Recognizing recurrent development behaviors corresponding to Android OS release life-cycle

Pavel Senin
Collaborative Software Development Laboratory
Information and Computer Sciences Department
University of Hawaii at Manoa
Honolulu, Hawaii, 96822
senin@hawaii.edu

***Abstract*—Within the field of software repository mining (MSR) researchers deal with a problem of discovery of interesting and actionable information about software projects. It is a common practice to perform analyzes on the various levels of abstraction of change events, for example by aggregating change-events into time-series. Following this, I investigate the applicability of SAX-based approximation and indexing of time-series with *tf∗idf* weights in order to discover recurrent behaviors within development process. The proposed workflow starts by extracting and aggregating of revision control data and followed by reduction and transformation of aggregated data into symbolic space with PAA and SAX. Resulting SAX words then grouped into dictionaries associated with software process constraints known to influence behaviors, such as time, location, employment, etc. These, in turn, are investigated with the use of *tf∗idf* statistics as a dissimilarity measure in order to discover behavioral patterns.**

As a proof of the concept I have applied this technique to software process artifact trails corresponding to Android OS[1] development, where it was able to discover recurrent behaviors in the "new code lines dynamics" before and after release. By building a classifier upon these behaviors, I was able to successfully recognize pre- and post-release behaviors within the same and similar sub-projects of Android OS.

Keywords: software process, recurrent behaviors, data-mining

I. Introduction

By the large body of previous research it has been shown, that software process artifact trail (change events and associated metadata) is a rich source of process and developers' information and characteristics. The ability to discover recurrent behaviors with Fourier Analysis of change events is explained in [1], while another work [2] connects recurrent behaviors and software product quality. Thus, potentially, it is possible to relate recurrent behaviors to software product quality and to software process efficiency. The main part of a toolkit aiding such research is not only an efficient mechanism of recurrent behaviors discovery, but a mechanism of recognition of social and project-related constraints modulating these behaviors. This paper presents my exploratory study resulted in a universal framework for temporal partitioning and mining of software change artifacts. As an evaluation example, it presents a recurrent behaviors discovery from the data extracted from Android SCM (software configuration management) system.

The rest of the paper is organized as follows. In Section 2, I discuss the motivation, results of previous work in MSR and present the research questions. In Section 3, I consider the workflow, data selection, collection, partitioning, and describe algorithms and methods. Section 4 presents results and the contribution. Finally, in Section 5, I discuss limitations and possible extension of this work.

II. Motivation

Software development is a human activity resulting in a software product. The software process is a structure imposed on the software development. This structure identifies a set and an order of activities performed to design, develop and maintain software systems. Examples of such activities include design methods; requirements collection and creation of UML diagrams; requirements testing; performance analysis, and others. The intent behind a software process is to structure and coordinate human activities in order to achieve the goal - deliver a software system successfully. Many processes and process methodologies exist today, and it has been found, that the amount of time and effort needed to complete a software project, and the quality of the final product, are heavily affected by the software process choice [3]. Thus, studying software processes is one of the important areas of software engineering.

Traditionally, the software process study is built from top to bottom: it requires the researcher to guess a whole process, or to notice a recurrent pattern of behavior upfront, and to study it in a variety of settings later. These empirical studies usually involve two expensive and limited in scale techniques: interviewing and monitoring of the developers. Furthermore, these techniques are virtually impossible to apply within open-source project settings where a diverse development community scattered over the globe. Fortunately, current advances in software configuration management (SCM) technologies enable researchers to study

[1] http://source.android.com

software process by mining software artifact trails [4], such as change logs, bug and issue tracking systems and mailing lists archives.

Mining of large software repositories demands advanced techniques allowing to tame with the complexities of data extraction and its analysis. These challenges are not new to the data-mining community and an enormous wealth of methods, algorithms and data structures have been developed to address these issues. While some of these approaches were already implemented within the field, such as finding of trends, periodicity and recurrent behaviors through the linear predictive coding and cepstrum coefficients [5], Fourier Transform [1] and coding [6], many are yet to be tried.

In this paper, I investigate the application of Symbolic Aggregate Approximation [7] and the term frequency–inverse document frequency weight statistics ($tf*idf$) [8] to the problem of discovering recurrent behaviors from software process artifacts. The motivation behind this choice is coming from the demonstration of outstanding performance by SAX in time-series mining, and from the wide range of successful applications of $tf*idf$ statistics, which is focusing on measuring the degree of dissimilarity as the opposite to convenient similarity metrics. Implementation of this approach I validate on Android SCM data.

A. Research question

In this exploratory work I am investigating the applicability of Lin&Keogh symbolic approximation technique combined with $tf*idf$ statistics to the discovery of recurrent behaviors from SCM trails of Android OS. The research questions I am addressing are:

- Which kinds of SCM data need to be collected for such analyzes?
- What is the optimal approach to data representation and a data storage configuration?
- Which partitioning (slicing) is appropriate, and which set of parameters should one use for SAX approximation?
- What is the general mining workflow?

III. Experimental setup and methods

In this section I explain the steps of the recurrent behaviors discovery workflow along with their theoretical background.

A. Data collection and organization

As with many other large open-source projects, Android OS has been in the development for many years. It is “an open-source software stack for mobile phones and other devices”, which is based on the Linux 2.6 monolithic kernel. Development of Android was begun by Android Inc., the small startup company. In 2005, the company was acquired by Google which formed the Open Handset Alliance - a consortium of 84 companies which announced the availability of the Android Software Development Kit (SDK) in November 2007. The Android OS code is open and released under the Apache License.

Google platform is used for hosting, issue and bug tracking systems, whether Git is used as the distributed version control system for Android. The source code is organized into more than 200 of sub-projects by function (kernel, UI, mailing system, etc.) and underlying hardware (CPU type, bluetooth communication chip, etc.). There are about two million change records registered in the Android SCM by more than eleven thousands of contributors within an eight year span. The richness of this data makes Android SCM very interesting repository for exploring.

By using provided Google Data API for bugs and issues data retrieval, and custom coded Git repository data collection engine, I have collected information about bugs and issues, the revision tree, authors and committers, change messages, and affected targets. In addition to that data, by creating a local mirror and by iterating over changes, I was able to recover the auxiliary data for the most of the change records. This auxiliary data provides quantitative summary of added, modified, and deleted targets, as well as the summary about LOC changes: added, modified or deleted lines. All this information was stored in the relational database. Main tables of this database correspond to change and issue events; these accompanied with change target tables, issue details, comments, and tables for contributor authentication. Overall, the database was normalized and optimized for the fast retrieval of change and issue information using SQL language.

The collected data constitute almost full set of collectible artifacts. The only lacking information is the precise information for source-code line changes, which I intentionally omitted in this step due to the storage space and collection time constraints. Despite of being collected, bugs and issues data has not been included into recurrent behaviors discovery experiments in this work mostly due to the complexity of change-issue relations. However, as was shown by previous research, this data is a valuable source of information for recurrent behaviors discovery [2].

B. Temporal data partitioning

By following the previous research targeting social characteristics of committers [2], as well as the release pattern discovery [6], I have partitioned and organized the collected change trails by the time of the day using time windows of

- Full day, 12AM - 12AM
- Late night, 12AM - 04AM
- Early morning, 04AM - 08AM
- Day, 08AM - 05PM
- Night, 05PM - 12AM

For every of these windows, I then aggregated values for commits, added, edited, or deleted targets and lines, producing equidistant time-series abstraction of software development activity.

One of the effects of this data transformation is an instant increase of the number of change data entities by the factor of 5 and production of very sparse equidistant time-series. In order to reduce the sparseness and the complexity (dimensionality) of data, two additional procedures were applied within the post-collection data treatment step: PAA and SAX.

C. Piecewise Aggregate Approximation (PAA)

PAA performs a time-series feature extraction based on segmented means [9]. Given a time-series X of length n, application of PAA transforms it into vector $\bar{X} = (\bar{x}_1, ..., \bar{x}_M)$ of any arbitrary length $M \leq n$ where each of $\bar{x}_i$ is calculated by the following formula:

$$\bar{x}_i = \frac{M}{n} \sum_{j=n/M(i-1)+1}^{(n/M)i} x_j \tag{1}$$

This simply means that in order to reduce the dimensionality from n to M, we first divide the original time-series into M equally sized frames and secondly compute the mean values for each frame. The sequence assembled from the mean values is the PAA transform of the original time-series.

It worth noting, that PAA reduction of original data satisfies to a bounding condition, and guarantees no false dismissals in upstream analyzes as shown by Keogh et al. [10] by introducing the distance function:

$$D_{PAA}(\bar{X}, \bar{Y}) \equiv \sqrt{\frac{n}{M}} \sqrt{\sum_{i=1}^{M} (\bar{x}_i - \bar{y}_i)} \tag{2}$$

and showing that $D_{PAA}(\bar{X}, \bar{Y}) \leq D(X, Y)$.

D. Symbolic Aggregate approXimation (SAX)

Symbolic Aggregate approXimation extends the PAA-based approach, inheriting algorithmic simplicity and low computational complexity, while providing satisfactory sensitivity and selectivity [7].

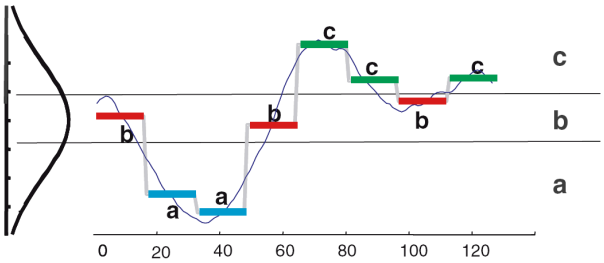


Figure 1: The illustration of the SAX approach taken from [7] depicts two pre-determined breakpoints for the three-symbols alphabet and the conversion of the time-series of length $n = 128$ into PAA representation followed by mapping of the PAA coefficients into SAX symbols with $w = 8$ and $a = 3$ resulting in the string **baabccbc**.

SAX transforms a time-series X of length n into a string of arbitrary length ω, where $\omega << n$ typically, using an alphabet A of size $a \geq 2$. The SAX algorithm consist of two steps: during the first step it transforms the original time-series into a PAA representation and this intermediate representation gets converted into a string during the second step. Use of PAA at the first step brings the advantage of a simple and efficient dimensionality reduction while providing the important lower bounding property. The second step, actual conversion of PAA coefficients into letters, is also computationally efficient and the contractive property of symbolic distance was proven by Lin et al. in [11].

Discretization of the PAA representation of a time-series into SAX is implemented in a way which produces symbols corresponding to the time-series features with equal probability. The extensive and rigorous analysis of various time-series datasets available to the authors has shown that normalized by the zero mean and unit of energy time-series follow the Normal distribution law. By using Gaussian distribution properties, it's easy to pick a equal-sized areas under the Normal curve using look-up tables [12] for the cut lines coordinates, slicing the under-the-Gaussian-curve area. The x coordinates of these lines called "breakpoints" in the SAX algorithm context. The list of breakpoints $B = \beta_1, \beta_2, ..., \beta_{a-1}$ such that $\beta_{i-1} < \beta_i$ and $\beta_0 = -\infty$, $\beta_a = \infty$ divides the area under $N(0, 1)$ into a equal areas. By assigning a corresponding alphabet symbol $alpha_j$ to each interval $[\beta_{j-1}, \beta_j)$, the conversion of the vector of PAA coefficients $\bar{C}$ into the string $\hat{C}$ implemented as follows:

$$\hat{c}_i = alpha_j, \text{ iif } \bar{c}_i \in [\beta_{j-1}, \beta_j) \tag{3}$$

SAX introduces new metrics for measuring distance between strings by extending Euclidean and PAA (2) distances. The function returning the minimal distance between two string representations of original time series $\hat{Q}$ and $\hat{C}$ is defined as

$$MINDIST(\hat{Q}, \hat{C}) \equiv \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w} (dist(\hat{q}_i, \hat{c}_i))^2} \tag{4}$$

where the $dist$ function is implemented by using the look-up table for the particular set of the breakpoints (alphabet size) as shown in Table I, and where the singular value for each cell (r, c) is computed as

$$cell_{(r,c)} = \begin{cases} 0, \text{ if } |r - c| \leq 1 \\ \beta_{\max(r,c)-1} - \beta_{\min(r,c)-1}, \text{ otherwise} \end{cases} \tag{5}$$

As shown by Lin et al., this SAX distance metrics lower-bounds the PAA distance, i.e.

$$\sum_{i=1}^{n} (q_i - c_i)^2 \geq n(\bar{Q} - \bar{C})^2 \geq n(dist(\hat{Q}, \hat{C}))^2 \tag{6}$$

	a	b	c	d
a	0	0	0.67	1.34
b	0	0	0	0.67
c	0.67	0	0	0
d	1.34	0.67	0	0

Table I: A look-up table used by the MINDIST function for the $a = 4$

It worth noting, that SAX lower bound was examined by Ding et al. [13] in great detail and found to be superior in precision to the spectral decomposition methods on bursty (non-periodic) data sets.

E. Symbolic approximation and indexing

As explained above, application of SAX to the single time-series results in its symbolic representation which is much shorter (reduced in the dimensionality) and easier to manipulate.

By following a sliding window sub-series extraction and SAX indexing technique described in detail by Lin et al. in [7] and Keogh et al. in [11], I have built a number of symbolic indexes for every time-series generated at partitioning step (III-B) combining following parameters for SAX transformation:

- three sizes for sliding window reflecting natural intervals of a week (7 days), two weeks (14 days) and a month (30 days);
- 4 PAA steps for a weekly window, 6 PAA steps for a bi-weekly window, and 10 PAA steps for a monthly window;
- 3 letters alphabet for weekly window, 5 letters for bi-weekly, and a 7 letters alphabet for monthly window.

These indexes were stored in the same relational database, organized and indexed in order to allow the fast retrieval of SAX words and their frequencies for a specific project, a contributor, a time-interval, a SAX parameters set, or any combination of these fields.

F. Behavioral portrait

Here I define a term of "behavioral portrait" of a contributor c as the set of all observed SAX words in her software artifact trail(s):

$$BP_c = \{(w_1, f_1), (w_2, f_2), ..., (w_n, f_n)\} \tag{7}$$

where each pair (w_1, f_1) corresponds to the observed SAX word and its frequency. This portrait can be further specified by project, time-interval and SAX parameters set. Also it can be easily extended from the individual contributor to a team, whose "behavioral portrait" is a union set of "behavioral portraits" of team members.

G. Token-based distance metrics application to behavioral portraits

In my previous experiments I have measured the performance of three similarity metrics when applied to the behavioral portraits.

The first metrics I have tried is weighted by SAX Euclidean similarity distance defined for common to two behavioral portraits words:

$$D(S,T) = \sqrt{\sum_{S \cap T} (MINDIST(s_i, t_i) * \|F_{s_i} - F_{t_i}\|)^2} \tag{8}$$

where S and T are two behavioral portraits whose words are ordered by frequency.

The second metrics I have tried is the Jaccard similarity coefficient between two behavioral portraits S and T which is simply

$$J_\delta(S,T) = \frac{|S \cup T| - |S \cap T|}{|S \cup T|} \tag{9}$$

The third metrics I have tried is the *tf∗idf* similarity which defined as a dot product

$$TFIDF(S,T) = \sum_{\omega \in S \cap T} V(\omega, S) \cdot V(\omega, T) \tag{10}$$

where

$$V(\omega, S) = \frac{V'(\omega, S)}{\sqrt{\sum_{\omega'} V'(\omega, S)^2}} \tag{11}$$

is a normalization of *tf∗idf* (product of token frequency and inverse document frequency):

$$V'(\omega, S) = \log(TF_{\omega,S} + 1) \cdot \log(IDF_\omega) \tag{12}$$

where $TF_{\omega,S}$ is a normalized token frequency

$$TF_{\omega,S} = \frac{|\omega|}{|S|} \tag{13}$$

and IDF_ω is a measure of the general importance of the pattern among all users

$$IDF_\omega = \frac{|D|}{DF(\omega)} \tag{14}$$

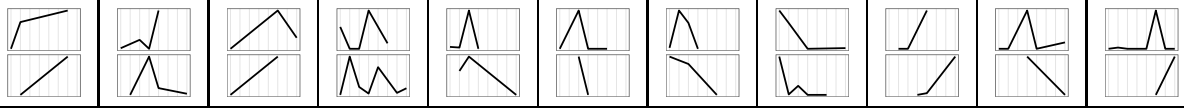
where $|D|$ is cardinality of D - the total number of users, and $DF(\omega)$ is the number of users having ω pattern in their activity set.

While first two metrics demonstrated very poor performance in the clustering tests (discussed in the section III-H), the *tf∗idf* similarity statistics performed very well and is presented in this work.

H. Clustering

As a universal tool for the exploration of derived behavioral portraits through their partitioning, and for assessment of the metrics' performance, I used hierarchical clustering. The k-means clustering was used in the validation of the class assignment and for general assessment of the validity of the approach.

Table II: Patterns observed within pre- and post-release behavioral portraits, their *tf∗idf* weights and sample, **not normalized** curves. *(here pre-x.x and post-x.x rows of the upper table correspond to pre-release and post-releases of Android OS version x.x; columns of the table correspond to non-trivial patterns observed in all behavioral portraits; cells of the table contain tf∗idf weights computed for a particular SAX word in a particular behavioral portrait)*

release	"bbac"	"abca"	"babc"	"bbba"	"bcaa"	"bcbb"	"ccaa"	"cbaa"	"bbcb"	"bbbb"	"bbbc"
post-2.0	0.63	0	0.63	0	0	0	0	0.39	0.24	0.06	0
post-1.0	0	0.93	0	0	0	0	0	0	0	0.09	0.36
post-1.5	0	0	0	0	0	0	0	0	0.79	0.61	0
pre-1.5	0	0	0	0.23	0.23	0.91	0	0.14	0.18	0	0.09
pre-2.0	0	0	0	0	0	0	0	0	0	1	0
pre-1.0	0	0	0	0	0	0	0.79	0	0	0.08	0.61
unnormalized sample curves corresponding to patterns											

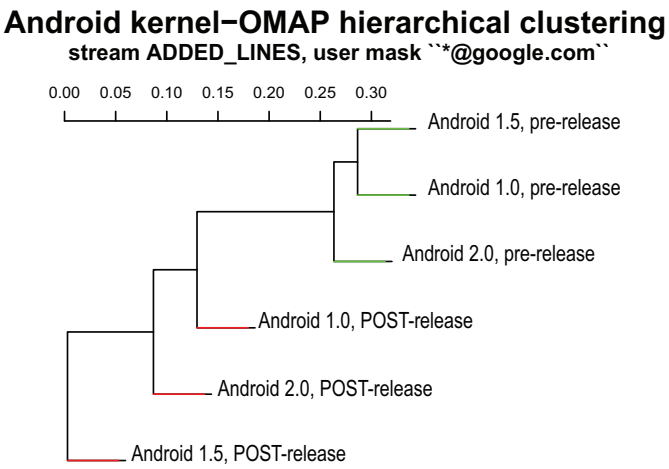


Figure 2: Hierarchical clustering of pre- and post- release behavioral portraits corresponding to the new code lines dynamics of google.com affiliated contributors.

IV. Results

For the experiments related to this work I have tried a number of contributors partitioning schemes, variety of time-intervals and sub-projects selections observing satisfactory performance of investigated approach. However, due to the space constraint of this paper, I present only single validation experiment in this section as the proof of concept.

A. Kernel-OMAP life cycle patterns discovery

I have arbitrary selected the Android kernel-OMAP project as one of the large sub-projects in Android OS. It is the Android kernel implementation for OMAP-based (a proprietary system on chips based on ARM architecture processor by Texas Instruments) devices.

As a "training set" for discovery of behavioral portraits of pre- and post-release patterns, I chose three Android releases: *Android 1.0*, *Android 1.5 "Cupcake"* and *Android 2.0 "Eclair"*. For each of these I generated behavioral portraits corresponding to four weeks before the release - *pre-release behavioral portrait*, and to four weeks after release - *post-release behavioral portrait* having in place an additional constraint on contributors and the artifacts trail. I have selected contributors affiliated with $google.com$ e-mail domain only, expecting that paid developers will have much more consistent behavior [2]. By selecting the $added\_lines$ artifacts stream only, I additionally limited the scope of the analyzes and the complexity of captured behaviors to the "new code lines dynamics" only. The almost perfect clustering picture (Figure 2) obtained with hierarchical clustering and *tf∗idf* statistics as the distance function indicates, that there are significant differences in the pre- and post-release weekly behaviors of contributors in selected time-windows.

While hierarchical clustering is a good sanity test for the data exploration, the performance of K-means clustering is much more valuable [14]. I performed k-means on the symbolic representation of data using *tf∗idf* statistics and Euclidean distance. The algorithm converged after two iterations separating pre- and post-release dictionaries with a single mismatch for the Android 2.0 pre-release.

By using centroids of two resulting clusters as a basis for pre- and post-release patterns I tested the classifier on the rest of Android kernel-OMAP releases. The classifier was able to successfully classify more than 81% of pre- and post-release behaviors (Table III). When applied to the similar project - kernel-TEGRA - it demonstrated the error rate less than 15%.

The classifier demonstrated a weak, almost random performance on other sub-projects, such as user-interface related projects and e-mail client. However, when re-trained on the platform-external-bluetooth-bluez project, its performance on other bluetooth-related sub-projects, such as platform-external-bluetooth-glib, platform-external-bluetooth-hcidump, and platform-system-bluetooth, recovered to 20% miss-classification.

Table III: Pre- and post-release development patterns classification results for kernel-OMAP.

Release	Classification
1.6-pre	misclassified
1.6-post	OK
2.2-pre	OK
2.2-post	OK
1.1-pre	OK
1.1-post	OK
2.3-pre	OK
2.3-post	OK

Release	Classification
beta -pre	OK
beta-post	OK
2.0.1-pre	OK
2.0.1-post	misclassified
2.1-pre	OK
2.1-post	OK
2.2.1-pre	OK
2.2.1-post	misclassified

B. Contribution

To the best of my knowledge, this work is the first attempt to study the applicability of symbolic aggregate approximation and term frequency–inverse document frequency weight statistics to the mining of software process artifacts. This methodology has a number of advantages. First of all, SAX facilitates significant reduction of the large complexity (dimensionality and noise) of temporal artifacts and opens the door to application of a plethora of string search and text-mining algorithms. In addition, the $tf{*}idf$ statistics provides an efficient mechanism for discrimination of the signal by ranking symbolic data while focusing on dissimilarity. Finally, the third component I have used - the relational database - facilitates efficient data slicing, indexing, and retrieval.

As an example of a possible data-mining workflow demonstrating the resolving power and correctness of the approach, I presented a case study of building a classifier for pre- and post-release recurrent behaviors. Whereas this classifier demonstrates a good performance within the project it was trained on with less than 20% miss-classification, it has less than 15% miss-classification rate in similar Android OS kernel sub-projects.

V. Discussion

The presented approach and workflow employs two novel techniques in order to discover and rank recurrent behaviors from software process artifact trails. While the approach demonstrates satisfactory performance, the interpretation of the captured behaviors requires more work. The discovered behavioral patterns are organized in Table II by their occurrence: the first three columns belong to the post-release time-window, the four next columns belong to pre-release time-window, while the rest are the behavioral patterns observed in both. The bottom row of the table contains plots visualizing examples of the raw-data streams corresponding to symbolic behavioral patterns. By the visual examination of these examples, it appears that during pre-release most of the added lines within a week fall on the Monday and Tuesday, whereas during post-release time, most of the lines are added during the end of the week and the week-end. While an explanation of these findings requires an additional study to be made, one of the interpretations of such behavior could be based on the contributors employment profile. For example, if the coding activity of developers paid to work on Android (thus mostly commit during working days) has fallen below the activity of developers working on their own volition (who commit mostly off business hours); which, in turn, could be a consequence of removing of a pre-release code-freeze, or that the paid developers switched in post release period to design, documentation, or bug-fixing activities.

VI. Acknowledgment

I thank to Philip Johnson for his time, useful discussions, and comments.

References

[1] A. Hindle, M. W. Godfrey, and R. C. Holt, "Mining recurrent activities: Fourier analysis of change events," in *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, May 2009, pp. 295–298. [Online]. Available: http://dx.doi.org/10.1109/ICSE-COMPANION.2009.5071005

[2] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 153–162. [Online]. Available: http://dx.doi.org/10.1145/1985441.1985464

[3] L. McLeod and S. G. MacDonell, "Factors that affect software systems development project outcomes: A survey of research," *ACM Comput. Surv.*, vol. 43, no. 4, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1145/1978802.1978803

[4] A. E. Hassan, "The road ahead for mining software repositories," in *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE, Sep. 2008, pp. 48–57. [Online]. Available: http://dx.doi.org/10.1109/FOSM.2008.4659248

[5] G. Antoniol, V. F. Rollo, and G. Venturi, "Linear predictive coding and cepstrum coefficients for mining time variant information from software repositories," in *Proceedings of the 2005 international workshop on Mining software repositories*, ser. MSR '05, vol. 30, no. 4. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1145/1082983.1083156

[6] A. Hindle, M. W. Godfrey, and R. C. Holt, "Release Pattern Discovery via Partitioning: Methodology and Case Study," in *Proceedings of the 29th International Conference on Software Engineering Workshops*. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: http://dx.doi.org/10.1109/ICSEW.2007.181

[7] J. Lin, E. Keogh, L. Wei, and S. Lonardi, "Experiencing SAX: a novel symbolic representation of time series," *Data Mining and Knowledge Discovery*, vol. 15, no. 2, pp. 107–144, Oct. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10618-007-0064-z

[8] T. Roelleke and J. Wang, "TF-IDF uncovered: a study of theories and probabilities," in *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, ser. SIGIR '08. New York, NY, USA: ACM, 2008, pp. 435–442. [Online]. Available: http://dx.doi.org/10.1145/1390334.1390409

[9] B. K. Yi and C. Faloutsos, "Fast Time Sequence Indexing for Arbitrary Lp Norms," in *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 385–394. [Online]. Available: http://portal.acm.org/citation.cfm?id=645926.671689

[10] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases," *Knowledge and Information Systems*, vol. 3, no. 3, pp. 263–286, Aug. 2001. [Online]. Available: http://dx.doi.org/10.1007/PL00011669

[11] J. Lin, E. Keogh, S. Lonardi, and B. Chiu, "A symbolic representation of time series, with implications for streaming algorithms," in *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, ser. DMKD '03. New York, NY, USA: ACM, 2003, pp. 2–11. [Online]. Available: http://dx.doi.org/10.1145/882082.882086

[12] R. J. Larsen and M. L. Marx, *An Introduction to Mathematical Statistics and Its Applications (3rd Edition)*, 3rd ed. Prentice Hall, Jan. 2000. [Online]. Available: http://www.worldcat.org/isbn/0139223037

[13] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1542–1552, Aug. 2008. [Online]. Available: http://dx.doi.org/10.1145/1454159.1454226

[14] *Initialization of Iterative Refinement Clustering Algorithms*, 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.3469