become an important part of our reflexive behavior.
- **Even without thinking**, we now bring up an app on our mobile device or on a laptop to get us instantaneously what it is that we are looking for. We could be looking for how to get to a restaurant, how to spell a long word, how to use an idiom properly, or about any of a very large number of other things.

Code Search Permeates the Process of Creating New Software

- With instantaneous search having become such a deeply ingrained part of our nature, **it is not surprising that when we write code we constantly invoke search tools to not only look at the documentation related to our project but to also seek out constructs similar to what are creating.**

- **Search is not always about looking at other people's code.** When creating new constructs in code, the developers often revisit similar constructs created by them previously in order to get a quick fix on what it is they need to do differently now in relation to what was done previously in a similar situation.

Searchability of Software Goes Beyond Using Easy-to-Remember Names for Variables, Methods, etc.

- Our intuition suggests that the searchability in a software library would depend primarily on the extent to which the developers used meaningful names for the classes, methods, variables, etc., that reflect the usage of those names.

- Unfortunately, as experience has shown, while using meaningful names for the variables, classes, methods, etc., is necessary, **it is not sufficient for large software in order to make it searchable**.

Code Search Based on Just String Matching Does Not Work for Large Software

- Using straightforward string-matching based search (as with, say, 'grep' and other similar tools) results in too much search noise that is difficult to wade through. Simple tools are incapable of giving us results with any usable retrieval precision. They cannot ignore nondiscriminating terms.

- Using regex based string matching tools for searching in a large library is like locating a runtime fault by examining the core dump. It can be done, but, in most cases, it would be frustrating.

- **Efficient search depends on our ability to create a compact deterministic or probabilistic model for a software library.**

Let's Next Take Up the 'M' in SCUM — Maintainability

- As you all know, a large part of software maintenance consists of fixing faults in response to bug reports.
- How well the different parts of a bug fixing process — triaging, duplicate bug-report detection, fault reproduction, etc. — can be executed depends, on the one hand, on how well the code is modularized, and, on the other, on **how searchable the code base is vis-a-vis the bug reports.**
- The reason I took up 'M' after 'S' is because both searchability and maintainability place very similar demands on what they need from Machine Learning — as you will soon see.

Let's Now Talk About the 'C' in SCUM — Changeability

- The different parts of a library must not be so intertwined that a change to one part makes it necessary to change a large number of other parts.
- Ease of changeability is closely related to the **quality of modularization** and whether or not the modules interact only through their published APIs.
- **Fortunately, we now have excellent metrics for evaluating the quality of modularization for both the procedural code and large object-oriented code.**

Finally, We Get to Talk About the 'U' in SCUM — Usability

- What makes software usability a complex issue is that it relates to the interface between the human and the software. A programmer brings to bear much **subjectivity** to his/her interaction with a software library.
- With regard to this subjectivity, an important thing to note is that a software library is experienced mostly through the APIs exposed by the library. Fortunately, we as a community can now list several commonly held beliefs about what constitutes a good API.
- **And, what's even more significant, we now possess metrics that give us a quantitative characterization of API usability.**

Outline

Searchability and Maintainability: Both Require Us to Create a "Model" for the Software

- If direct string-matching based search is not a scalable solution to the problem of identifying the files that contain a code snippet or that are impacted by a bug report, we are faced with the following issue:

  Is it possible to create a compact representation for the software in such a manner that the size of the representation grows sublinearly with the size of the library, and can we, at the same time, get a good 'S' and 'M' solution for the software?

- **The answer to this question is yes.** In general, we refer to these more compact representations as **models** for a software library.

Deterministic Models Based on the Bag-of-Words Assumption

The Vector Space Model (VSM): Each file is represented by a vector whose elements represent the frequencies of the different terms in the file. **Think of this vector as a histogram.** This histogram must have a bin for every term that is important in the library.

The Latent Semantic Analysis Model (LSA): We take the VSM model one step further by carrying out an SVD of the matrix formed by all the file-based vectors.

- With LSA, the library can be represented by a rather small number of the eigenvectors that are retained.

Deterministic Bag-of-Words Models (contd.)

- **LSA results in a highly compact representation** of a software library since each file is now represented by a small number of scalar coefficients, each coefficient being the dot-product of the vector for the file and one of the eigenvectors.

- In both VSM and LSA, we can get rid of the non-discriminating terms (*these are terms that occur in too many files*) by multiplying the file-based terms frequencies with what is known **as the inverse-document-frequencies**.

- VSM and LSA are deterministic examples **Bag-of-Words (BoW) models** because they pay no attention to the **order and proximity** relationships between the different terms in the files.

Probabilistic Models Based on the Bag-of-Words Assumption

The Smoothed Unigram Model (SUM): This is a probabilistic version of VSM. We now treat the histogram of the term frequencies as a first-order probability distribution.

The Latent Dirichlet Allocation Model (LDA): You now inject additional "hidden variables" into the probabilistic model of the library. We refer to these variables as **topics**. You consider the content of each file to be a mixture of topics. Each topic is a probability distribution over the terms.

- For the LDA case, if we have K topics, **each file is represented by a K-dimensional probability vector**, each element of which is the proportion in which the corresponding topic is represented in the file.
- SUM and LDA are **probabilistic examples of BoW models**.

Outline

The Notion of a Query

- For code search, a query can be a code fragment.
- For code maintenance, the query is a bug report.
- **We represent a query in exactly the same manner as the files in the library.** For deterministic BoW, that would be by a vector of the term frequencies. For probabilistic BoW, that would be by a probability distribution that represents the query term frequencies.

Retrieving the Relevant Files — Deterministic Case

- With VSM modeling of a software library, you **compute the cosine similarity** between the query term-frequency vector and the vectors that represent each of the files.
- With LSA modeling, **you project the query vector into the eigenspace that represents the library.** Subsequently, you can use one of several methods for locating the files that are closest to the query in the subspace.
- For example, with LSA, you could use the **NN algorithm** to identify the files most relevant to the query. Each file is a point in a low-dimensional subspace spanned by the eigenvectors. You select the files whose points are closest to the query point in the subspace.

Retrieving the Relevant Files — Probabilistic Case

- The term probability distribution for the query may be compared with the term probability distributions for the individual files with any of a large number of similarity criteria. **A commonly used criterion is the K-L divergence.**

- For the LDA case, you can compare the query with each of the files by **comparing the topic mixture weights in the query vis-a-vis the topic mixture weights for the files.** You can again use the K-L divergence for this comparison.

- For the LDA case, you also have the choice of comparing directly the first-order term probability distribution for the query with similarly first-order term probability distributions you can obtain for each of the files from their topic mixture distributions.

Some Very Practical Observations Related to BoW Modeling

Two conclusions you are likely to draw from the results shown on the next few slides:

- **The simpler the better**. As you'll see, simple models like VSM and SUM outperform the more complex models like LSA and LDA.

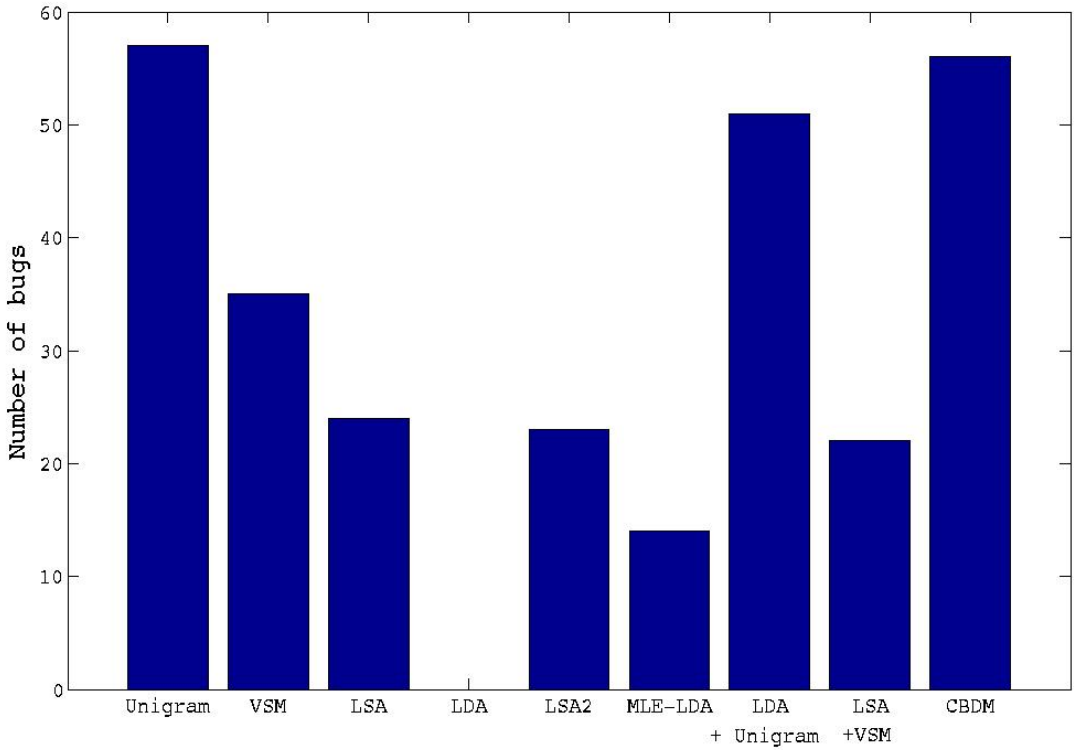
- You get superior results **if you augment a simple model with smart heuristics rather than use a complicated model to begin with.**

Some Bug Localization Results with the Bag-of-Words Models

The results shown on the next two slides were obtained for the iBUGS dataset created by Dallmeier and Zimmermann for the AspectJ library. Its various parameters are:

Software Library Size (Number of files)	6546
Lines of Code	75 KLOC
Vocabulary Size	7553
Number of bugs	291
Avg No of relevant files/ bug	1

The height of the bars shows the number of queries (bugs) for which at least one relevant source file was retrieved at rank 1. (Rao and Kak, MSR'2011)



Adding Oomph to a BoW Model with File Metadata

- We now know how to increase the retrieval precision of the probabilistic BoW models by exploiting the **metadata that is associated with the files by version control systems like SVN.**
- *If the version control system is telling us that a set of files is being modified simultaneously in response to a change request or a bug report, one can argue that we should somehow increase the odds of those files being retrieved together in response to certain bug reports.*
- Sisman and Kak (MSR'2012) **presents a well-principled probabilistic framework** in which the prior probabilities associated with a file to be buggy are calculated on the basis of its modification history and bug-fixing history.

Adding Oomph to a BoW Model with File Metadata (contd.)

- The framework In Sisman and Kak (MSR'2012) makes a distinction between two types of changesets (svn:changeset): Feature Implementation **(FI)** changesets and Bug Fixing **(BF)** changesets.
- The FI changesets are used for calculating the *Modification History Based Priors* and the BF changesets for calculating the *Defect History Based Priors*. These priors are then used in a Bayesian framework to estimate the relevance of a file to a bug.
- Since it is known that the files that have not been changed in a long time are less likely to be bug prone, the framework **also incorporates a time decay factor** in the probability update formulas to estimate the relevance of a file to a bug.

Some Parameters of the Dataset Used for Demonstrating Improved Retrieval Precision When We Take Into Account File Modification and File Defect Histories

Table: *AspectJ Project Properties*

K	$\|FI\|$	$\|BF\|$	$\|Q\|$	Analysis Period
6,271	5,165	1214	291	2001-01-16 - 2008-10-06

where

- K: the number of changesets extracted from SVN
- FI: the number of Feature Implementation changesets
- BF: the number of Bug Fixing changesets
- Q: the number of bugs

Retrieval Results When File Modification and Defect Histories are Taken into Account (Sisman and Kak, MSR'2012)

Baseline:

Model	MAP	P@1	P@5
DLM	0.1349	0.1271	0.0851
tf-idf	0.1264	0.1062	0.0712

With Modification History and Defect History Based Priors:

Model	MAP	P@1	P@5
DLM	0.1660 (+23%)	0.1546 (+21%)	0.0928
tf-idf	0.1651 (+30%)	0.1409 (+32%)	0.1065

When a Time-Decay Factor is Also Included:

Model	MAP	P@1	P@5
DLM	0.2041 (+51%)	0.2268 (+78%)	0.1423
tf-idf	0.2258 (+78%)	0.2646 (+148%)	0.1512

Adding Oomph to a BoW Model with Query Reformulation

- Another way to improve the retrieval performance of a BoW model is through **Query Reformulation (QR)** that judiciously adds additional terms to a user-supplied query to make it more discriminating.

- In QR, you examine a subset of the top-ranked files that are retrieved for the original query. **The system then automatically adds some of the terms extracted from these files to the original query.** The reformulated query is then used for retrievals shown to the user.

- In the **Source Code Proximity (SCP)** approach to QR, the additional terms injected into a query are those that are deemed to be "close" to the original query terms on the basis of positional proximity.

The Dataset Used for Demonstrating the Power of QR Based on SCP

Table: *Projects Evaluated for Demonstrating the Power of SCP for QR*

Project, Description	Language	$\|B\|$	$\#Files$	$\#Terms$	Analysis Period
ECLIPSE v3.1 IDE	Java	4,035	12,825	19,955	2001-04-28 - 2010-05-21
CHROME v4.0, WEB browser	C/C++	358	8,420	349,958	2008-07-25 - 2010-05-20

A Comparative Evaluation of the Different QR Methods (Sisman and Kak, MSR'2013)

Table: *Retrieval accuracies with three QR methods*

	Eclipse					Chrome				
	MAP	P@1	P@5	R@5	R@10	MAP	P@1	P@5	R@5	R@10
SCP	**0.2296**	**0.1893**	**0.1011**	**0.2849**	**0.3739**	**0.1820**	**0.1788**	**0.0933**	**0.2021**	0.2775
RM	0.2154	0.1670	0.0970	0.2729	0.3631	0.1666	0.1480	0.0844	0.1966	**0.2853**
ROCC	0.2163	0.1713	0.0961	0.2743	0.3663	0.1564	0.1397	0.0793	0.1816	0.2779
Baseline	0.2089	0.1653	0.0921	0.2632	0.3559	0.1535	0.1453	0.0832	0.1838	0.2601

SCP: Source Code Proximity
RM: Relevance Model
ROCC: Rocchio's formula

In **ROCC**, *you add terms to a query based on their frequencies in the top-ranked documents retrieved in response to the original query. In* **RM**, *you estimate the co-occurrence probabilities for these words vis-a-vis the words in the original query.*

Incremental Machine Learning for Faster Implementations of BoW Models

- Strictly speaking, the models we have talked about so far should **only be used for mature software** libraries that change infrequently.
- For software that changes frequently, the methods we have discussed will result in **Query Latency Times** that are too long since you would need to recompute the model after every change to the library.
- Rao, Medeiros, and Kak (WCRE'2013) demonstrates how you can **update** the VSM and Unigram models **incrementally** (and without any error) with each change to the software. And Rao, Medeiros, Kak (IWSM'2013) presents incremental versions of the LSA and LDA models (although now you have to accept small errors).

Outline

Shortcomings of the Bag-of-Words Models

- All Bag-of-Words methods **miss out on the information contained in the term-term relationships** in the queries and in the files.
- Consider a query that contains the terms **"multiplexed"** and **"socket"** in close *proximity* and in that *order*. Let's say you are searching for files that not only contain these two terms but also satisfy the **proximity and order** requirement.
- With Bag-of-Words, you **cannot** enforce such constraints.
- **This observation applies as much to simple models such as VSM and SUM as it does to more complex models like LDA.**

Modeling the Term-Term Relationships in Software

- Over the years, researchers in the Machine Learning community have devoted much energy to the investigation of methods **for the probabilistic modeling of arbitrary dependencies amongst a collection of variables.**

- The methods that have been developed are all based on graphs. **The nodes of such graphs represent the variables and the arcs the pairwise dependencies between the variables.**

- The graphs may either be directed, as in **Bayesian Belief Networks**, or undirected, as in the networks derived from **Markov Random Fields**. In both, the set of variables that any given variable directly depends on is determined by how the nodes are connected.

Bayesian Belief Networks and Markov Networks

- In a Bayesian Belief Network, **the probability distribution at a node q is conditioned on only those nodes that are at the tail ends of the arcs incident on q**. Each directed arc represents the node at the tail being causal to the node at the head.

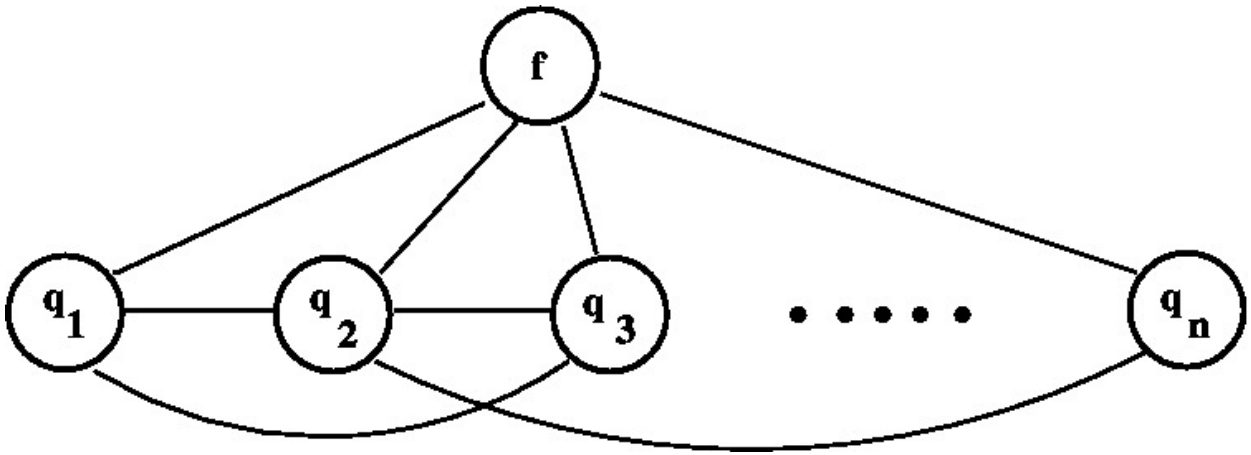
- In a Markov Network — also known as a Markov Random Field (MRF) — the probability distribution at a node q **depends only on the nodes that are immediate neighbors of q** without associating any directionality with the arcs.

- In the context of retrieval from natural language corpora, the work of Metzler and Croft (SIGIR'2005) has shown that **Markov Networks are particularly appropriate for the modeling of the term-term dependencies.**

MRF for Modeling Term-Term Relationships in a Query vis-a-vis a File

- We construct a dependency graph G that contains one node for the file being evaluated for its relevance to the query, and we use the other nodes for representing the query terms.

- Typically, we denote the node that stands for the file by f and the other nodes by the query terms $Q = \{q_1, q_2, \ldots, q_n\}$.



MRF for Modeling (contd.)

- Our goal is to assess how relevant the file f is to the the query. A concept that is important to this assessment is that of a **clique** in a graph. A clique is a set of nodes that are mutually interconnected.

- The dependency graph on the previous slide has several cliques. Just to name a few: the set of nodes $\{f, q_1, q_2, q_3\}$ forms a clique, as does the set $\{f, q_1, q_2\}$, as also does the set $\{q_1, q_2, q_3\}$, and so on.

- **The relevance of a file f to the query Q is obviously given by the joint probability $P(f, Q)$.** When this joint probability can be expressed in the product form shown on the next slide, we are guaranteed that **the graph G will act as an MRF**.

MRF Modeling (contd.)

- Let $C_1, C_2, \ldots, C_K$ denote the cliques in G. Under MRF modeling, the joint probability distribution $P(f, Q)$ is expressed as:

$$P(f, Q) \quad = \quad \frac{1}{Z} \prod_{k=1}^{K} \phi(C_k) \quad =_{rank} \quad \sum_{k=1}^{K} log(\phi(C_k))$$

  where $log(\phi(C_k))$ is called the **potential function** over the cliques and Z the normalization constant. The product form for the joint probability is referred to as the **Gibbs distribution for the graph** G.

- This formula captures the practical intuition that the joint distribution $P(f, Q)$ **depends on the frequency with which all of a given collection of mutually dependent query terms (that is, the terms in a clique) occur together in the file** f in relation to other groupings of such mutually dependent terms.

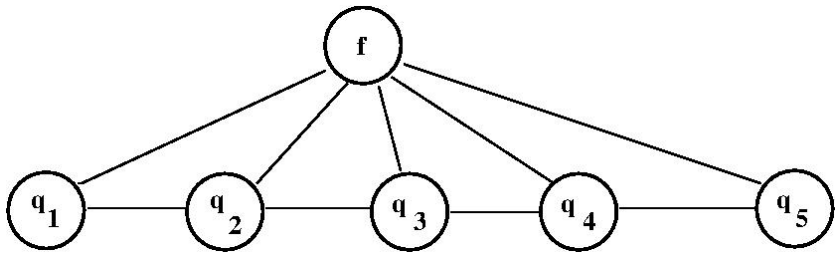
Invoking the Markov Property

- *The Markov property implies that arbitrary term-term relationships amongst the query terms can be encoded in the form of how a node depends on its immediate neighbors only.*

- A major implication of the Markov property is that even when our belief says that a specific term q depends simultaneously on, say, 10 other terms, that fact can be captured by drawing arcs from q to the nodes representing the 10 terms. These 10 nodes become the immediate neighbors of node q.

- Note that we are free to assume any connectivity pattern between the query nodes and any potentials for the cliques. Our choices **reflect our beliefs** about how the terms are related to one another.

Capturing Order and Proximity in MRF



- Assume a query with five terms and assume that our dependency graph G has the connectivity pattern as shown above. By Markov property, $P(q_2|Q, f) = P(q_2|q_1, q_3, f)$ and $P(q_3|Q, f) = P(q_3|q_2, q_4, f)$.

- Regarding Order and Proximity, let's start by noting that since the **potentials** we choose for the cliques are an expression of our beliefs, we can choose for the clique $\{f, q_1, q_2\}$ a potential that depends on the frequency of occurrence of the pairs $q_1 q_2$ in the file f. Likewise, we choose for the clique $\{f, q_2, q_3\}$ a potential that depends on the frequency of the occurrence of the pairs $q_2 q_3$ in the file; and so on.

Capturing Order and Proximity (contd.)

- Choosing the potentials in the manner described on the previous slide imposes an **order** on what sequence of terms in the file f is considered relevant to the given query. Making our potentials proportional to the frequency of occurrence of the pairs $q_1 q_2$ and $q_2 q_3$ implies that the greater the frequency with which these pairs occur in a file, the greater the relevancy of the file to the query.

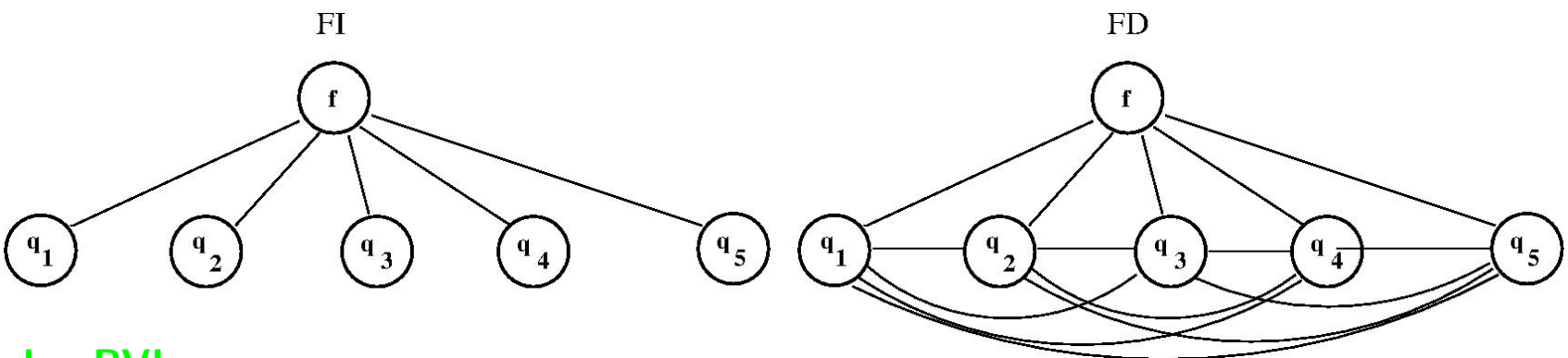
- Since the two pairs taken together imply q_2 follows q_1 and q_3 follows q_2, that captures **the ordering of the terms q_1, q_2, and q_3.**

- Here the **proximity constraint** is implied the fact that we want q_2 to immediately follow q_1 and we want q_3 to immediately follow q_2. More **distal term-term connections** will imply proximity constraints involving longer distances between the terms.

MRF Modeling: FI and FD Assumptions

- If assume that there exist no arcs at all between the query terms, that means we do not care about any dependencies between the terms. **Referred to as the full-independence (FI) assumption, this is the same as the Bag-of-Words assumption.**

- At the other end of the spectrum, we can assume that there is an arc between every pair of query terms in the graph. **That is referred to as the full-dependency (FD) assumption.**



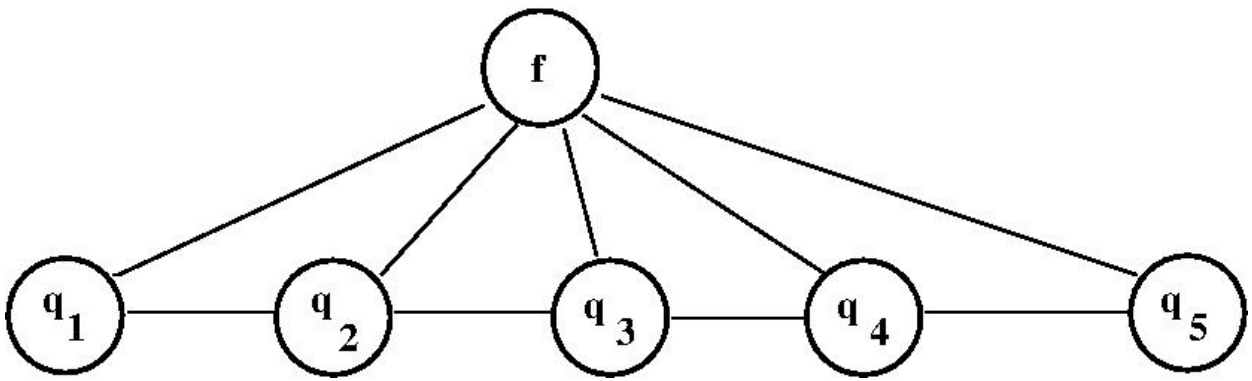
Potentials for the Cliques Under the FD Assumption

- As you can see in the FD graph on the previous slide, the FD assumption results in cliques that do not involve the file node f.
- When a clique includes f, we have the convenient interpretation for the potential that was previously stated. **How do we specify potentials for the cliques that do not include f?**
- The potential associated with a clique is really meant to be a measure of our belief regarding the mutual compatibility of all of the nodes in the clique. When a clique does NOT include the node f, the potential associated with the clique has to reflect our prior belief in the mutual compatibility of the query terms involved.

The Special Case of SD Assumption

- Of particular importance to us is the **Sequential Dependency (SD)** assumption that is captured by placing an arc between the consecutive pairs of nodes that represent the query terms.

- Under SD, in order to determine whether a file is relevant to a query, we compare the frequencies of ordered pairs of terms in a file for the pairs formed by the **consecutive** query terms.



Evaluation Datasets Used for MRF Based Modeling

Project, Description	Language	$\|B\|$	$\|RF\|$
AspectJ	Java	291	3.09
Eclipse v3.1	Java	4,035	2.76
Chrome v4.0	C/C++	358	3.82

$|B|$: Number of bugs

$|RF|$: Number of relevant files per bug

The next few slides show the results for AspectJ only. The Sisman and Kak, 2014, publication presents results on all three datasets.

Some Experimental Results in Bug Localization with MRF Modeling of the Software (Sisman and Kak, 2014)

Table: *Retrieval accuracy with the "title-only" queries. (We treat FI as baseline since it is synonymous with SUM.)*

	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.1410 (+12.89%)**	**0.1409**	**0.0832**	**0.1794**	**0.2420**	124
SD	0.1348 (+7.93%)	0.1340	0.0790	0.1675	0.2382	**125**
FI	0.1249	0.1375	0.0708	0.1498	0.2079	111

Some Experimental Results in Bug Localization with MRF Modeling of the Software (Sisman and Kak, 2014)

Table: *Retrieval accuracy for the "title+desc" queries. (We treat FI as baseline since it is synonymous with SUM.)*

	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.1945 (+14.08%)**	**0.2131**	**0.0997**	**0.2322**	0.2996	142
SD	0.1794 (+5.22%)	0.1856	0.0990	0.2203	**0.3096**	**148**
FI	0.1705	0.1856	0.0880	0.2003	0.2693	126

Some Experimental Results in Bug Localization with MRF Modeling of the Software (Sisman and Kak, 2014)

Table: *Retrieval accuracy for the "title+desc" queries with Query Conditioning (QC). (We treat FI as baseline since it is synonymous with SUM.)*

	MAP	P@1	P@5	R@5	R@10	H@10
FD	**0.2307 (+8.31%)**	**0.2715**	**0.1155**	**0.2703**	0.3400	161
SD	0.2263 (+6.24%)	0.2646	0.1148	0.2658	**0.3554**	**167**
FI	0.2130	0.2440	0.1052	0.2438	0.3215	147

Query Conditioning is carried out for only those bug reports that contain stack traces and code patches. The terms yielded by special preprocessing applied to these segments of a bug report are included in the query. The code patches are rare in bug reports (8/4035 for Eclipse). Stack traces occur more frequently (519/4035 for Eclipse).

Outline

Changeability as Measured Through Modularization Metrics

- Changeability of a software library depends very strongly on how well it is modularized.
- The currently **best-known metrics for measuring the quality of modularization of legacy procedural code** are by Sarkar, Rama, and Kak (IEEE-TSE 2007):

1. Module Interaction Index	7. Cyclic Dependency Index
2. Non-API Function Closedness Index	8. Layer Organization Index
3. API Function Usage Index	9. Module Interaction Stability Index
4. Implicit Dependency Index	10. Testability Index
5. Module Size Uniformity Index	11. Concept Domination Metric
6. Module Size Boundedness Index	12. Concept Coherency Metric

These metrics measure modularization quality from different perspectives that include inter-module coupling, intra-module architecture, similarity of purpose and commonality of goals, etc.

Changeability as Measured Through Modularization Metrics (contd.)

- And the currently **best-known metrics** for measuring the quality of modularization of **large object-oriented software** are by Sarkar, Kak, and Rama (IEEE-TSE 2008):

1. Module Interaction Index	6. State Access Violation Index
2. Non-API Method Closedness Index	7. Size Uniformity Metrics (3)
3. Base Class Fragility Index	8. Plugin Pollution Index
4. Inheritance-Based Inter-Module Coupling Index	9. API Usage Index
5. Association Induced Coupling Index	10. Common Use of Module Classes Index

- **Machine Learning can tell us how to establish decision thresholds on the metric values for discriminating between the software packages that are good with regard to modularization and the packages that are not so good.**

Usability as Measured Through API Metrics

- Usability of Modern Software is intimately tied to the quality of its APIs.
- However, the quality of APIs requires a human-centric assessment since they are meant to be used by programmers for creating new code.
- **Fortunately, we now possess a set of commonly held beliefs regarding the structure of high-quality APIs.**
- Given a set of metrics that can translate these beliefs into numerical characterizations of the APIs, the role that Machine Learning can play is to make it possible for us to discriminate between good APIs and not-so-goad APIs.

Translating Commonly Held Beliefs About What Constitutes a Good API into a Set of Metrics

- The world's **best metrics for measuring the quality of APIs** are by Rama and Kak (SPE 2013):

1. API Method Name Overload Index	5. API Method Grouping Index
2. API Parameter List Complexity Index	6. API Thread Safety Index
3. API Parameter List Consistency Index	7. API Exception Specificity Index
4. API Method Name Confusion Index	8. API Documentation Index

- These allow us to characterize the quality of any API purely on the basis of what is declared in the API document of a module.

- **Machine Learning can help us determine the acceptable ranges for the metric values that programmers are comfortable working with.**

Outline

To Summarize

- My talk reviewed how one can create compact representations of large software that open up new avenues for humans to interact with the software with regard to code search and code maintenance.
- I started by describing the simplest of these representations. These are typically based on the Bag-of-Words (BoW) assumption.
- I then mentioned how we can augment the power of BoW models with query reformulation and by using file metadata related to file modification and file defect histories.

To Summarize (contd.)

- I also mentioned that for software that is rapidly evolving, you would want to use incremental versions of the data modeling algorithms.
- Subsequently, I reviewed a modeling approach that is currently believed to be the most powerful method for designing search engines for large software. This approach is based on Markov Random Fields. This approach allows us to incorporate order and proximity constraints in code search.
- Finally, I alluded to how Machine Learning can play an important role in our ability to properly interpret the numerical characterizations of the changeability and usability attributes of large software.

Thank you all!