DHQ: Digital Humanities Quarterly
2015
Volume 9 Number 3

A Textual History of Mozilla: Using Topic Modeling to Trace Sociocultural Influences on Software Development

Michael L. Black <michael_black_at_uml_dot_edu>, University of Massachusetts Lowell

Abstract

This essay applies a digital humanities methodology to the study of digital media by framing software history as a problem of big data and textual criticism. While many scholars have already identified source code as an important site for the cultural study of software, this essay proposes that close reading is not suited to the scale and scope of modern application software. Applying text analysis tools to large bodies of source code can help address these problems as well as provide a basis for narrativizing its development and contextualization amidst a complex network of economic, political, and other sociocultural processes. Using source code produced over 15 years of development by the Mozilla foundation, this essay argues that software interfaces act as an extension of public relations narratives, often representing a developer's idealized version of their software which may not necessarily correspond with the state of the application's source code. Topic modeling Mozilla's source code allows us to explore a more nuanced narrative of development that shows how Mozilla has negotiated between the political ideals of open source software and the corporate ideology of its parent companies.

I. Introduction

Despite a growing interest in the cultural study of source code, few scholars have addressed the complex ways that programming's inscriptive practices inform the materiality of software. Among the earliest advocates for source code studies in the humanities is technology historian Michael S. Mahoney, who argues that source code could support a style of historiography that escapes from the "who did it first" questions that have been a constant focus of the history of computing and instead permits scholars access to the varying motivations, attitudes, and assumptions programmers have about the processes they model and the users that manipulate them [Mahoney 1996]. Studying software from the perspective of source code could also help us to avoid privileging a programmer's intended representation of software in the sense that we would not be wholly reliant on the interface as a point of entry, allowing us to uncover not just the complex algorithms it sublimates but also the contexts of labor it effaces [Markley 1996, 77]. As an object of study, source code offers us an alternate history, a textual narrative that can complement or counter the ones provided by corporate insiders. Unsurprisingly, source code has recently garnered the interest of scholars in literary studies who want to explore entanglements between natural language and programming by enacting a form of code review that incorporates aesthetic, cultural, and hermeneutic inquiry [Berry 2011] [Black 2012] [Marino 2006] [Sample 2013] [Wardrip-Fruin 2009]. This article is supportive of the belief that source code can be a productive site of inquiry for software studies; however, I contend that "close reading" source code as a methodology is limited in its ability to describe the scale and scope of the sociocultural processes that leave their marks on the iterative inscription of modern application software. 1

To address these limitations, I frame source code study as both a problem of textual criticism and of big data. Software is often described as more fluid than other media, but the programs we run on our desktops, laptops, tablets, phones, and other devices represent a fixed moment in software's development when a continually revised body of coded text is frozen and transformed, or "compiled," into an executable object. Programming can continue but the compiled executable is effectively separated from the source code used to produce it, representing only one edition or "version" of 2

that body of text. As defined by Jerome McGann, the traditional goal of textual criticism has been to establish a single text from multiple versions that represents an author's final or original intentions [McGann 1992, 15]. Yet more recently, textual criticism has shifted towards a practice of documenting the material transmission of a text and collating the various versions into "critical editions". These critical editions allow scholars to view the history of textual changes and interpret them through a variety of historical and theoretical frameworks. Documenting and studying changes to source code over the lifetime of a program's development would afford scholars similar opportunities in software studies.

3 Scholars reviewing source code face the same logistical problems as developers when working with large codebases: tracking and interpreting minute changes in a complex, networked body of textual segments comprised of hundreds of thousands, if not millions, of lines of code and composed by dozens of authors. Many software development models in use today acknowledge these complexities and assume that developers will have a necessarily limited understanding of the software they work on. Developers have adapted to these conditions with the help of semi-automated revision control software like git and Apache Subversion that not only track changes across multiple simultaneously constructed "branches" of inscription but also assist with merging them into distinct "versions". Scholars in the humanities must also turn towards semi-automated methods to engage with "big" sources, and in this article I demonstrate that topic modeling using the latent dirichlet allocation (LDA) algorithm can be applied to source code in order to delineate the structural and textual development of software. Yet when applying LDA to source code, or any other form of cultural data, it is important to do so in ways that reflect the critical and interpretive assumptions we would bring to texts using more traditional humanities methodologies. This study therefore uses a workflow around LDA that, as described below, demonstrates a tacit understanding of how software's textuality informs its materiality.

4 This study takes as its object software produced by the Mozilla Foundation. Currently, Mozilla boasts over half a billion users of its software world-wide, and Wikimedia estimates that 14.1% of all web traffic occurs through Mozilla software, making the company the second most popular developer of non-mobile browsers.[1] Unlike other web browser developers, Mozilla has a comparatively long history of open source releases, resulting in a textual ancestry that can be traced back nearly 20 years through publicly accessible repositories from the latest version of the Firefox browser to the open source release of Netscape Communicator. Importantly, Netscape's decision to release its software as open source and form Mozilla as an office to manage community-driven software development also makes Mozilla's software an important site to test the political promises of the free and open source software movements. While Mozilla has worked to cultivate a public image of protecting user freedom by maintaining transparency in its development process, this study finds that Mozilla's software leverages modularity in order to conceal elements of its software from the interface that do not align with the development narratives of its public relations campaigns. While modularity is often discussed in programming as a technique for designing components independently, here it more resembles the social strategy described by Tara McPherson of partitioning off and hiding from view the undesirable [McPherson 2012]. The textual history of Mozilla's software thus demonstrates concerns voiced by free and open source software advocates [Stallman 2002] [Torvalds and Diamond 2001, 149], and echoed in broader public contexts by political theorists and activists [Fung et al. 2007] [Lessig 2009], that making the development process visible alone is not enough to protect the freedoms of users.

5 Following a brief review of scholarship in new media studies that highlights the need for cultural histories of source code, this essay explains how LDA topic modeling can help address the limitations of close reading if it is part of a theoretically informed methodology. Because the LDA algorithm relies on Bayesian probability to identify word distributions, it tends to favor static portions of text when applied to multiple versions of the same documents. Addressing this technical problem also acknowledges the complex bibliographic history that underlies the materiality of software: a constantly shifting body of source code that reflects assumptions about and reactions to the sociocultural context of an application's development. A textual history approach to source code allows us to identify moments in software history where its production deviates from the narratives its developers tell, allowing us to step outside of the rhetoric of objectivity and instrumentality that envelops technoscientific practice. Specifically in this study, I show that textual histories of software can provide a sense of context that allows us to distinguish between design decisions that are made due to technological dependence on past software and ones made due to other factors that technoscientific rhetoric tries to sublimate such as corporate influence, social pressure, or political strategies of self-representation.

II. Background

a) Using Source Code to Highlight the Cultural Work of Programming

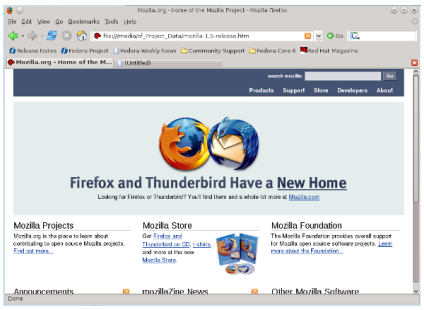
As an interdisciplinary field, software studies encompasses many different methodologies; however, most research in this area focuses on describing how users experience software or interact with each other through it. The popularity of hypertext studies among literary scholars during the 1980s and early 90s initially framed software as an interaction between users and textual processes [Aarseth 1997] [Bolter 1991] [McGann 2001] [Murray 1997]. In this period, scholars often drew on familiar literary terms related to narratology or semiotics to describe how users could manipulate text in software systems. More recent studies moved away from a focus on software as textual, describing use as an engagement with complex multimodal rendering processes and reflecting the visual shift in computing that accompanied the widespread adoption of graphic user interfaces in the 1990s [Galloway 2004] [Hayles 2008] [Manovich 2001]. In particular, Bolter and Grusin's idea that software "remediates" other media within itself helped to open software studies to a wider array of cultural and interpretive heuristics [Bolter and Grusin 2000]. Alongside close studies of software, research on use practices and cultures of users have also broadened software studies to include ethnographies of user communities [Boellstorff 2008] [Nardi 2010] as well as projects examining more specifically how race [Nakamura 2009], gender [Jansz et al. 2010], and sexuality [Gray 2009] are understood through software and performed by its users. While this focus on how users interact with and define themselves based on what they see on-screen has done an excellent job of articulating how software is intervening in and altering the course of long-standing sociocultural practices, a user-centric approach affords a necessarily incomplete view of software's materiality. 6

If one goal of software studies is to understand the assumptions and motivations that shape programing, then we need a methodology that can expose and map the inscriptive practices that comprise the earliest stages of software's production. As with other fields of technoscience, the sociocultural assumptions of programmers are often sublimated by an instrumental logic [Latour 1987]; however, placing source code within a cultural context allows us to see how programmers respond to particular cultural moments in ways that studying user experience cannot. Following the theorization of user-friendly design in the 1980s, a primary goal of software design has been to condense complex algorithmic operations into cognitively simpler visual metaphors [Norman 1988] [Winograd and Flores 1986]. While this strategy has removed barriers of access imposed by technological complexity, software now has a "cryptographic" nature that not only obscures its dependence on hardware but also hides the complex semiotic operations that occur both in response to user input and independently of user input [Kittler 1997]. Users see a proposed action, indicate that they'd like to execute it, and then see only the results of that action. They are often not allowed to observe the processes that select and execute sets of incremental operations on their behalf nor those performed automatically without notifying or asking permission of the user [Bolter and Gromala 2003]. Modern software, in short, functions around a "profound ignorance" in the sense that users can perform complicated chains of precise commands without anything more than a very abstract understanding of their actions [Chun 2011]. Whereas in the earliest years of personal computing users had to act as programmers to design and manage informational processes, there is now a firm divide between the two roles, with programmers making assumptions about how users want to structure and access their data and with users having little choice but to accept these models of data access. If user-friendly design conceals important structural assumptions about how the software we use shapes our digital agency, source code offers an opportunity to expose them, provided we can interpret it in ways that tease out the complex relationships it establishes between users, programmers, and data. 7

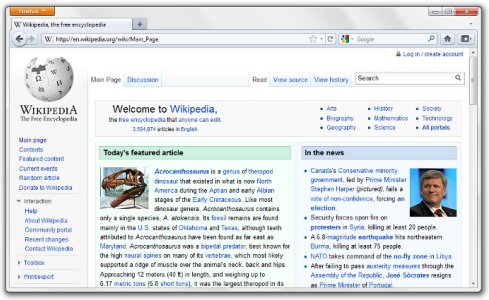
The user interface is not a reliable site to study software change. While software interfaces do often change over time, they serve as a form of self-representation that allows developers to choose deliberately which components to hide from or show to users. Figure 1 illustrates the relatively static nature of Mozilla's web browser interface which remained largely unchanged for 13 years following Netscape's adoption of an open source model in 1998. While Mozilla has adjusted elements of the browser's visual aesthetic — changing the look of the buttons, the thickness of dividing lines, or the obtrusiveness of its side panel — the functional representation of the browser's user interface has remained largely static since the introduction of tabbed browsing in version 0.9.5 in 2001. A significant feature addition for its time, the change in user experience introduced by tabs was minor compared to the minimalist re-envisioning of the interface 8

presented by version 4.0 of Firefox in 2011. In short, because the relationship between interface and internals is largely arbitrary, Mozilla's software outwardly reflects only those internal changes that its developers choose to reveal. Users may notice the browser's rendering capabilities have increased continually to support the increasingly complex web of HTML, CSS, and Javascript code that comprise modern Internet sites, but these changes are also the result of web developers updating and changing their websites and services. Studying Mozilla's software from the perspective of its source code could expose changes invisible to users that make possible advances in web language standards, opening them to sociocultural analysis in ways that studying the terse notes developers provide in public changelogs or even those left in their own internal documentation cannot. Already, scholars like those associated with Critical Code Studies have argued that closely reading source code can expose logics not visible in the interface [Marino 2006]; however, any attempt to critically read source code faces limitations of scale and code when applied to modern application software that is comprised of dozens of modules and millions of lines of code.

Mozilla Application Suite 0.95 (2001) adds tabbed browsing.



Firefox 1.0 (2004) adds search engine integration.



Firefox 4.0 (2011) combines navigation buttons, removes the bottom status bar, and replaces the drop down toolbar with a compressed "Firefox" button.

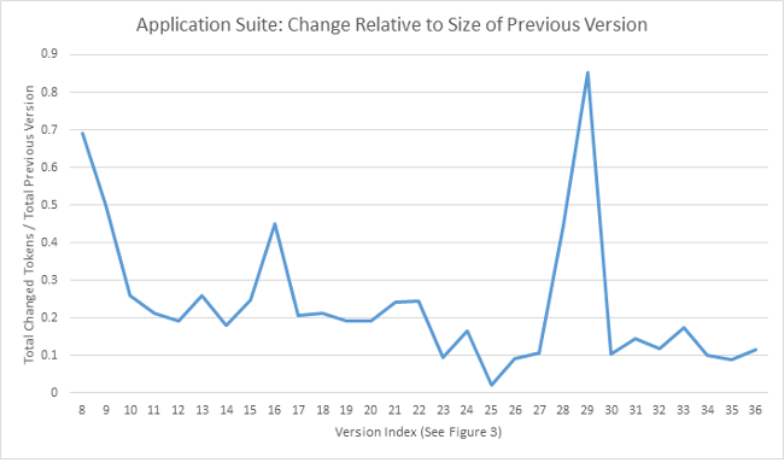
This third image was retrieved from http://en.wikipedia.org/wiki/History_of_Firefox#mediaviewer/File:Firefox_4.png and licensed for use under the Creative Commons Attribution, Share Alike license.

Figure 1. With the exception of adding tabbed browsing in 2001, Mozilla's software maintained a relatively unchanging look and feel until 2011, when Firefox's interface was "streamlined".

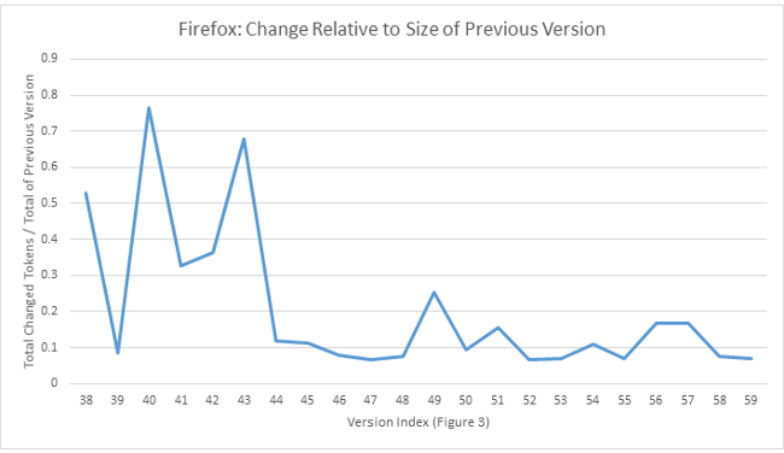
b) Addressing Problems of Scale with LDA Topic Modeling

Before considering how text analysis can help us trace software history, it's important to establish how iterative inscription informs the material production of software. The Mozilla corpus illustrates this process well, see Figure 2. After an initial flurry of large updates that establish a basic core of components, programmers can turn to smaller, incremental updates. Rather than releasing a final version that is left unchanged, or lightly "patched" once or twice to remove bugs, the source code to modern application software is often continually revised and re-compiled, with each successive version representing a fixed moment in that application's textual history. The expectation that personal

computers are always online whenever they are powered on allows developers to produce a steady stream of “rolling releases” that update component periodically rather than bundling them together into monolithic new versions released every year or two years. As Cusumano and Yoffie note, Netscape was an early adopter of this strategy, leveraging the flexibility of its smaller design team into enacting an “agile” strategy that allowed it to compete against Microsoft during the browser wars of the 1990s [Cusumano and Yoffie 1998]. Today, rolling release development is now the norm among open source projects and increasingly common in commercial development. Mozilla has even gone so far as to switch to a “once every six weeks” release schedule for Firefox starting with version 5 in 2011. In the 22 months between Mozilla’s switch to rolling release and the end of the data collection period for this study, Mozilla released 15 versions of its browser, compared to 4 versions in the 22 months preceding the switch. Figure 2 shows how rolling releases encourage frequent, but small, revisions to source code. While there are periods of significant change when the sum of tokens added and removed between versions of Mozilla’s software is greater than 50% of the number of unchanged tokens, the majority of the time that amount is less than 25%.



Average change between versions of Mozilla Application Suite: 23.86%



Average change between versions of Firefox: 20.41%

Figure 2. Graphs for Application Suite and Firefox of the total change in code size over time, calculated as the sum of the tokens changed (ABS(δ-remove) + δ-add) relative to the total number of tokens in the previous version.

Computer scientists have already shown that studying large codebases using automated or semi-automated algorithmic methods can highlight complex patterns not easily traceable in software through close review of source code even with assistance from version management systems. In a largely practical context, datamining software repositories can assist in enriching human-written documentation [Witte et al. 2008], predicting future allocation of development resources [Lukins et al. 2008] [Ying et al. 2004], or predicting the likeliness that bugs will be identified in particular components [Williams and Hollingsworth 2005]. Recently, studies that textmine source code have also shown that these

methods can address questions that fall outside of the normal purview of science and engineering, such as identifying patterns of social relationships by tracing source code contributions from developers on the same project [De Souza et al. 2005]. Just as humanists and social scientists are already adapting machine learning and informational retrieval techniques to study traditional sources within their fields, so too can we draw on these techniques to advance sociocultural studies of software [Clement et al. 2013] [Jockers 2013] [Underwood and Sellers 2012].

11 In particular, topic modeling can perform the same keyword identification and summarization functions on documents composed in source code that it does when applied to those in natural language. Although there are significant stylistic differences between the two, topic modeling uses a “bag of words” approach that effectively collapses them by ignoring word sequence. Source code documents thus can be preprocessed similarly to natural language documents by treating frequently used functions or flow control operations like “print”, “for”, or “while” as stopwords; breaking compound variable, object, and function names created through the practice of camel case into their component words; and segmenting the corpus along basic units of composition such as packages, classes, or methods instead of chapters, pages, or paragraphs [Kuhn et al. 2007]. As with natural language corpora, the results of a topic model performed on source code can be used to categorize segments, make comparisons between them, and interpret chronological or networked relationships among them [Gethers and Poshyvanyk 2010] [Linstead et al. 2007].

12 Yet as Thomas, et al. observes, LDA topic modeling is not well suited to studying a versioned corpus due to the excessive repetition of text [Thomas et al. 2011]. When applying LDA to several versions of source code of the same software, their study found that those passages which remained relatively unchanged effectively assimilated patterns of change emerging in later versions. As a result, distinct patterns of additions and removals to the code base that would be recognizable to human readers were often partially or totally obscured by static passages. This problem, they observed, is due largely to LDA’s reliance on Bayesian probability. LDA assumes that all documents are distinct, and repetition across versions leads LDA to recognize static passages as tokens with a higher probability of co-occurring in documents, making those patterns of word distributions that emerge earlier in the corpus more prominent than those which appear later. To account for this repetition, Thomas, et al. recommends using the Unix utility “diff,” or a similar tool, to isolate the added and removed passages of texts in each successive version [Thomas et al. 2011]. This project replicates their DiffLDA method, described in more detail below.

13 In addition to its practical benefits, DiffLDA also implicitly foregrounds the textual practices that inform software’s material production. Rather than assuming that a single version of software is authentic or final, DiffLDA acknowledges not only that software is produced through the iterative inscription of source code but also that no version should be privileged as more authentic than others. As McGann notes, attempts to locate or reconstruct an edition of a text that represented an author’s “final intentions” are problematic: “many works exist of which it can be said that their authors demonstrated a number of different wishes and intentions about what text they wanted to be presented to the public, and these differences reflect accommodations to change circumstances, and sometimes to changed publics” [McGann 1992, 32]. Thomas, et al (2011) implicitly acknowledges that LDA’s algorithms privilege unchanged code from earlier versions of an application as more authoritative than any code added later, even if new contributions also remain static over time. While early source code contributions are important to understanding the general role envisioned for a piece of software, they cannot help us understand how the changing circumstances of its development influenced its composition. As with literary texts, the initial vision for a software project can and often is revised according to the software’s changing role within digital culture. In the case of Mozilla, its vision was and continues to be influenced by changing web standards negotiated with the W3C, shifting corporate relationships, a desire to protect the rights of users by implementing an open source design philosophy, and competition from other browsers, both closed and open source.

III. Methods

14 Data collection for this project took place in May 2013. I retrieved all source code related to “major versions” and “significant revisions” of Netscape Communicator, Mozilla Application Suite, and Mozilla Firefox available on Mozilla’s public repository. With the exception of unnumbered and “milestone” early releases, Mozilla follows the convention of using a sequence of three numbers to identify major releases, significant revisions, and minor revisions, respectively

(e.g., 4.1.5). I also retrieved early unnumbered releases and milestones — represented by Mozilla simply as M followed by a number — because Mozilla's press releases described each as a significant step towards the initial public release of its software. Finally, in the interest of maintaining a regular timeline, I did not begin retrieving versions of Firefox until after the final major release of Application Suite when Mozilla's development resources shifted fully to the newer browser. The resulting corpus represents sixty versions of Mozilla software produced over the course of 15 years. A complete list of versions comprising this corpus can be found in Table 1. Readers should also note that all graphs in this essay show change over time based on index numbers rather than a regular temporal scale and should therefore refer to Table 1 to match index number to version numbers and release dates. This table also contains data about the changing corpus size, specifically: the change in total size on disk of all files, the change in size on disk of the C++ files used in topic modeling, as well as the total number of words added or removed between each version.

Version Index	Version Number	Release Date	Size (Bytes)	Net Change (Bytes)	C++ Change (Bytes)	Total δ Tokens
Netscape Communicator (Open Source)						
00	None	03/31/98	40203077	N/A	N/A	233285
01	None	04/08/98	59131622	18928545	9694228	692924
02	None	04/29/98	65089258	5957636	548493	714233
03	None	06/03/98	70166069	5076811	1223271	753252
04	None	07/28/98	84099725	13933656	8086869	1132744
05	None	09/04/98	90469752	6370027	330190	1144931
06	None	10/08/98	83588207	-6881545	-1708705	1048406
Mozilla Application Suite						
07	(M1?)	12/11/98	44647824	N/A	N/A	674816
08	(M2?)	01/28/99	55504133	10856309	4673001	865565
09	M3	03/19/99	65940062	10435929	4740616	959360
10	M4	04/15/99	70438485	4498423	2334716	1031601
11	M5	05/05/99	73321774	2883289	1766415	1113046
12	M6	05/29/99	85172946	11851172	2240285	1190649
13	M7	06/22/99	86226782	1053836	1453313	1269994
14	M8	07/16/99	90528214	4301432	2082572	1347142
15	M9	08/26/99	94343632	3815418	1752674	1342563
16	M10	10/08/99	99242381	4898749	413658	1393442
17	M11	11/16/99	105517092	6274711	1403584	1477091
18	M12	12/21/99	110274354	4757262	1731966	1464024
19	M13	01/26/00	108091563	-2182791	-216838	1540253
20	M14	03/01/00	112325279	4233716	1751420	1630439
21	M15	04/18/00	117337663	5012384	1845194	1783563
22	M16	06/13/00	123233166	5895503	3866597	1839983
23	M17	08/20/00	141779386	18546220	1289320	1882548
24	M18	10/12/00	145153396	3374010	903169	1896079
25	0.6	12/06/00	135022311	-10131085	271163	1917088
26	0.7	01/09/01	152227947	17205636	507036	1936538
27	0.8	02/14/01	152513233	285286	293603	2090149

28	0.9	05/07/01	170953881	18440648	3763559	2274789
29	1.0	06/05/02	224325235	53371354	6189989	2332033
30	1.1	08/26/02	228137982	3812747	1363139	2402289
31	1.2	11/26/02	228896289	758307	1737235	2433636
32	1.3	03/13/03	229003326	107037	725868	2436192
33	1.4	06/30/03	230873828	1870502	157801	2427898
34	1.5	10/15/03	184967631	-45906197	-68683	2447614
35	1.6	01/15/04	188991165	4023534	290254	2503244
36	1.7	06/17/04	206097164	17105999	1299795	2545786
Mozilla Firefox						
37	1.0	11/09/04	197018982	N/A	N/A	2809989
38	1.5	11/30/05	208257684	11238702	5285960	2155079
39	2.0	10/24/06	218235493	9977809	2380870	2237129
40	3.0	06/17/08	227187445	8951952	-15144818	2241148
41	3.5	06/30/09	278296889	51109444	2564959	2531559
42	3.6	01/21/10	280683836	2386947	690404	2504039
43	4.0	03/22/11	346351093	65667257	5805397	2483528
44	5.0	06/21/11	352386092	6034999	-729812	2469320
45	6.0	08/16/11	347554884	-4831208	-516552	2456907
46	7.0	09/27/11	349362417	1807533	-769400	2511091
47	8.0	11/08/11	354846376	5483959	-253822	2686495
48	9.0	12/20/11	364515482	9669106	1535342	2694930
49	10.0	01/31/12	372937453	8421971	4338895	2714671
50	11.0	03/13/12	385911428	12973975	-55856	2740373
51	12.0	04/24/12	388876784	2965356	472535	2790036
52	13.0	06/05/12	393197798	4321014	576292	2864587
53	14.0.1	07/17/12	407165259	13967461	1061975	2891327
54	15.0	08/24/12	395983732	-11181527	-2924526	2976568
55	16.0	10/09/12	429430729	33446997	552219	3167788
56	17.0	11/20/12	435263141	5832412	1455213	3182711
57	18.0	01/08/13	465517793	30254652	4674165	3216161
58	19.0	02/19/13	471549259	6031466	292543	233285
59	20.0	04/02/13	476262707	4713448	774360	692924

Table 1. General information about the Mozilla Corpus. Note that the x-axes of other figures correspond to the Version Index Numbers provided in the leftmost column of this table.

To implement DiffLDA on the Mozilla corpus, I used Python to capture the output of the diff command on the extracted C++ source code for each pair of successive versions. Following the recommendations of Thomas, et al, my scripts parsed the output into a pair of δ-add and a δ-remove documents for each cpp implementation file, thereby removing all repeated text and reducing each version to a record of its changes [Thomas et al. 2011]. In the event that entire cpp files were added or removed between versions, the entire file would be parsed as a δ-add or δ-remove document, respectively. At points of transition between Communicator, Application Suite, and Firefox, all cpp files from the older browsers were parsed as δ-remove and all cpp files from the new browsers were parsed as δ-add documents. Once all cpp files were processed through diff, I tokenized the resulting documents following the recommendations of Kuhn, et al

by splitting all camel case and underscore compound method and variables names, removing all non-alphabetic types, and filtering out both standard English stopwords as well as variable type declarations, flow control commands, object type declarations, and commonly used commands [Kuhn et al. 2007].

Thus prepared, I performed LDA on the corpus using MALLET's ParallelLDA implementation in order to transform every file in each version into a pair of δ-add and δ-remove feature vectors and sum them into virtual representations of each version. For example, if there are 10 files in version Y, diff processing makes note of the changes since version X in 20 files: pairs of documents that list the additions or removals for each of the 10 files. The records of changes for all documents are then pooled together into a corpus for LDA. Following LDA, the record of changes in version Y is now represented by 20 feature vectors with each column value in the vectors representing the total number of tokens assigned to each topic in the records of additions or removals. The resulting topic assignment vectors for all documents are then transformed into virtual representations of each version by cumulatively summing each column, producing a vector for each version wherein all columns are the sum of all previous δ-add values less the sum of all previous δ-remove values. Column 1 in the virtual vector for version 4, for example, is the sum of all additions minus the sum of all removals in column 1 of versions 1, 2, and 3. These summed vectors thus represent the net tokens assigned to each topic in each version. After several initial trials, I settled on a topic model size of 60 because it produced topic key terms lists that could both be traced to specific descriptions of components in Mozilla's own documentation with minimal duplication of topics across the three software applications included in the corpus. As shown in Figure 3, I noticed that topic modeling would re-create distinct trends regardless of the number of topics chosen. Using a larger number of topics generally produced "new" topics that reflected the normal narrative of object-oriented programming, described below, rather than highlighting further trends of interest to this study.

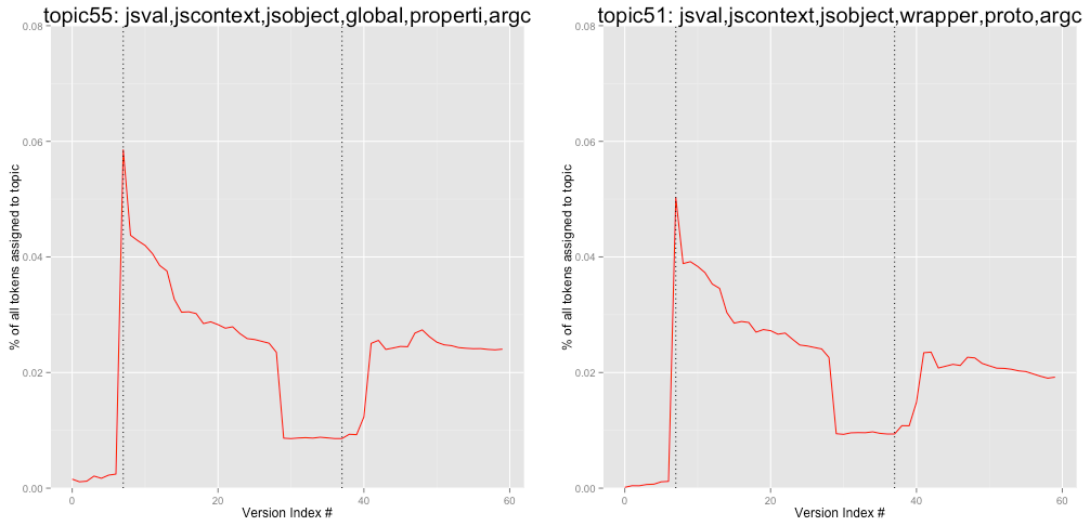


Figure 3. Distinct trends are visible regardless of the number of topics chosen. The left image is from a model that uses 60 topics and the right from one that uses 100 topics.

While my implementation of DiffLDA is largely faithful to the process described by Thomas, et al, I did make two changes to account for irregularities I noticed in the initial results [Thomas et al. 2011]. Like Thomas, et al, I found that the probabilistic calculations underlying the LDA process could be overly sensitive to change, resulting in a version having less than zero tokens assigned to a topic after cumulatively summing the δ-add or δ-remove documents [Thomas et al. 2011]. I noticed that these negative sums tend to occur during periods of large textual change, particularly when Mozilla decided to change or revise its license agreements. In open source projects, a licensing statement typically occupies the first several dozen lines of a source code document and will be identical across all documents it appears within. Changes to a project's licensing therefore results in broad patterns of change affecting all documents, often producing a sudden drop in token counts that would lead to a net total of several hundred or even over a thousand tokens below zero due to the "fuzziness" of LDA's Bayesian probability assignments. To avoid this

problem, I removed all licensing statements during preprocessing. Additionally, because the Mozilla corpus includes three distinct web browsers, I restarted DiffLDA parsing at the transition from the open sourced Communicator to Application Suite and from Application Suite to Firefox, as described above, by setting the values of all topics to zero and treating all documents from the newer software as δ-add documents. The points of transition between these browsers is noted in all topic graphs by vertical dotted lines. Together, I found that these two modifications eliminated almost entirely the need for norming that Thomas, et al propose to address problems of sensitivity [Thomas et al. 2011].

IV. Results & Discussion

For most topics, graphing the net tokens as a percentage of total tokens over time produces either a gradual increase or decrease across at least two of the three applications in this collection. This trend resembles the typical object-oriented development narrative, showing components developed steadily at regular intervals according to a design plan, with many reaching a point of standardization. Object-oriented programming languages like C++ encourage a highly modular development model, not just in the interest of dividing complex software into discrete components that can be distributed among programmer workloads but also so that the basic components that more complex components will depend upon can be written once and then left alone, or modified only slightly to account for any unanticipated needs. Relatively unchanged components would therefore show a steady decline over time relative to the rest of a continually expanding corpus. Furthermore, periods of stability and moments where components appear to be revisited can be traced to Mozilla's internal "Bugzilla" reporting system. 18

On the other hand, there are also a significant number of topics which do not reflect this ideal narrative and instead show sudden spikes, drops, or other irregularities. These topics represent development decisions made in response to the demands of their users, changing Internet standards, and other extra-technical influences on the development process. Here, in other words, topic modeling source code exposes software's relationship to its sociocultural context, removing it from the technoscientific narratives that sublimate process in favor of presenting only finished products and proven theories [Latour 1987]. The remainder of this section interprets two of the trends that deviate from the expected shape of the typical object-oriented development narrative, contextualizing the traces of material production visible in the textual history of Mozilla's source code by reading them alongside public relations material, blog posts written by Netscape and Mozilla employees, technical white papers, and corporate histories.[2] 19

a) E-mail

Mozilla's negotiation of corporate and open source principles is visible in the topics representing e-mail components, a function bundled into Communicator and Application Suite but not included in Firefox. Figure 4 shows the graph for topic 23, which is associated with the key terms "header", "line", "mime", "attach", "part", and "field". These terms refer both to the standardized code that e-mail servers wrap messages in to help route them — the message "header" and Multi-purpose Internet Mail Extensions ("mime") standard — as well as Mozilla's internal abstract representation of the information included in that code as "fields". Aside from the erratic Communicator period, the graph for topic 23's membership shows a steady decline over time, suggesting that the e-mail components reached a state of completion and were left alone. Given that e-mail protocols have not changed significantly in the last decade, this is not surprising. Shortly after the transition to Firefox, however, topic membership drops off almost entirely, indicating that e-mail related components were removed from the source code. Topics comprised of other e-mail related keywords like "msg", "mail", "folder", "server", "download", "account", and "pop" like topic 4 and topic 54 exhibit similar trends. The topics related to e-mail highlight a trend visible across many topics that the Application Suite is largely a recreation of Netscape's original design. Given Mozilla's announcements of a fresh start surrounding its formal separation from Netscape and the first glimpses of Firefox, both in 2003, it is surprising that this recreation and continuation of Netscape's design is visible in the source code as late as 2008. The comparatively seamless transition visible in these graphs between the Suite and Firefox periods shows that despite a rhetoric of renewal and fresh approaches, there was considerable lag between Mozilla's vision of reinvention and the implementation of its new designs. 20

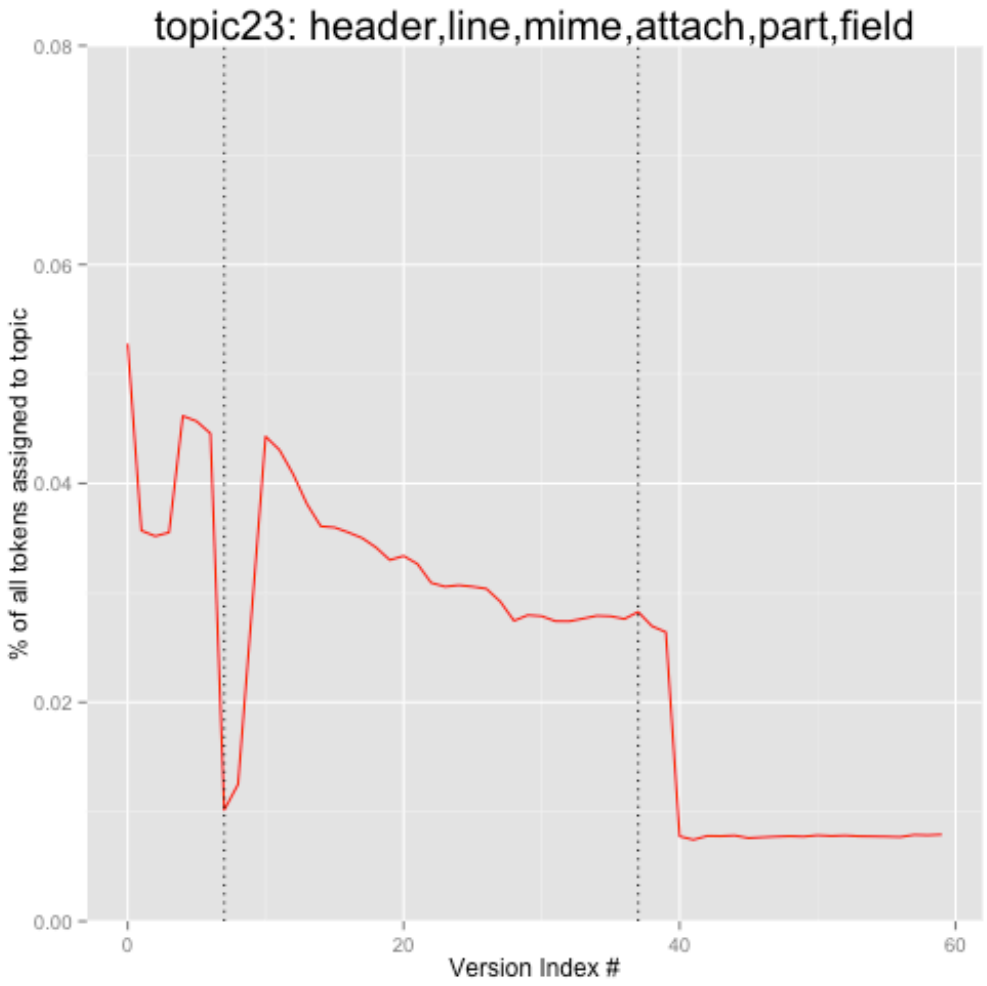


Figure 4. This topic is representative of “e-mail” related components. It exhibits a similar trend to topics 4 and 54 (not pictured).

Mozilla began as an office within Netscape, formed in 1998 just before the company’s purchase by America Online. Mozilla’s mission was to transition development of Netscape’s browser technology from a small corporate team into an open source, community driven project. Netscape’s browsers from this point forward more or less were re-branded versions of Mozilla’s Application Suite with features added or removed to appeal to Netscape’s corporate clientele. Despite having more independent control over development than any of Netscape’s other departments, Mozilla appears to have continued its parent company’s former strategy of positioning its software as an all-encompassing gateway to Internet activity. Interviews with Netscape employees conducted during the “browser wars” period prior to the formation of Mozilla describe a dual-voiced approach to marketing [Cusumano and Yoffie 1998]. On the one hand, Netscape represented its software to users as solutions to common Internet tasks, a one-stop hub for browsing, e-mail, chat, and webpage design. Yet Netscape also pitched its software to corporate clients as “platform independent” and a hub for “network centric” application development [Andreesen 1996]. Netscape gave its browser away for free to users, but it charged corporate clients for software development kits. Implicitly, Netscape’s corporate partners were thus buying access to the browser’s users. Even in Mozilla’s own later press releases, this dual-representation of its software remained: one that highlights the user experience of its browser and e-mail functions and another aimed at corporate audiences that described the software as a “cross-platform toolkit” [Mozilla 2002]. It was not until 2003, when AOL moved to dissolve Netscape and the majority of its development teams left to form the Mozilla Foundation, that Mozilla abandoned this rhetoric. Formally announcing work on a new browser later that year — initially named “Phoenix” and later “Firefox” — Mozilla claimed that it would split its software suite into separate applications, focusing its efforts on enhancing the user experience of each component. This division of its software and team was presented as a symbol of the foundation’s rebirth and a renewed commitment to the needs of its users over those of corporate partners.

While many elements of this insider history comprised of employee interviews and company press releases are also visible in the topic graphs, a textual approach allows for a more nuanced narrative. Although there are a handful of components that demonstrate some sort of continuity across the two applications, most resemble the transition in Figure 4, showing either a sharp drop in the first version of Application Suite and/or exhibiting a much different trend in subsequent versions (e.g., a sudden rise in topic membership during the earliest versions of Application Suite following a steady drop in Communicator). Generally, the transition between Communicator and Application Suite is not smooth across all topics, suggesting that even though Netscape's corporate strategy shaped design decisions behind Mozilla's software the company's earlier source code introduced few if any technological constraints shaping future browsers. In other words, Mozilla's decision to continue the dual-voiced strategy was not the result of having to work within the structural limitations imposed by re-using components from the older Communicator browser. Rather, this decision was made at least in part for extra-technical reasons such as AOL exerting control over Netscape/Mozilla, a continued belief that Netscape/Mozilla could supplant Microsoft's control over personal computing by making users dependent on its software for all Internet related tasks, or a desire by Netscape/AOL executives to maintain the corporate partnerships that Netscape had established before the buyout.

22

Importantly, the lack of a technical obligation for continuing this strategy points to concerns raised by free software advocates [Stallman 2002] and academics [Golumbia 2009] that "open source" is often a term used to impose corporate policy over volunteer communities. While Netscape's original open source announcement of Mozilla and its new open source policy explains that it will give its community of users complete access to its software, "ignite[ing] the creative energies of the entire Net community and fuel unprecedented levels of innovation in the browser market," there is also considerable reference to "new services" that will "reinforce Netscape's strategy of leveraging market penetration" [Netscape 1998]. In short, the recreation of most components visible in the topic graphs and the resumption of its dual-voiced strategy following the move to an open source format suggests that Mozilla, at least at its outset, was not simply a gesture of good will on Netscape's part to its users. Perhaps with the move to open source Netscape felt that making the source code available to all, and accepting community contributions, would paradoxically allow them more control over their software, preventing or slowing any attempts by AOL to impose its own designs. Regardless of the exact reasons, the abruptness of the transition to Application Suite shows that little, if any, of the Communicator code was retained in later browsers, making Netscape's initial foray into open source more important symbolically and/or politically than technologically with respect to the rest of the corpus.

23

By comparison, the transition from Application Suite to Firefox is smooth across most topic graphs. Mozilla explains this trend in white papers as the result of wanting to maintain Application Suite during Firefox's initial development so as not to disrupt users who depended on their software. However, the duration of technological influence that Application Suite exerts over Firefox visible in the graphs is unexpected given the efforts that Mozilla later made to distance itself publically from Netscape's software. Like the open source announcement, Firefox was initially presented in 2003 as a renewal of Mozilla's image, coinciding with Mozilla's formal and legal split from Netscape into an independent, non-profit foundation [Mozilla 2003]. Unlike the transition from Communicator, however, the source code to Application Suite was not abandoned by Firefox during or immediately after this transition. Trends across all of the topic models indicate that Firefox shares a substantial amount of source code with its immediate predecessor and thus has a considerable dependence on the previous application's technology. Components related to e-mail, for example, are still present in Firefox source code packages even though those components are not accessible in the compiled application. The revised development roadmap that Mozilla released in 2003 describes Firefox as an effort to further modularize Application Suite, disentangling the individual components while still maintaining a shared core that would allow them to communicate with one another through intermediating "extensions". Mail components remain in the web browser's code packages as late as 2008 and are finally removed following a new initiative announced by Mitchell Baker, then CEO of Mozilla's for-profit arm, to increase funding and administrative support for e-mail client development [Baker 2007]. Mozilla, in short, proposed modularizing its technology so that Application Suite could be divided into independent applications; yet before this actually occurred, it simply hid those components from users that were not part of the user experience they intended. While Mozilla intended Firefox to represent another break from its past, and altered the browser's interface to give the appearance of such a break, significant technological influence of past technology remained in place for at least 5 years.

24

b) Windows

Considering that modularity is a core principle of object-oriented programming, it is more accurate to say that all components are partitioned off by default and that the interface includes only those components that developers choose to represent their software with to users. The gap between interface and algorithm that allowed Mozilla's developers to hide components ahead of stripping them out therefore also permits developers to alter large portions of an application, potentially even restructuring it entirely, while maintaining a seamless, largely unchanging external appearance to users. Although all topics show to varying degrees the amount of change modern software undergoes over time through iterative inscription, its shifting materiality is most visible in those topics related to drawing and managing the user interface. While the e-mail topics point to a specific publicity strategy, the restructured user interface components suggest a much more pervasive corporate influence from Mozilla's earliest days that continues into Firefox's development.

25

As Figure 1 illustrated above, Mozilla's software maintained a similar user interface for over a decade. Presumably, maintaining a uniform look and feel across all major operating system platforms represented one aspect of the Netscape strategy to produce a universal Internet toolkit that would behave identically all major operating system platforms; however, as Figure 5 shows, there appears to be considerably more programming labor devoted to maintaining the interface during the Communicator period than in either the Suite or Firefox periods. Topic 18 is comprised of the key terms "window", "win", "icon", and "dialog", referring to components related to interactive browser components like windows, icons, and buttons, as well as the "dialog" boxes that contain menus, file browsers, or more complex operations requiring a combination of inputs from users. Topic 18 alone sustains approximately 10% of total token membership over five versions of Communicator, rivaled only by the mail related components described above over two versions during that same period. At its peak topic 18 has a membership of over 100,000 tokens, a number not exceeded in any other topic for nearly 10 years. Those few Firefox components which do eventually exceed the token membership of topic 18 represent abstract graphics drawing tools. Topic 48, for example, is comprised of abstract visual keywords like "rectangle", "frame", "style", and "region" rather than specific user interface elements like "window" or "icon". Given that web languages and HTML have in recent years supported more advanced graphics rendering techniques, this latter trend is not surprising; however, because Mozilla maintained a similar look and feel across all of its software until the release of Firefox 4.0 in 2011, topic 18 suggests Firefox has a substantial structural difference from Communicator: the algorithms defining components of the user interface have been stripped away, leaving only the more primitive components used to draw those objects.

26

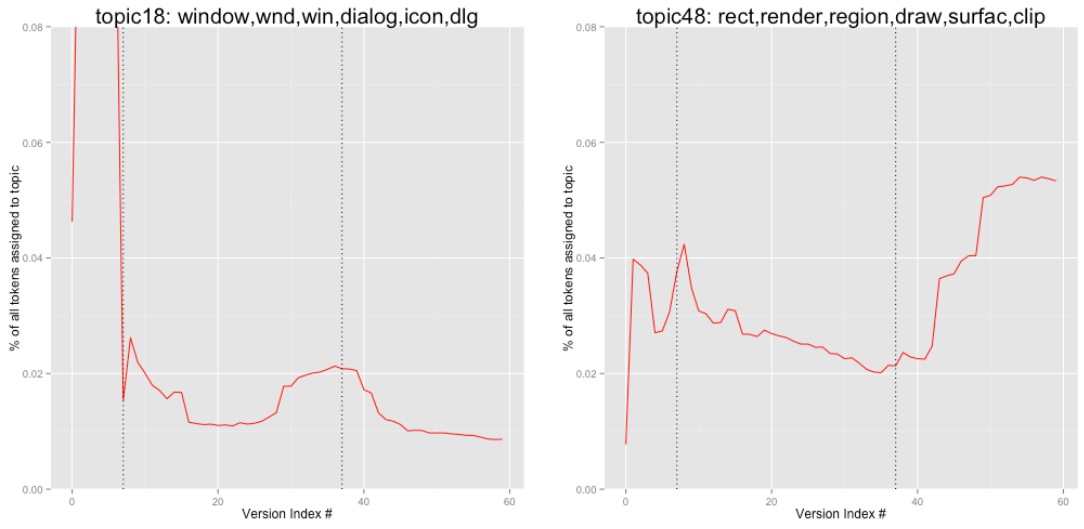


Figure 5. Topics representing user interface components (18) and primitive drawing components (48). Despite the organization of user interface and the addition of new features, the amount of code devoted to describing it has decreased. The amount of code devoted to new graphics, on the other hand, has increased as expected to support the changes visible to users.

There are two explanations for the trend exhibited in topic 18. First, while Mozilla was rewriting the basic interface components that were abandoned with the rest of the Communicator code, its developers were able to implement a much more compact and efficient design. This possibility is visible in the Application Suite period as topic 18 and related components exhibit a long period of stabilization before being revisited late in the software's lifetime. During the Firefox period, however, topic 18 is actually reduced in size significantly. The second possibility is that these functions were handled elsewhere in the Mozilla source code, yet none of the components describing user interface objects show increases that offset the source code removed after the release of Firefox 3.0 in 2008 (index number 40).[3] If anything, Firefox's interface has become more complex than the two previous applications, supporting a wider variety of visual styles and allowing for more complicated forms of interaction than just button pushing or simple text entry. The sudden drop in user interface components in topic 18 considered alongside an increase in the complexity of the browser's graphics libraries in topic 48 is surprising; at the very least, the user interface logic should have remained constant, even if the rise in graphics rendering algorithms were just added window dressing. In other words, the components represented in these two and similar topics shows that Firefox underwent a significant structural change with respect to the relationship of its interface to the rest of its components.

27

Returning to the unprocessed source code collected for this study shows that these functions were likely evacuated from the C++ codebase and rewritten in another language. As Figure 6 shows, there is a steady rise in JavaScript code beginning early in the Suite period that suddenly increases during the Firefox period at version 40, quickly reaching a point at which over 40% of the source code for the browser is no longer in C++. Mozilla's changing user interface, in other words, was preceded by a substantial material change to the relationship between the browser's user interface and core components. Without topic modeling the JavaScript code, the exact nature of the relationship between the two sets of source code cannot be determined; however, this trend does help to highlight a narrative present in developer documentation that does not appear in Mozilla's public announcements, blogs, or other avenues used to advertise its software.

28

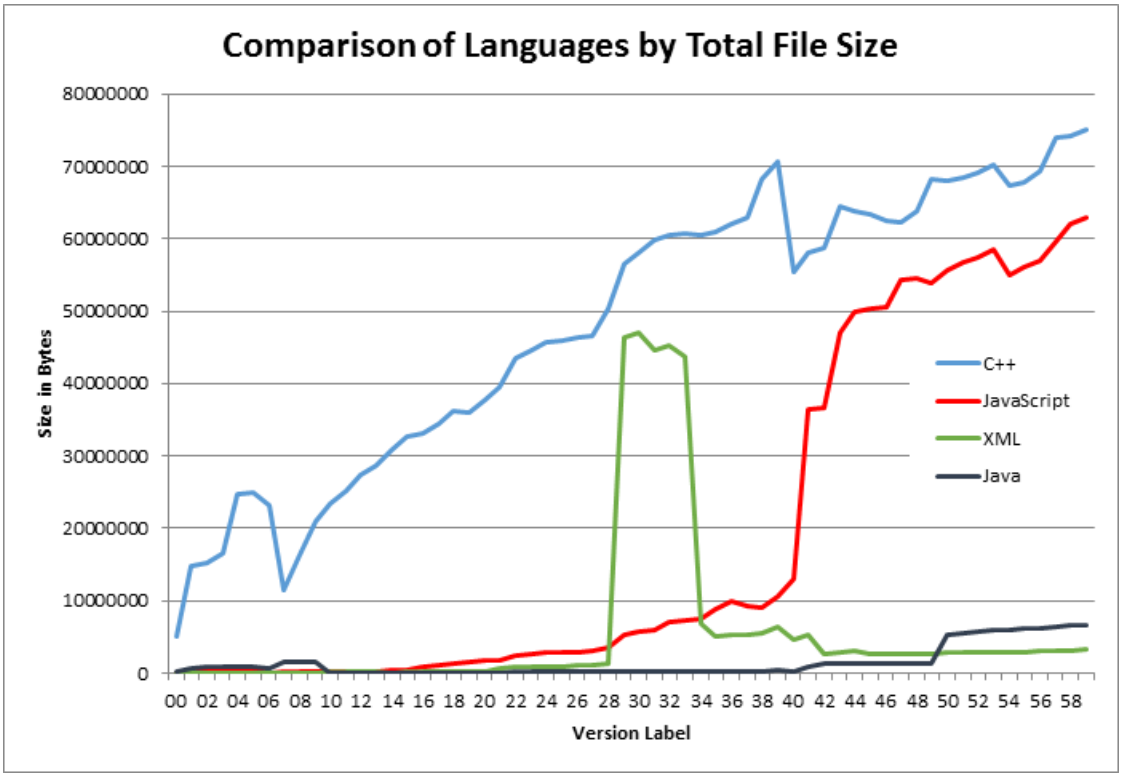


Figure 6. Amount of source code from major languages used in the Mozilla Corpus. XML includes all known variants used by Netscape/Mozilla (XML, XUL, RDF).

The initial drop in user interface component between Communicator and Suite reflects Mozilla's recovery and re-envisioning of the cross-platform toolkit strategy. As Mozilla's lead designers note in the 2003 developer roadmap,

29

abandoning Netscape's code "allowed [them] to write a cross-platform application front end once, instead of writing native-OS-toolkit-based front ends for at least three platforms" [Eich and Hyatt 2004]. Collectively, Figure 5 and 6's graph give us a way to visualize the scope of this change and its impact on the structure of Mozilla's software. In short, Netscape's original insistence on having an *exactly* replicated user experience across Windows, MacOS, and Linux, meant that its software could not simply draw user interfaces using native graphics libraries. Instead, there would be three versions of each component, one for each operating system that would on the surface look and feel identical to its counterparts. A universal interface, in other words, would only be possible by suppressing the defining features of each platform that Netscape's toolkit was built for, marking the desktop as a site of struggle between Netscape, Apple, Microsoft, and the UNIX community. Users, of course, had little role in this struggle aside from Netscape's hope that they would more readily associate tasks with its visual signifiers rather than those of its competitors.

30 Even though Mozilla would abandon Netscape's totalizing desire to erase from the desktop all visual signifiers other than its own, the alternative would exert a lasting influence on later software. Rather than produce multiple logically separate, yet visually identical, interfaces, Mozilla elected to build a flexible cross platform support structure for interfaces. Because the content rendered within web browser would look the same on every platform in order to fulfill the shared standards for HTML, CSS, and JavaScript, Mozilla began to rewrite its user interface logic in those languages. The interface would no longer be a static design, written in C++ and frozen in place during compilation; rather, it would exist separately from the browser code and be handled by the same core components that managed web content, referred to within Mozilla as the GECKO engine. Because Firefox's interfaces are now separate from the C++ code base and loaded at runtime into the GECKO engine, they could in theory be modified any time to almost any degree. By Version 3.0 of Firefox (index number 40), the browser is less a distinct application, with components that are essentially frozen in place via compilation until the next version's release, as it is a highly malleable configuration of the GECKO engine shaped around those tasks that Mozilla determines comprise "web browsing". Here, the results of Mozilla's plan to break up and modularize Application Suite becomes clearer. Rather than abandoning the cross platform toolkit entirely, Mozilla would pare down its toolkit to a minimal core set of components and distribute its applications in the form of separate, highly specific configurations of GECKO. While Mozilla has succeeded in rebranding itself through a narrative of open source independence, a textual history of its software shows that a kernel of Netscape's corporate influence remains tightly knit into the core of its software, continuing to shape its production.

V. Conclusions

31 This project shows the value of cross disciplinary dialog between digital media studies and the digital humanities. In addition to digital humanities methods assisting in the study of an increasingly complicated information culture, digital media's interrogation of computer science can help us find new ways to shape analytic algorithms around the methodological and theoretical goals of the humanities beyond investigations of formalism or structuralist principles. In particular, text analytics can help provide a sense of narrative in what Matthew Fuller and Andrew Goffey refer to as "grey media", or software that appears to resist hermeneutics because it provides little in the way of aesthetic, narratological, or cultural engagement [Fuller and Goffey 2012]. Even in those digital genres that do invite hermeneutic and cultural readings, text analytics could add another dimension of analysis by opening up the grey software that support interactive components. A game's internal logics could be examined for cultural or political assumptions about the game world or its players through modding and data extraction tools. Additionally, textual approaches to software history and LDA in particular are capable of providing a sense of scale and scope not accessible via paratextual sources like developer interviews or documentation. While many of the moments discussed in this essay can be traced to developer documentation, references to "a new interface" or "a more streamlined JavaScript interpreter" don't reflect the amount of code changed or give any indication of the steps taken to make possible such alterations. Methods like DiffLDA may even be capable of identifying patterns that aren't well documented by developers. The trends shown in Figure 3 are beyond the scope of the narrative presented in this article, but it's worth noting that they highlight a significant change in source code that was not presented by developers in documentation and only traceable through careful review of the notes they kept internally via their Bugzilla tracking system.

32 Beyond methodological considerations, this case study reminds us that software is often produced within a corporate and commercial space. A commitment to openness and transparency does not necessarily produce an environment in

which users are free to use software or data as they see fit. Openness for the sake of openness accomplishes little, especially if those in control of an open resource institute changes that users may object to without their consent. In regards to software studies, scholars should treat all interfaces — even to open access databases or open source software — with suspicion. The textual history of Mozilla highlights the arbitrary nature of the linkages between the interface and the core components of a software system. Interfaces not only represent the developer's intended use for their software but also how they want their software represented within the broader information economy. Developers can represent their algorithms in any way they choose provided their software appears to be working "correctly" to its users. The aesthetic dimension of the "representational gap" described by Black here becomes political: programmers can wrap their software in interfaces that suit the narratives or personas they wish to be seen through even if their software contains elements which may not align with those self-representations [Black 2012]. Scholars in digital media should therefore approach interfaces with suspicion in much the same way that literary scholars tend to disregard authorial intent as a form of prima facie evidence. While digital humanists should continue to commit to principles of openness, we should also be aware that a rhetoric of openness can be co-opted in the service of exclusionary practices and take care to consider the implications of any access policy we institute over the public face of our projects. The care that free software advocates take to distinguish their principles from those of the open source community is a good example in this regard.

Importantly, adapting LDA to work within the constraints of textual history shows that analytics algorithms can and should be examined closely for the ways in which the implicit theoretical assumptions they make about interpretive practices conflict or complement humanities critical practices. Although this study still leaves much to be done, it nonetheless does show how digital humanists can begin to bring critical theory into an algorithmic space. Every step in a workflow — from preprocessing to statistical measures to postprocessing — should be understood as a critical and interpretive move because each will inevitably shape the scholarship performed before, during, and following analysis. Whereas the scientists and engineers who produce these algorithms are concerned primarily with their instrumental value, humanists can fulfill a much needed role in exploring how these algorithms change data in ways that privilege one form of interpretation over another. Digital humanists have a richer, more complex, and nuanced understanding of cultural sources and have more experience at applying heuristics to them than data scientists. We should not be afraid to explore the critical possibilities of these algorithms even if only to be able to talk back to them in substantive ways.

Notes

[1] These numbers were taken from Mozilla's "At a Glance: Fast Facts" and Wikimedia's "Traffic Analysis Report for March 2015" on August 14, 2015. They can be found at https://blog.mozilla.org/press/ataglance/ and http://stats.wikimedia.org/archive/squid_reports/2015-03/SquidReportClients.htm, respectively.

[2] Readers are welcome to view the complete gallery of topic graphs at http://mblack.us/moztm-data/index.html

[3] The raw token counts show a very gradual increase over time during the dip on the graph prior to version 30, indicating that almost nothing was added rather than that something was removed. By contrast, the raw token counts following version 40 show a significant decrease.

Works Cited

Aarseth 1997 Aarseth, E. J. Cybertext: perspectives on ergodic literature. Johns Hopkins University Press, Baltimore (1997).

Andreesen 1996 Andreesen, M. ONE for All (1996). Available from http://web.archive.org/web/19961026213506/http://www3.netscape.com/comprod/columns/techvision/one.html. [4 September 2014]

Baker 2007 Baker, M. Mozilla's New Focus on Thunderbird and Internet Communications (2007). Available from http://blog.lizardwrangler.com/?p=158. [31 August 2014]

Berry 2011 Berry, D. M. The philosophy of software. Palgrave Macmillan, London (2011).

Black 2012 Black, M. L. "Narrative and Spatial Form in Digital Media A Platform Study of the SCUMM Engine and Ron Gilbert's The Secret of Monkey Island." Games and Culture, 7.3 (2012): 209-237.

Boellstorff 2008 Boellstorff, T. Coming of age in Second Life: An anthropologist explores the virtually human. Princeton University Press, Princeton (2008).

Bolter 1991 Bolter, J. D. Writing space: The computer, hypertext, and the history of writing. Lawrence Erlbaum Associates, Hillsdale (1991).

Bolter and Gromala 2003 Bolter, J. D. and Gromala, D. Windows and mirrors: Interaction design, digital art, and the myth of transparency. MIT Press, Cambridge (2003).

Bolter and Grusin 2000 Bolter, J. D. and Grusin, R. A. Remediation: Understanding new media. MIT Press, Cambridge (2000).

Chun 2011 Chun, W. H. K. Programmed visions: Software and memory. MIT Press, Cambridge (2011).

Clement et al. 2013 Clement, T. Tcheng, D., Auvil, L. Capitanu, B., and Monroe, M. “Sounding for Meaning: Using Theories of Knowledge Representation to Analyze Aural Patterns in Texts”, Digital Humanities Quarterly 7.1 (2013). Available from http://www.digitalhumanities.org/dhq/vol/7/1/000146/000146.html. [29 August 2014].

Cusumano and Yoffie 1998 Cusumano, M. A., and Yoffie, D. B. Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft. Simon and Schuster, New York (1998).

De Souza et al. 2005 De Souza, C., Froehlich, J., and Dourish, P. “Seeking the source: software source code as a social and technical artifact.” Proceedings of the 2005 international ACM SIGGROUP conference on supporting group work (2005): 197-206.

Eich and Hyatt 2004 Eich, B. and Hyatt, D. Mozilla Development Roadmap (2004). Available from https://web.archive.org/web/20040610135931/http:/www.mozilla.org/roadmap.html. [5 September 2014]

Fuller and Goffey 2012 Fuller, M., and Goffey, A. Evil Media. MIT Press, Cambridge (2012).

Fung et al. 2007 Fung, A., Graham, M., and Weil, D. Full Disclosure: The Perils and Promise of Transparency. Cambridge University Press, Cambridge (2007).

Galloway 2004 Galloway, A. R. Protocol: how control exists after decentralization. MIT Press, Cambridge (2004).

Gethers and Poshyvanyk 2010 Gethers, M., and Poshyvanyk, D. “Using relational topic models to capture coupling among classes in object-oriented software systems.” Software Maintenance (ICSM), 2010 IEEE International Conference (2010): 1-10.

Golumbia 2009 Golumbia, D. The cultural logic of computation. Harvard University Press, Cambridge (2009).

Gray 2009 Gray, M. L. Out in the country: Youth, media, and queer visibility in rural America. NYU Press, New York (2009).

Hayles 2008 Hayles, N. K. How we became posthuman: Virtual bodies in cybernetics, literature, and informatics. University of Chicago Press, Chicago (2008).

Jansz et al. 2010 Jansz, J., Avis, C., and Vosmeer, M. “Playing The Sims 2: An exploration of gender differences in players’ motivations and patterns of play.” New Media and Society, 12.2 (2010): 235-251.

Jockers 2013 Jockers, M. L. Macroanalysis: Digital methods and literary history. University of Illinois Press, Champaign (2013).

Kittler 1997 Kittler, F. A. “There is No Software.” Literature, Media, Information Systems. G + B Arts, Amsterdam (1997), pp 147-155.

Kuhn et al. 2007 Kuhn, A., Ducasse, S., and Gírba, T. “Semantic clustering: Identifying topics in source code.” Information and Software Technology, 49.3 (2007): 230-243.

Latour 1987 Latour, B. Science in action: How to follow scientists and engineers through society. Harvard University Press, Cambridge (1987).

Lessig 2009 Lessig, L. “Against Transparency.” The New Republic (2009). Available from http://www.newrepublic.com/article/books-and-arts/against-transparency [15 August 2015]

Linstead et al. 2007 Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., and Baldi, P. “Mining concepts from code with probabilistic topic models.” Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (2007): 461-464.

Lukins et al. 2008 Lukins, S. K., Kraft, N. A., and Etzkorn, L. H. “Source code retrieval for bug localization using latent dirichlet allocation.” Reverse Engineering, 2008. WCRE'08. 15th Working Conference (2008): 155-164.

Mahoney 1996 Mahoney, M. S. “Issues in the History of Computing.” History of programming languages---II. ACM Press, New York (1996), pp. 772-781.

Manovich 2001 Manovich, L. (2001). The language of new media. MIT Press, Cambridge (2001).

Marino 2006 Marino, M. C. “Critical code studies.” Electronic book review 4 (2006). Available from http://www.electronicbookreview.com/thread/electropoetics/codology [4 July 2014]

Markley 1996 Markley, R. “Boundaries: Mathematics, Alienation, and the Metaphysics of Cyberspace.” In R. Markley (ed), Virtual Realities and Their Discontents. Johns Hopkins University Press, Baltimore (1996), pp. 55-78.

McGann 1992 McGann, J. J. A critique of modern textual criticism. University Press of Virginia, Charlottesville (1992).

McGann 2001 McGann, J. J. Radiant textuality: Literature after the world wide web. Palgrave, New York (2001).

McPherson 2012 McPherson, T. “US Operating System at Mid-Century: The Intertwining of Race and UNIX.” In L. Nakamura and P. Chow White (eds), Race after the Internet. Routledge, New York (2012), pp. 21-37.

Mozilla 2002 Mozilla Launches Mozilla 1.0 (2002). Available from http://blog.mozilla.org/press/2002/06/mozilla-org-launches-mozilla-1-0. [2 September 2014].

Mozilla 2003 Mozilla.org Announces Launch of the Mozilla Foundation to Lead Open-Source Browsers Efforts (2003). Available from https://web.archive.org/web/20030719114119/http:/www.mozillafoundation.org/press/mozilla-foundation.html [2 September 2014]

Murray 1997 Murray, J. H. Hamlet on the holodeck: The future of narrative in cyberspace. MIT Press, Cambridge (1997).

Nakamura 2009 Nakamura, L. “Don't hate the player, hate the game: The racialization of labor in World of Warcraft.” Critical Studies in Media Communication, 26.2 (2009): 128-144.

Nardi 2010 Nardi, B. My life as a night elf priest: An anthropological account of World of Warcraft. University of Michigan Press, Ann Arbor (2010).

Netscape 1998 Netscape Announces Plan to Make Next-Generation Communicator Source Code Available for Free on the Net (1998). Available from http://web.archive.org/web/20021001071727/wp.netscape.com/newsref/pr/newsrelease558.html [4 September 2014]

Norman 1988 Norman, D. A. The psychology of everyday things. Basic books, New York (1988).

Sample 2013 Sample, M. “Criminal Code: Procedural Logic and Rhetorical Excess in Video Games.” Digital Humanities Quarterly, 7.1 (2013). Available from http://www.digitalhumanities.org/dhq/vol/7/1/000153/000153.html [6 August 2014]

Stallman 2002 Stallman, R. M. “Why ‘Free Software’ is Better than ‘Open Source.’” In J. Gray (ed), Free Software, Free Society: The Selected Essays of Richard M. Stallman. GNU Press, Boston (2002), pp. 57-62.

Thomas et al. 2011 Thomas, S. W., Adams, B., Hassan, A. E., and Blostein, D. “Modeling the evolution of topics in source code histories.” Proceedings of the 8th working conference on mining software repositories (2011): 173-182.

Torvalds and Diamond 2001 Torvalds, L and Diamond, D. Just For Fun: The Story of an Accidental Revolutionary. HarperCollins, New York (2001).

Underwood and Sellers 2012 Underwood, T., and Sellers, J. “The emergence of literary diction.” Journal of Digital Humanities, 1.2 (2012). Available from http://journalofdigitalhumanities.org/1-2/the-emergence-of-literary-diction-by-ted-underwood-and-jordan-sellers/ [7 August 2014]

Wardrip-Fruin 2009 Wardrip-Fruin, N. Expressive Processing: Digital fictions, computer games, and software studies. MIT Press, Cambridge (2009).

Williams and Hollingsworth 2005 Williams, C. C., and Hollingsworth, J. K. “Automatic mining of source code repositories to improve bug finding techniques.” Software Engineering, 31.6 (2005): 466-480.

Winograd and Flores 1986 Winograd, T. and Flores F. Understanding computers and cognition: A new foundation for design. Intellect Books, Addison-Wesley, Reading (1986).

Witte et al. 2008 Witte, R., Li, Q., Zhang, Y., and Rilling, J. “Text mining and software engineering: an integrated source

code and document analysis approach.” IET software, 2.1 (2008): 3-16.

Ying et al. 2004 Ying, A. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. “Predicting source code changes by mining change history.” Software Engineering 30.9 (2004): 574-586.