Measuring software development productivity: a machine learning approach

Jean Hélie Ian Wright Albert Ziegler
{jean,wright,albert}@semmle.com
Semmle Inc.

We apply machine learning to version control data to measure software development productivity. Our models measure both the quantity and quality of produced code. Quantity is defined by a model that predicts the labor hours supplied by the 'standard coder' to make any code change, and quality is defined by a model that predicts the distribution of different kinds of problems identified by a static code analysis tool.

1 Why measure software development productivity?

The productivity of an industrial process is the ratio of its outputs to inputs. Software developers supply their time (inputs) to produce useful software applications (outputs). Can we measure the productivity of software development? Such a measure would help us identify and propagate best practices by comparing productivity across teams, projects, companies and sectors; we would avoid waste and reduce costs to consumers; and we could verify whether new tools or methodologies actually increase productivity, etc. Few enjoy the thought that their productivity may be tracked; and, indeed, such metrics may be abused. But if we wish to transform software development from a craft to an engineering discipline, we have to measure our productivity. The key challenge is to measure it well. Clearly, we cannot capture the full richness of real life with a single metric, but we can measure important aspects of it, and therefore contribute additional information to human decision makers.

2 Productivity: quantity *and* quality

Existing methods rely on subjective methodologies (e.g., function point analysis [1]) or models constructed from small datasets (e.g., COCOMO [5]) that often require manual data collection. Instead we aim to automatically measure productivity leveraging big datasets produced as a side-effect of software engineering (i.e., version control).

The main function of a software developer is to produce source code. Developers, of course, produce other outputs (such as designs, plans, code reviews etc.), but these are typically produced as a side-effect. We aim, therefore, to measure the *coding productivity* of developers who submit code changes to a version control repository. The 'amount of output' produced by a coder has two aspects: highly productive developers not only produce 'more' code change but also changes of 'higher' quality. So we (somehow) must measure both the quantity and quality of coding output. We breakdown coding productivity into *coding output*, which measures the 'amount of effort' required to produce code changes, and *code quality*, which measures the 'amount of quality' embodied in source code. We therefore need datasets of (i) code changes, and the effort required to make those changes, and (ii) the quality of source code.

Presented at the 'Machine Learning for Programming' Workshop affiliated to the 30th International Conference on Computer Aided Verification, 18–19th July 2018, Oxford UK.

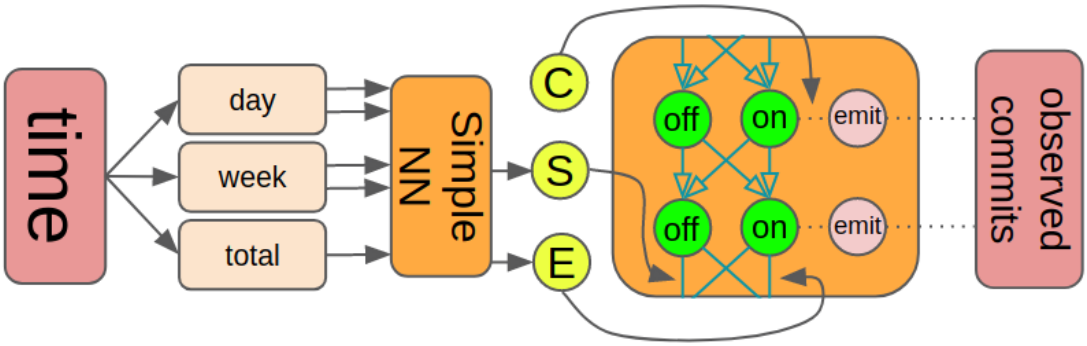


Figure 1: The NN computes time dependent transition probabilities S and E for the HMM. The HMM's state probabilities are fitted to the observed sequence of commits with the forward-backward algorithm. The NN and emission probability C are trained to maximize the likelihood of the sequence.

3 lgtm.com: a big dataset of the quantity and quality of coding output

Semmle's LGTM ('looks good to me') analyses over 10M commits submitted by $\approx$ 300K developers to $\approx$ 56K open source projects (see lgtm.com). Each commit produces a new source code 'snapshot' that LGTM analyzes using QL [2], a declarative, object-oriented logic programming language for querying complex, potentially recursive data structures encoded in a relational data model. QL treats 'code as data' and is particularly well suited for static code analysis and code queries.

LGTM executes hundreds of QL queries on each snapshot, where each query identifies a potential problem in the source code – from simple mistakes such as uninitialized variables, to complex errors, such as taint analysis via routes from data sources to sinks. The queries, developed over multiple years by language experts, encode domain knowledge about code quality. Each query is a quality probe that may generate one or more 'alerts'. LGTM therefore supplies rich code quality data. In addition, every commit is time-stamped. We now explain how we construct productivity metrics from this data.

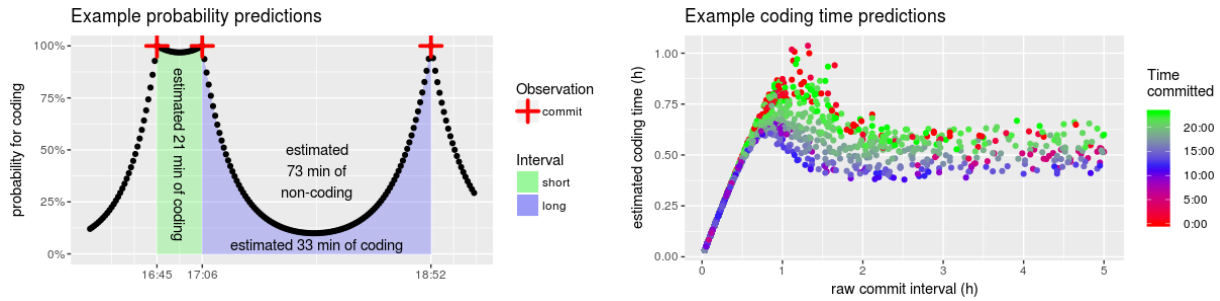
4 Coding output

Two different code changes contain the same 'amount of output' if they typically take the same labor time to produce. Consider two consecutive commits, A and B, by the same author. The commit interval is the time difference between them (e.g., 10 minutes, or 3 days). In general, the labor time that produced the code change from A to B is less than or equal to the commit interval. Short intervals (e.g., 15 minutes) may coincide with coding time but longer commit intervals (e.g., 8 days) include coding breaks (especially in open source development). Can we extract coding time information from commit intervals?

4.1 Coding time: a Neural Hidden Markov model

We train a neural hidden Markov model[1] to predict the probability an individual developer is currently coding for each 1 minute of their recent history (e.g., $\approx$ 2 years). We then use the model to estimate the expected actual coding time between two successive commits.

We assume two states: coding and non-coding. We observe the timestamped sequence of commit events: a commit can only happen when coding (with a trained probability C per minute). Coding developers end coding, within the next minute, with probability $E(t)$, and non-coding developers start



(a) The HMM assigns a coding probability to each minute. Short commit intervals often indicate uninterrupted coding, whereas longer intervals suggest coding interruptions. We estimate time spent coding as the expectation value, which can be computed by taking the area under the probability curve.

(b) Short intervals (e.g., $<$ 1 hour) are often spent coding nonstop. So, counter-intuitively, they provide evidence for more coding time than medium intervals (e.g., 2 - 5 hours), which probably include coding breaks. This model picks up that the developer has fewer interruptions in the early morning.

Figure 2: Using the neural HMM to estimate coding times from commit intervals.

coding with probability $S(t)$. In contrast to classical hidden Markov models, probabilities E and S vary over time to account for habits (e.g., preponderance for evening coding) and schedule (e.g., only coding on week days). A simple neural network (NN) with 5 inputs – the sine and cosine of the angle of an imaginary dial on a day-long and week-long clock, and normed overall time – supplies the probability values. These features can encode daily or weekly patterns that may shift over time. The NN is then trained by back propagating likelihood gradients through the HMM part (see Figure 1).

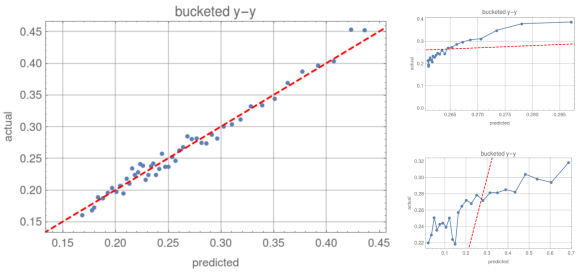
The neural HMM is a flexible solution to a general problem: for any process that alternates between active and inactive phases, it infers the probability of being active at any time from an observed rhythm of potentially infrequent 'life signs' (in this case commit events). The neural HMM can discover night-time coders, or developers that code only on weekdays, or weekends etc. The upshot is a model that maps commit intervals to estimates of coding time (see Figure 2).

4.2 The standard coder: a Deep Mixture Density Network

We use the coding time models, described in the previous section, to create datasets that associate code changes with estimated coding times. Next, we train a model that, given any code change, predicts the coding time typically required to produce it. Such a model would then represent the coding time profile of the 'average' or representative 'standard coder'.

How do we represent code changes, and what kind of regression model should we apply? On our datasets, a variant of bag-of-words features plus a deep mixture density network (MDN) yields the best performance.

A single commit may affect multiple files. For each language we construct a 'token dictionary' consisting of separators (e.g., `.`, `(`, `)`, `+`, `*`, etc.), keywords (e.g., `if`, `return`, `while`, `class`, etc.) and top n most frequent words (e.g., popular variable names). For each touched file we compute the diff (lines either deleted and inserted) as per standard utilities (e.g., Unix's `diff` command). We assume code deletion also requires coding effort. So we concatenate all diffs, both insertions and deletions, to form a composite change string. We compute a bag-of-words feature vector, which represents the quantity of token turnover, or code change, introduced by the commit.



(a) Each data point is binned hold-out examples with similar predicted coding times in hours (x-axis). The y-axis is the mean of the actual coding times. The MDN captures the 'average' coding time of developers, whereas linear regression (inset top) and random forest (inset bottom) fail on this dataset.

```
See http://conda.pydata.org/docs/config.html for details on all the options
"""""
from conda.gateways.adapters.localfs import LocalFSAdapter
from conda.gateways.adapters.s3 import S3Adapter
from requests.adapters import HTTPAdapter, BaseAdapter
            raise ProxyError(dals("""""Could not find a proxy for %r. See
            http://conda.pydata.org/docs/html#configure-conda-for-use-behind-a-proxy-server
            for more information on how to configure proxies.""""" % proxy_scheme))
```

```
from ..base.constants import CONDA_HOMEPAGE_URL
See %s/docs/config.html for details on all the options
""""" % CONDA_HOMEPAGE_URL
from requests.adapters import BaseAdapter, HTTPAdapter
from .base.constants import CONDA_HOMEPAGE_URL
from .gateways.adapters.localfs import LocalFSAdapter
from .gateways.adapters.s3 import S3Adapter
            raise ProxyError(dals("""""
            Could not find a proxy for %r. See
            %s/docs/html#configure-conda-for-use-behind-a-proxy-server
            for more information on how to configure proxies.
            """""" % (proxy_scheme, CONDA_HOMEPAGE_URL))
```

(b) An author deleted 8 and added 12 lines across 9 files of Python code. The commit interval was 3.96 minutes with an estimated coding time of 3.70 minutes. The MDN model predicts this work is equivalent to 10.6 minutes of the standard coder's labor time.

Figure 3: Validating the standard coder.

The relationship between code change and coding time is irreducibly noisy due to fundamental properties of the generative process (e.g., due to skill differences between developers and random noise due to work interruptions). In consequence, the regression model must be robust to noise. Plus, we want to bound the uncertainty of our coding time predictions. A MDN [4] fulfills both criteria. Deep MDNs are multi-layered neural networks that predict the parameters of a mixture distribution (i.e., predict a random, rather than deterministic, target), where the parameters are a nonlinear function of input features. We train with a likelihood loss. At prediction time we may compute the mixture mean, or use the full distribution.

4.3 Model validation

The MDN learns distributions of coding times that correspond to similar kinds of code changes. To validate the model we need a ground truth. We know, from our experience of software development, that very short commit intervals (e.g. < 1 hour) typically correspond to sustained periods of actual coding.[2] We therefore select, as our ground truth, holdout data with commit intervals < 1 hour.[3]

We trained a MDN model on 347K examples of Python code changes (with commit intervals replaced by coding times). The MDN predicts a mixture of 10 Gaussians using 140 bag-of-token features and 3 hidden layers of size 28.[4] We neither expect, nor require, that our model predict individual examples with high accuracy. Rather, we aim to predict the 'average' coding time of the population of developers. So, for model validation, we collect holdout examples into bins of similar predicted coding times, and then compare against the bin means of actual coding times. Figure 3a illustrates that the MDN model accurately captures the behavior of the standard coder. The upshot is a measure of the 'amount of output' represented by code change in units of labor hours supplied by the standard coder (see figure 3b).

5 Code quality: an ensemble of nonlinear quantile regression models

Code quality has many facets. For example, run-time bugs imply poor quality, but the source code of an error-free program may nonetheless contain high technical debt, and therefore be difficult and costly

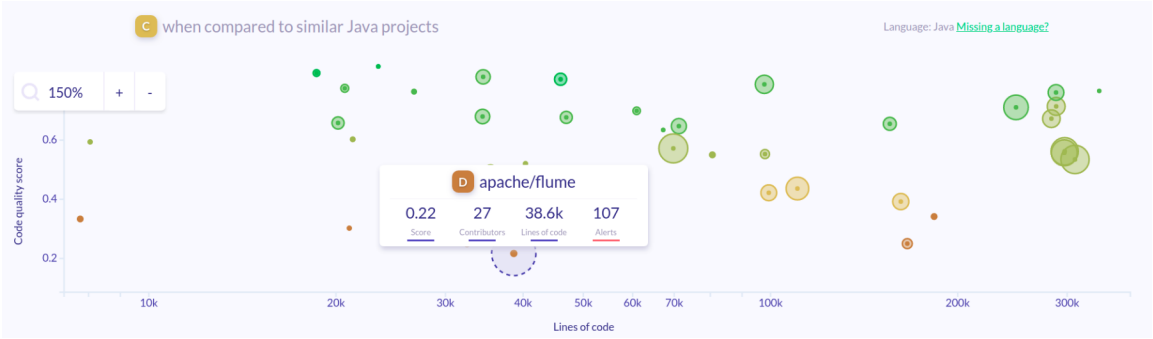


Figure 4: A code quality comparison chart from `lgtm.com`. The x-axis is LOC and the y-axis is quality. Each bubble is an open-source Java project. The highlighted project, `apache/flume`, has 107 alerts, a quality score of 0.22 and final grade `D`.

to maintain. LGTM alerts detect a wide range of potential problems in source code, including problems indicative of run-time bugs[5] and technical debt.[6] We therefore use the number and kinds of alerts in source code as a proxy for code quality. Intuitively, the more alerts triggered by some code, the worse its quality. However, code quality must also consider both the properties of the code that triggers alerts and their types: e.g., a small project with a single, serious alert has worse quality than a large project with a few less serious alerts.

LGTM classifies alerts into different categories (e.g. errors or warnings, with differing severities and precision).[7]. For every alert category for every language we fit a non-linear quantile regression model that predicts the alert distribution for a project with a given size in lines-of-code (LOC).[8] We train with data from $>$ 50K LGTM projects. We constrain the models to output monotonously increasing functions of LOC (to enforce the simple heuristic that alerts increase with LOC).[9] The models feed into a percentile-rank model that outputs an estimate of the cumulative distribution function $F_{l,i}$ of the expected number of alerts in category i given l LOC. The 'quality score' for each category is $s_i = 1 - F_{l,i}(n_i)$ where l is the project's LOC and n_i is the actual number of alerts in category i. The interpretation of s_i is straightforward: it measures how well the project performs (0 = worst, 0.5 = as expected, 1 = best) when compared to the expected distribution of alerts predicted by the model. The final code quality score is a weighted sum of the s_i, where the weights are defined by simple heuristics based on the statistical distribution of alert categories in LGTM data; e.g. more severe alerts weigh more, rarer alerts weigh more etc.

We also assign a quality grade (A+ best, E worst) by comparing a project's quality score to a reference set. `A+` means the project is best amongst its peers, whereas `E` means the project has room to improve. The upshot is a quantitative measure of the quality of source code that summarizes the results of hundreds of independent, and often complex, quality probes (see figure 4).

6 Conclusion

We construct community-based standards of software productivity from high volumes of empirical data on the quantity and quality of code changes. We apply machine learning techniques to construct models of *coding output* and *code quality* that function as standards of measurement. The models reduce complex, structural code changes to quantitative measures of the quantity and quality of software developers' output. We plan to make our productivity metrics available on `lgtm.com` for testing and feedback.

Notes

[1]Since the 90s, efforts have been made to combine the flexible power of neural networks with the sequential approach of HMMs. [3] used NN to generate the HMMs outputs, while recently [6] obtained excellent results by obtaining HMM parameters from NN outputs for parts-of-speech tagging.

[2]The coding time model confirms this expectation since, for all developers, it discovers a near one-one relation between coding times and short commit intervals (see Figure 2b).

[3]And, to avoid any possible interference from the coding time model, we simply set coding times equal to raw commit intervals.

[4]Note that the holdout data is not drawn from the same distribution as the training set. In consequence, at prediction time, we truncate the mixture distribution so the entire probability mass is contained within the interval $[0, 1]$ hour. The final prediction is the mean of the truncated distribution.

[5]See `lgtm.com/blog/testing_and_alerts_3`.

[6]See `lgtm.com/help/lgtm/query-tags`.

[7]This information is available at `github.com/lgtmhq/lgtm-queries`

[8]Future extensions will take into account other properties of source code (e.g. complexity) that also affect code quality.

[9]This is strongly supported by the data with Kendall τ_b correlation coefficients between LOC and overall number of alerts of 0.69, 0.72 and 0.73 for Javascript, Python and Java projects respectively.

References

[1] A. J. Albrecht (1979): *Measuring Application Development Productivity*. In: *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, IBM Corporation, Monterey, California, pp. 83–92.

[2] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones & Max Schäfer (2016): *QL: Object-oriented Queries on Relational Data*. In Shriram Krishnamurthi & Benjamin S. Lerner, editors: *30th European Conference on Object-Oriented Programming (ECOOP 2016), Leibniz International Proceedings in Informatics (LIPIcs)* 56, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 2:1–2:25, doi:10.4230/LIPIcs.ECOOP.2016.2. Available at `http://drops.dagstuhl.de/opus/volltexte/2016/6096`.

[3] Y. Bengio, R. De Mori, G. Flammia & R. Kompe (1991): *Global optimization of a neural network-hidden Markov model hybrid*. In: *Neural Networks, IJCNN-91-Seattle International Joint Conference on Neural Networks* 2, IEEE, Seattle, WA, USA, doi:10.1109/IJCNN.1991.155435.

[4] Christopher Bishop (1995): *Neural Networks for Pattern Recognition*. Oxford University Press, Inc.

[5] Barry Boehm, Crhis Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifier & Bert Steece (2000): *Software Cost Estimation with COCOMO II (with CD-ROM)*. Prentice-Hall, Englewood Cliffs, NJ.

[6] Ke Tran, Yonatan Bisk, Ashish Vaswani, Daniel Marcu & Kevin Knight (2016): *Unsupervised Neural Hidden Markov Models*. In: *Proceedings of the Workshop on Structured Prediction for NLP*, Association for Computational Linguistics, Austin, TX, pp. 63–71.