Software Process Extraction and Identification

Abram Hindle, Michael W. Godfrey, Richard C. Holt
University of Waterloo
{ahindle,migod,holt}@cs.uwaterloo.ca

ABSTRACT

Industrial software is planned, managed, and created using a variety of approaches: sometimes a formal Software Development Life-cycle (SDLC) model is used, sometimes an approach that is less formal but has clearly identifiable stages is employed, and sometimes little if any discernible process is followed. In this paper, we extract and correlate the software development process with behaviour and data found within a project's source control repository. We do so by analyzing fine grained changes, revisions, and aggregations of revisions, so that we can correlate them with the stage of the software development process that the project was in at the time they were made. To label intervals of revisions we use machine learning and artificial intelligence techniques including N-Nearest Neighbours classifiers, Markov models, Hidden Markov Models, Markov Decision Processes and Partially Observable Markov Decision Processes. These techniques initially learn the stages from annotated data and then classify unknown data. We describe how to pose the problem using these tools and we evaluate their effectiveness on several case studies.

1. INTRODUCTION

The software engineering discipline has attempted to formalize how to build software. Some of the earliest efforts in formalizing how to build software were the iteration based models and the Waterfall model [15]. If developers had stages of software development, much like the assembly line, perhaps the gains seen in productivity from the industrial revolution could be translated into gains in the productivity of software developers.

Software development processes or software development life cycle (SDLC) models developed from strict and static processes like the Waterfall Model [15] into more iterative processes like the Rational Unified Process or eXtreme programming.

SDLC models are often composed of iterations and stages. Stages are usually part of iterations, but in a SDLC model like the waterfall model, stages are of a larger granularity supposedly that is because there is only one real iteration. Iteration based models will often have a sequence of stages per each iteration. They could also have multiple types of iterations which different stages associated with each.

SDLC models can include stages such as requirements elicitation, design, maintenance, feature freeze, integration, etc. SDLC models relate directly to software evolution. SDLC models attempt to tell you how software should be made. Software evolution tells us how it was made. Our problem is given the repository of a project how do we reverse-engineer the process from this data?

Software evolution is the study of how software changes over time [13]. Software evolution research often delves into *software trails* [3], that is the evidence left behind by developers when they make changes to a project. Trails consist of data from CVS logs, their patches, and the associated developer mailing lists. These trails can be rebuilt in order to study the changes.

Software evolution also concerns the measurement of change. If one is to comprehend a system that changes over time one should consider how the measurements of the system change. A common measure is the lines of code (LOC) or clean lines of code (no comments or extra whitespace). Some software evolution metrics measure systems before and after a change, as well as measuring change itself [10, 12, 4].

Using these kinds of metrics, Lehman formalized laws of software evolution such as square root growth of modules in a system over time [11]. More recent research on Open Source software, such as Godfrey's analysis of the Linux Kernel [6], seemed to contradict some of Lehman's laws.

In the field of process discovery, Cook has described frameworks for event based process data analysis [2]. Multiple methods for correlation and measurement are used on processes. The metrics used included string distances between a process model and the actual process data, work-flow modeling, and Petri-nets. Also, Cook [1] discusses grammar inference, neural networks and Markov models as applied to process discovery.

Our work extends that of Cook's in a novel direction. Instead of attemping to insert sensors and monitors into the development process, we analyze the data available to us and attempt to determine to which development stage it belonged. This is done by analyzing fine-grained changes to source control systems (SCS), such as CVS [16].

Process extraction and correlation are important because they tell us what kind of changes the programmers made to the system and as well as contexts of their changes. If we

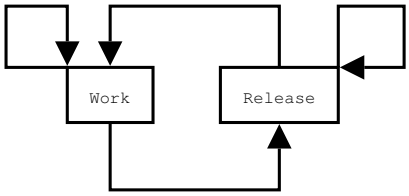

Figure 1: Transition Model of *work* and *release*

knew which stage a revision belonged to we could aid in the querying of changes. This information would be useful to new maintainers or new people joining a project. As well it would be useful as a post-mortem tool for projects. We note that Cook has used this approach to verify if programmers were following the model.

The techniques we employ include machine learning techniques[9], Markov Models[1] and Hidden Markov Models (HMM)[14]. Hollmen et al.[8] discuss how to discretize time series data for HMMs. This is important as if we are to use a HMM, it will be necessary to discretize the continuous data from CVS.

2. PROBLEM

Thus our problem is, "*Given a CVS transaction, can we reliably determine the stage of software development that is associated with it?*" Our challenge is that we want to label intervals of revisions found in a SCS with the stage in their SDLC model that they belong to. We also wish to investigate if our techniques are reliable enough to suggest the kind of process model that has been used, and to compare this to what model the developers and managers believe they have followed.

Figure 2 shows the time-series of number of MRs and Number of Revisions for the Evolution project. It also shows how we wish to annotate the revisions with their associated stages.

2.1 Assumptions and Goals

We will assume that we have some CVS data that has been annotated, or we could have trained on a similar project. The CVS data consists of transactions (groups of revisions) and revisions. A revision is a patch to a previous version of a file. Lines refer to lines in a text file. Revisions have data and it is easy to compute metric values such as:

- Lines Added, Removed or Modified: The numbers of lines added, removed or modified by the patch.
- Entities Added, Removed or Modified: The number of entities (global variables, functions or macros) added, removed or modified by the patch.
- File: The pathname, filename and file-type of the file which the revision is being applied to.
- Previous Revisions: The revisions that were previous to this revision. These are the previous versions that were built upon to produce this revision
- Next Revisions: The revisions that built upon this revision.

Transactions are aggregations of revisions. Usually when an author commits a change to multiple files this is called a transaction. Transactions have obvious associated metrics such as: Number of Files Changed; Number of Revisions; Aggregation of lines added, removed or modified; Aggregation of entities added, removed or modified; Previous Revisions; and Next Revisions.

We have decided not to use entity-based metrics for several reasons. First, such metrics typically require access to and knowledge of the full source code (at least of the containing file) for a detailed analysis; for the sake of simplicity and generality, we wished to remain programming language independent. Second, CVS does not require the new code to be syntactically correct, so there is no guarantee that the new code would even be processable by a parser-like analyzer.

Aggregation can be used to compute values such as summation, average, standard deviation, etc. Aggregation can also be used on the attributes themselves by aggregating observations into intervals such as days or windows.

Since the behavior we initially noticed was on an aggregation of changes over time (per day) we therefore decided to use daily aggregations of transactions. This aggregation would include the number of authors, number of revisions and the number of lines changed per day.

3. ANALYSIS

In this section we analyze how to apply and how appropriate various machine learning and artificial intelligence techniques are with respect to our problem of labeling the software development stage of a transaction. We show how to pose and model our problem with these tools as well how to initialize and train these tools with annotated data.

There are various possible analysis methods that all have different costs and benefits. Some techniques require more work by the end user whereas others can be trained on external data or learn as they go. Some techniques require discretization of values, some can work with continuous values. Some techniques that used discretized values can be altered to handle continuous values. Discretization is difficult because one has to understand the data. One should be aware of the distributions of the attribute values being discretized. Intelligent manipulation of these distributions of values would allow us to produce discretizations that can highlight unique values, whereas naive discretizations could ignore the subtlety of the data.

The cost trade-offs all depend on the size of the repository being annotated and the size of available annotated data. Because we plan to evaluate these techniques we are going to use *release* and *work* stages. The *release* stage refers to the two week period that surrounds a project's release. These generally correspond to last minute bug fixes, code freezes, and after-release emergency bug fixes. The *work* stage refers to the feature additions and maintenance that occur between releases. See figure 1 for an diagram of the transition model of the *work/ release* process. These stages were chosen because we are able to evaluate process stage identification on a large number of projects, as we have access to their release histories. Also, it is straightforward to build a large corpus just from release dates and an extracted CVS repository.

3.1 Machine Learning

We used various machine learning algorithms on a feature vector of data. Unfortunately the results were often incon-

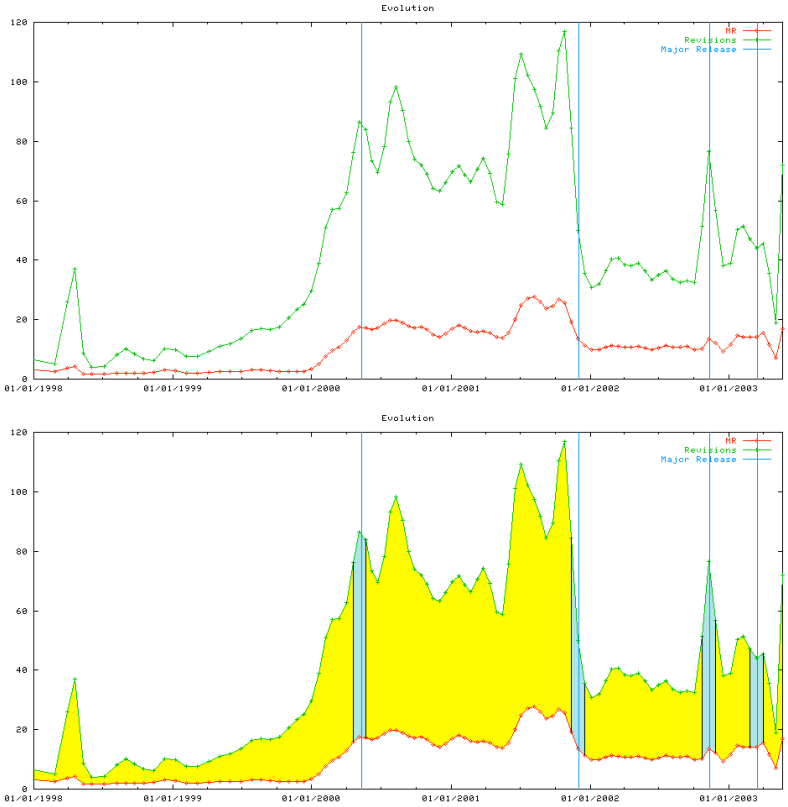

Figure 2: Evolution before and after stage annotation. The stages are yellow *work* and blue for *release*.

clusive or misleading. We posed our labeling problem as a classification problem: *Can the learner classify to which stage a change (an observation) belongs?*

For machine learning, we used feature vectors that described a day of development. The feature vectors consisted of the number of Modification Requests (MRs), the number of revisions, the number of authors, the average number of revisions per author, and the lines-added and lines-removed.

More formally, our machine learning framework consists of $< D, O, l >$:

- $D : d_0, d_1, ..., d_m$ - D is the set of stages (in our case $D = \{release, work\}$). d_i is a stage in the SDLC.
- $O : o_0, o_1, ..., o_n$ - O is the set of vectors of observations. o_i is a feature vector describing a change (in our case an aggregation of changes in a day).
- $l(o_i)$ is a learner trained on annotated data which outputs a vector of size m indicating the probability that a stage $d_i \in D$ is associated with the observation o_i

The machine learning techniques can be tested and trained in multiple ways: train off the beginning part of time-series; train off random selection of days from time-series; train off other project; train off multiple other projects. We used multiple learners such as Naive Bayes, Bayesian Networks, Gaussian Classifiers, 1-Nearest Neighbors and Perceptrons in our evaluation in section 4.1.

One benefit of the lack of temporality is that if a user wishes to annotate only the future or subsets of data they already know about, the training will work, where as for HMMs or Markov Models it is probably best to train on continuous intervals.

3.2 Markov Models

Markov Models permit a certain level of temporality. Unfortunately this can lead to an explosion in the number of states depending on the order of the Markov model. Our Markov Model is a first order model which consists of states that are a combination of the stage and the observation. The training data is simply pre-labeled states, this allows the calculation of transition probabilities and allows us to decide the probability of state transitions given new data.

Observations of various variables (the features described in section 3.1) can be combined as a vector. Since the values of the vectors are integers (or in some case real numbers) we need to discretize the values to reduce the number of states. We used the number of MRs, number of authors, number of revisions, and the sum of the number of lines added and removed. Because many of these are unbounded we normalize the values by taking the $\lfloor log_2(1+x) \rfloor$ where x is the positive value. We chose logarithms because we wanted the larger revisions to be noticed while still reducing the number of states.

More formally, our Markov Model consists of:

$$< D, O, S, Pr, Pr_{start} >$$

- $D : d_0, d_1, ..., d_m$ - D is the set of stages (in our case $D = \{release, work\}$); d_i is a stage in the SDLC.
- $O : o_0, o_1, ..., o_n$ - O is the set of vectors of discretized observations; o_i is a feature vector describing a change

PostgreSQL					
Classifier	Correct	*release* Precision	*release* Recall	*work* Precision	*work* Recall
BayesNet	24%	0.243	1.000	0.000	0.000
Naive Bayes	61%	0.229	0.246	0.753	0.734
$K*$	60%	0.177	0.169	0.738	0.749
1-NN	57%	0.255	0.400	0.765	0.626
Perceptron	75%	0.000	0.000	0.757	1.000

Table 1: Results of various machine learning classifiers on classifying *release* and *work* stages in PostgreSQL aggregate MRs (10% classify, 90% train)

.

Evolution					
Classifier	Correct	Release Precision	Release Recall	Work Precision	Work Recall
BayesNet	45%	0.000	0.000	0.457	1.000
Naive Bayes	46%	0.524	0.125	0.454	0.865
$K*$	49%	0.539	0.566	0.453	0.527
1-NN	53%	0.561	0.682	0.491	0.365
Perceptron	45%	0.000	0.000	0.457	1.000

Table 2: Results of various machine learning classifiers on classifying *release* and *work* stages in Evolution aggregate MRs (10% classify, 90% train)

.

(in our case an aggregation of changes in a day).

- $S : s_0, s_1, ...s_{m*n}$ - S is a set of states s_i; each state is associated with one observation in O and one stage in D thus $S = O \times D$.
- $Pr(s_t|s_{t-1})$ is the probability that state s_t occurs after state s_{t-1}; this can be represented as transition matrix of probabilities.
- $Pr_{start}(s_0)$ is the starting probability that the first stage is s_0; this is the prior.

If we don't discretize, we have to devise a function for $Pr(s_t|s_{t-1})$, and this can be difficult. We build up $Pr(s_t|s_{t-1})$ from the data that has been observed and labeled with a state.

After training on the annotated data we can run our algorithm. For each new observation we simply choose a state associated with that observation which has the highest transition probability. We then consider that observation to be labeled because that state is associated with a stage. We could try to use a forward algorithm to find the path of maximum probability through the Markov Model, although that is more suited to a Hidden Markov Model (see section 3.3).

A problem with a Markov model is that if the stages are not evenly distributed often it will be more difficult to choose a transition from states of one stage to states of another stage.

3.2.1 Alternative Markov Models

Alternative Markov models could be investigated:

Kth-Order Markov Model: We could use more previous elements to determine the probability of a Markov chain. This might cause the stage transition problem where one stage occurs more than another and will be very conservative, usually by avoiding the transition to another stage. Alternatively we could use a true forward algorithm.

Continuous State Markov Models: Instead of having set states, we simply use the feature vector to describe the states and interpolate probabilities and distributions from the data we have trained on. This would be quite similar to many of the machine learning algorithms.

Multiple Markov Models: By combining multiple Markov models one could assign stages to new changes by majority voting. Multiple Markov models would also simplify the state-space. If there was a state which was in the observations that wasn't in the training data, a single Markov model would suggest it had 0 probability of transitioning there, where as multiple Markov models, each only caring about 1 attribute, could possibly cover all the possible states and use prior evidence to determine the stage.

3.3 Hidden Markov Models

The Hidden Markov Model (HMM)[14] is very appropriate for our problem. We don't necessarily know the stage in which the project was in at the time, nor was this recorded, thus the stage is partially observable. The HMM is quite appropriate because the states (which are partially observable) will be the stage label, and stage labels are partially observable.

HMMs usually require discretized values. Our observations are discretized in the same fashion as the Markov model described in section 3.2.

More formally, our Hidden Markov Model consists of:

$$< D, O, Pr, Pr_{start}, Pr_{em} >$$

- $D : D_0, d_1, ..., d_m$ - D is the set of stages (in our case $D = \{release, work\}$); d_i is a stage in the SDLC.
- $O : o_0, o_1, ..., o_n$ - O is the set of vectors of discretized observations; o_i is a feature vector describing a change (in our case an aggregation of a day of changes).
- $S : s_0, s_1, ...s_{m*n}$ - S is a set of states s_i; each state is

associated with one observation in O and one stage in D, thus $S = D \times O$

- $Pr(s_t|s_{t-1})$ is the probability that state s_t occurs after state s_{t-1}; this is be represented as transition matrix of probabilities.
- $Pr_{start}(s_0)$ is the starting probability that the first stage is d_i; this is the prior.
- $Pr_{em}(o_j|d_i)$ is the emission probability that the stage is d_i given observation o_j

The Viterbi algorithm will give us the most likely explanation for the observations we provide, as it iterates through the observations using dynamic programming to discover the path that is worth the most.

Our transition matrix is the observed probabilities of changing to a different stage. Our emission probability is a matrix that determines given an observation o_j, what the probability is that the current stage is d_i. We train off of data which has been annotated with stages from D; thus, training is not on-line.

The Viterbi algorithm we use is slightly modified to accommodate the lack of precision of floating point numbers. If a state does not exist or has not been visited we give it a probability of $1/n$ where n is the number of observations. Also we often had to segment observations into blocks of 100 observations so that the precision would hold. The HMM should be more accurate than the MM.

3.3.1 Alternative Hidden Markov Models

Alternative Hidden Markov models could be investigated:

Continuous State Hidden Markov Models: The HMM could be extended to be less discrete and exist in multi-dimensional space. This could lead to issues where inappropriately interpolated probabilities are used. A continuous state HMM would probably employ some sort of sampling to approximate probabilities.

Multiple Hidden Markov Models: We could combine multiple HMMs, all trained on different attributes, and integrate them with a majority voting system for determining the stage based on the observation. This would avoid cases where unknown states exist because not all the combinations of values of each domain are in the training data.

3.4 Markov Decision Processes

Markov Decision Processes differ from MMs and HMMs in that they allow for actions. Thus, instead of hidden states we can permit actions, and these actions can be chosen based on reward. Our actions will be labeling the current observation. We want to emphasize smooth and consistent intervals of one stage. Maybe we don't want to change stages until we're sure the next few observations will belong to that stage.

Thus the benefits of MDPs include reward-based actions, possible on-line learning (Q-Learning) and the ability (and design) to handle more complex problems without need for discretization due to the reward function. The lack of discretization is definitely an advantage. This allows various attributes to be used unchanged and avoids error introduced due to discretization.

Unfortunately MDPs assume the state is fully observable. To get around this we made the action handle the partially observability of the labeling.

More formally, our Markov Decision Process consist of:

$$< D, O, A, Pr(d_t|a_{t-1}, o_{t-1}), R(o_t), \lambda, h >$$

- $D : D_0, d_1, ..., d_m$ - D is the set of stages (in our case $D = \{release, work\}$); d_i is a stage in the SDLC, and the stages are actions.
- O is the set of possible feature vectors describing changes; observations are states.
- $Pr(o_t|d_{t-1}, o_{t-1})$ is the transition model; it determines given an action and the current state, what is the probability of o_t occurring.
- $R(o_t)$ is the reward model; given an observation, what is its utility; it provides incremental rewards.
- λ is the discount factor; it helps us recognize that there are only finite observations.
- h is the Horizon; it is the number of observations.

For the purpose of the reward function we will want to have trained on data and associated data points with labels. Optionally we can use a distance function similar to Nearest Neighbor (NN) algorithm in the reward function if we want to determine which stage an observation might associated with.

The difficulty of using an MDP is that we have to define our reward function. Our reward function must positively reward consistent labeling of intervals, negatively reward not changing stages enough (once or more) and negatively reward labeling in which the label contradicts our distance function. Let $w = 1$ be the number of unique stages labeled is 1, let $w = 0$ if the number of unique stage labeled is greater than 1. Let x equal the number of past actions with the same label. Let t be the time (the count of elements). Let $d(o_t)$ be the normalized distance of o_t to the given label. Thus our reward function could feasibly be: $R(o_t) = -1 * w + x/t + -d(o_t)$

We calculate the optimal finite policy or near optimal finite policy, then we run through all the observations with that policy and get the labeling based on the actions chosen. Bellman's equation would allow us to find the optimal labeling although it could take exponential time. The main gain of the MDP over the HMM is the reward function. The reward function allows us to better specify want we want to see rather than expecting it to exist.

3.5 Partially Observable Markov Decision Processes

Partially Observable Markov Decision Processes (POMDP) are like MDPs except they allow for Partially Observable States. That is, the POMDP is more like a HMM and a MDP combined.

One advantage is we can have a disconnection between the detected state and our action. We want to label the observation and state pair. Our action might label it differently than the partially observed state thus allowing for more continuous intervals. Also if we assume there is error in our labeling we can use our reward function to attempt to alleviate the error.

More formally, our Markov Decision Process consists of:

$$< D, O, A, Pr(d_t|a_{t-1}, o_{t-1}), Pr(o_t|d_t), R(d_t), \lambda, h >$$

- $D : d_0, d_1, ..., d_m$ is the set of stages (in our case $D = \{release, work\}$). d_i is a stage in the SDLC. The stages are states.
- O is the set of observations, the set of possible feature vectors describing changes.
- A is the set of actions; actions in our POMDP will be what stage we label the current observation and state.
- $Pr(d_t|a_{t-1}, o_{t-1})$ is the transition model; it determines given an action and the current state, what is the probability of d_t occurring.
- $Pr(o_t|d_t)$ is the observational model; given a state d_t what is the probability of observation o_t occurring.
- $R(d_t)$ is the reward model, it determines the utility given an observation; it provides incremental rewards.
- λ is the discount factor; it helps us recognize that there are only finite observations.
- h is the Horizon, this is the number of steps to run (the number of observations).

Unfortunately calculating optimal policies for POMDPs is in exponential time with respect to the observations; for larger projects such as Mozilla that have 500,000 or more transactions, calculating the optimal policy of a POMDP is infeasible. Instead we'd probably have to use Sondik and Monahan's algorithm, point based value iteration or other approximations of POMDPs.

We could use a similar reward function from the MDP section 3.4. Although instead of Nearest Neighbors we could actually use $Pr(o_t|d_t)$ which has already been provided.

We suspect that the combination of rewards and partially observable state would lead to more accurate results. One possible benefit of a POMDP is that once an optimal policy is calculated we can communicate and reuse that policy without the need to recompute it.

4. EVALUATION

For our evaluation we implemented and used three different tools: Machine Learning techniques, Markov Models, and Hidden Markov Models. For each method we used the same data set (although states would have to be generated from the data). The data was daily aggregations of number of authors, number of revisions and aggregations of the number of lines changed. The data was extracted from various Open Source projects (table 3 contains a list of them). No time data, such as the date, was used. We used daily aggregation was because there was a lot of data and we had previously visually identified features that we thought would only appear in the aggregated time series data. The stages used to aggregate the data are *release* and *work*. These were partially chosen because it was easy to automate the annotation of the time series.

Each test consisted of the project's daily aggregations where the first 90% of observations were used as annotated training data and the last 10% of observations were used as the test data (which were annotated so we could test the accuracy).

The results will show the number of observations classified, the percentage of observations correctly classified and the precision and recall of *release* and *work* observations. One should probably pay more attention to the precision and recall of the *release* stage because the higher that is, the more effective the classifier is, as there are fewer *release* observations than *work* observations. Although poor *work* precision and recall statistics are bad too.

None of the algorithms used random values thus rerunning them would result in the same values. The data was extracted from the CVS repositories of many open source projects such as PostgreSQL, Evolution, Mozilla, etc. The *softChange* [5] system was used to extract the CVS data, the SCQL [7] tool was used to query the data, and scripts aggregated these values into daily aggregations.

4.1 Machine Learning

For the machine learning tests we used WEKA[9] and ran multiple classifiers on the data. The classifiers we ran were: Bayesian Network, Naive Bayesian Learner, K-Star ($K*$), 1-Nearest Neighbor and a Voter Perceptron (a neural network).

Tables 1 and 2 shows the results of various learners operating on PostgreSQL and Evolution data. We can see that none of them are very precise for *release* (partially because *release* is a smaller group than *work*). Out of all the machine learners Nearest Neighbor seems to do the best.

The Perceptron is more accurate than the other learners because it never choose a non-*work* label; it has poor *release* precision and recall. Thus to really evaluate these results one needs to look at the precision and recall for both *work* and *release*.

The machine learning results are disappointing, most learners have a hard time with identifying *release* observations. The BayesNet and Perceptron did the worst while Nearest Neighbor did the best. We suspect that the lack of temporality in the learning and training as well as the lack of entity based features hampered this identification.

4.2 Markov Model

We used the same kind of dataset as before, except instead of testing different learners, we tested against many different projects (more than just PostgreSQL and Evolution). In section 3.2 we discussed how we discretized the data into states and how we would predict which state a new observation would belong to.

The 90% training data was used to create a Markov Model that the other 10% would be run on. The results, in table 3, varied significantly between projects. Some projects were classified poorly (their precision or recall was 0.00) while others did better. Looking at the state data many of the Markov Model were too conservative to switch stages to *release*. This is partially because releases show up less often and thus transitions to them are less probable. Seeing as many of the results are quite erroneous one should be wary of using the Markov Model to determine the labels. 1-NN was more accurate for both PostgreSQL and Evolution.

One of the difficulties in implementing the Markov Model was that sometimes the observed state had never been seen before so some default error probability had to be used. We suggested in section 3.2 multiple Markov models might be able to get around this.

4.3 Hidden Markov Model

Our Hidden Markov Model experiments were done in the

same way as our Markov Model experiments in section 4.2. The big differences are that observations are states, that the Viterbi algorithm is used to determine the labellings and that the states are partially observable (labeled by the Viterbi).

A difficulty encountered with the implementation of the HMM and Viterbi algorithm was that sometimes the lack of precision of the floating point values would cause issues with the calculations. We had to break observations up into blocks and run the Viterbi on those blocks. We also allowed for unknown transitions to be worth a minimum probability.

The results, described in table 4, show that the HMM has better performance than the MM. There are many more cases where the precision and recall of *release* is reasonably higher than both the MM results and the Machine Learning approach. We suspect some of this is to do with the temporality and partial observability of HMMs. The HMM doesn't seem to beat Nearest Neighbor for Evolution or PostgreSQL for *release* precision and recall. Although it does better than NN for *work* precision and recall.

5. CONCLUSIONS

We posed the problem of labeling the stage of software development a CVS transaction was in. We then proposed and evaluated the use of various machine learning and AI techniques, such as 1-NN Classifiers, MMs, HMMs, MDPs and POMDPs. We evaluated all of these except for the MDPs and POMDPs. Our evaluation showed that 1-NN performed better than HMMs, while HMMs performed better than our MMs. We were able to achieve a reasonable labeling accuracy of 53% to 65% with our HMM and our 1-NN.

We recommend further investigation into MDPs and POMDPs as well as the alternative MMs and HMM models we suggested. We can even incorporate the successful 1-NN into our MDPs and POMDPs. We suspect the HMM can still fare well with more effort.

5.1 Future Work and Open Questions

We only tested two stages. More kinds of stages should be tested. We also should have evaluated different granularities of aggregations. Our discretizations were somewhat arbitrary and should have been based on extracted distributions. We should have a better comparison of 1-NN and HMMs over more projects. We should test K-NN for $K > 1$.

Overall what these results seem to suggest we might get better results with code, entity and architectural based metrics such as methods added, removed or modified or changes in architectural dependencies.

6. REFERENCES

[1] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 73–82, New York, NY, USA, 1995. ACM Press.

[2] J. E. Cook and A. L. Wolf. Balboa: A framework for event-based process data analysis. In *Proc. of the 5th International Conference on the Software Process*, pages 99–110, June 1998.

[3] D. M. German. Using software trails to rebuild the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, to appear, 2004.

[4] D. M. German and A. Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Proceedings of 11th International Software Metrics Symposium (Metrics 2005)*, 2005. To be presented.

[5] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softchange. In *Proceedings SEKE 2004 The 16th Internation Conference on Software Engineering and Knowledge Engineering*, pages 336–341, 3420 Main St. Skokie IL 60076, USA, June 2004. Knowledge Systems Institute.

[6] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of International Conference on Software Maintenance*, pages 131–142, 2000.

[7] A. Hindle and D. German. Scql: A formal model and a query language for source control repositories. In *MSR 2005: International Workshop on Mining Software Repositories*, May 2005.

[8] J. Hollmn and V. Tresp. Hidden markov model for metric and event-based data. In *10th European Signal Processing Conference, EUSIPCO 2000*, pages 737–740, September 2000.

[9] E. F. Ian H.Witten. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor, October 1999.

[10] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics, 2002.

[11] M. M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.

[12] T. Mens and S. Demeyer. Evolution metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001. ACM Press.

[13] A. Mockus, R. T. Fielding, and J. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[14] L. R. Rabiner and B. H. Juang. An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 4–15, January 1986.

[15] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–339. IEEE Computer Society Press, Mar. 1987.

[16] T. Zimmermann and P. Weisgerber. Preprocessing CVS data for fine-grained analysis. In *1st International Workshop on Mining Software Repositories*, May 2004.

Project	#days Labeled	% Correct	*release* Precision	*release* Recall	*work* Precision	*work* Recall
apache13	181	48%	0.1235	0.3030	0.7700	0.5203
distcc	38	55%	0.0000	0.0000	0.5526	1.0000
embperl	52	46%	0.0000	0.0000	0.5455	0.7500
evolution	162	33%	0.4074	0.5000	0.1852	0.1351
gnumeric	176	43%	0.4375	1.0000	0.0000	0.0000
httpd2	220	21%	0.2136	1.0000	0.0000	0.0000
kdevelop	226	22%	0.2257	1.0000	0.0000	0.0000
metacity	83	85%	0.8974	0.9459	0.2000	0.1111
mozilla	156	41%	0.0000	0.0000	0.4103	1.0000
openssl	160	88%	0.0000	0.0000	0.8875	1.0000
orbit2	75	45%	0.5714	0.1860	0.4262	0.8125
postgresql	268	61%	0.2466	0.2769	0.7590	0.7291
ppp	36	27%	0.2778	1.0000	0.0000	0.0000
rsync	67	86%	0.0000	0.0000	0.8657	1.0000
samba	239	80%	0.0000	0.0000	0.8033	1.0000
tomcat	86	82%	0.0000	0.0000	0.8256	1.0000
xerces	129	90%	0.0000	0.0000	0.9070	1.0000
Average	138	56%	0.2000	0.3654	0.4787	0.5917

Table 3: Results of Markov Model labeling of Projects (10% classify, 90% train)

Project	#days Labeled	% Correct	*release* Precision	*release* Recall	*work* Precision	*work* Recall
apache13	181	84%	0.5789	0.3333	0.8642	0.9459
distcc	38	60%	0.5385	0.8235	0.7500	0.4286
embperl	52	57%	0.5000	0.2000	0.6364	0.8750
evolution	162	53%	0.5733	0.4886	0.4828	0.5676
gnumeric	176	60%	0.5606	0.4805	0.6364	0.7071
httpd2	220	66%	0.0000	0.0000	0.7552	0.8382
kdevelop	226	65%	0.1915	0.1765	0.7654	0.7829
metacity	83	32%	0.8276	0.3243	0.0741	0.4444
mozilla	156	40%	0.5152	0.1848	0.3902	0.7500
openssl	160	75%	0.0000	0.0000	0.8686	0.8380
orbit2	75	46%	0.6667	0.0465	0.4306	0.9688
postgresql	268	75%	0.0000	0.0000	0.7575	1.0000
ppp	36	75%	0.6000	0.3000	0.7742	0.9231
rsync	67	77%	0.2500	0.4444	0.9020	0.7931
samba	239	71%	0.2222	0.1277	0.8066	0.8906
tomcat	86	82%	0.0000	0.0000	0.8256	1.0000
xerces	129	87%	0.2500	0.1667	0.9174	0.9487
Average	138	65%	0.3691	0.2410	0.6845	0.8060

Table 4: Results of Viterbi Algorithm executing on HMMs of Projects (10% classify, 90% train)