UNDER REVIEW

RESCIENCE C

Reproduction / Pattern Recognition

[Rp] Reproducing ”Typographical Features for Scene Text Recognition”

Jerod Weinman[1]
[1]Grinnell College, Grinnell, Iowa, USA

Edited by
(Editor)

Reviewed by
(Reviewer 1)
(Reviewer 2)

Received
–

Published
–

DOI
–

Abstract This article reports on the largely successful reproduction of the author's decade-old conference paper. The original 2010 paper demonstrated that character recognition performance could be improved on difficult problems of scene text recognition by leveraging font-specific correlations between character identity and width. The work relied on a sizeable array of languages, tools, and libraries. The computation proceeded in three major phases: data synthesis, training statistical models on artificial image data and actual text data, and finally finetuning and running a parser with the trained models applied to the test images. Using the learned models stored from the original paper's experiments, the original parser code successfully reproduced the results exactly. The training code required minor changes to run with current host environments and libraries. Although the updated experimental results are not identical, they follow the same general trends, indicating a successfully repeated experiment.
A replication of [1] .

1 Introduction

As readers of this journal know, reproducibility of computational experiments presents an important but surmountable challenge. This article reports on a peer-reviewed, archived conference paper by the author originally published in August 2010 [1]. That paper constituted the author's first work to be fully conceived, developed, submitted, and published as a full-time faculty member following doctoral training and the transition to a new institution two years before. Although some of the raw benchmark data used in the paper dates to several years prior (2003), most of the experimental data for training the system dates to 2009, with the final reported experiments dating to January 2010 for the submission that same month.
It is possible to reconstruct the arc of the work because of the author's use of a homespun experimental data repository created in 2009 precisely so that the details of experiments could not only be retraced, but ideally reproduced if necessary [2]. The motivation for that data repository was precisely because of the author's own earlier inability to accurately trace and ultimately reproduce some dissertation results [3] after transitioning institutions and computational environments. It therefore seems quite fitting that the decade-old paper tested here represents the "first fruits" of that system and the first whose long-term reproducibility has been assessed.
The remainder of the article gives some additional details on the nature of the author's local archival system for experiments and data (Section 2), then describes more about the work under examination including the details of the software and computations in Section 3. Section 4 presents the work involved in bringing the experimental system back up to speed. A variety of ablation tests in the experimental chain attempt to identify or eliminate possible sources of variation in the reproduced results; Section 5 concludes with updated experimental results that match precisely in some cases, while the trend of relative performance in other cases still align with the original results.

2 Data Repository for Reproducible Research

The author's data repository for understanding reproducible research closely resembles the distributed cached computations framework of Peng and Eckel [4]; additional details are in Weinman [2]. Computations are cached in immutable objects called collections, and subsequent collections may utilize the results of other (previous) collections through a large dependency graph. Each collection is housed within a namespace hierarchy that helps identify and understand its role; the descriptive name of each collection also features a timestamp. For example, collection *eB* in Figure 1 is named `experiments/text/ngrams/bigrams/tied_nums_intracase_L1_validation-20090708075734`. Version numbers of source controlled code are documented within a collection for reproducibility (and in current practice, the source code and appropriate revision are checked out of the source repository as part of the collection build process). For simplicity, the repository resides in the file system, managed by a few scripts and adherence to the practice; nothing "forces" a collection to be write-only. Use of the native file system makes both the generating code and resulting data relatively transparent, highly accessible, and easily portable, all key qualities for a framework deeply entangled with everyday work.

The key properties of the data collections are (i) that they may only access experimental software through well-defined source code repositories (system software, programming environments, or external libraries have not been precisely tracked), and (ii) they may only access other experimental data through the data dependency graph. In particular, scripts running in collections may not directly access other collections' source code nor any part of the host file system except through the dependencies. The only source code "native" to a collection is a small set of scripts that invoke the library code (a copy of which is typically "checked out" or "cloned" into the collection) on the data of interest (a depenency from another collection), together with any parameters needed. A simple `Makefile` is required so that invoking an argument-free `make` command generates the data after checking (and compiling, if necessary) any additional source code. In this way, the data collections make it fairly easy to understand how a particular computation or result was achieved. Other utilities make it easy to copy collections (sans data) for alteration (e.g., parameter tweaking) in a way that traces the provenance (which assists bug tracking/recovery) and access the dependencies (a simple text file) in a variety of programming languages.

The collections are divided into three categories: *raw*, *processed*, and *experiments*. Raw collections house curated or manually annotated data that is not programmatically generated. Associated code might be stored with these collections, such as download scripts or interactive prompts for label acquisition. They may also have dependencies, as in the case when the collection stores the annotation of another raw data set. The processed collections are for relatively straightforward programmatic transformations of data, such as denoising, format conversion, feature extraction, etc. Finally, the experimental collections house the most interesting data.

Notably, none of the restrictions are programmatically enforced; the relatively lightweight framework operates by convention. However, because it uses the file system, collections are easily transportable across host machines. Experimental computations have been deployed to remote high-performance computing clusters on which the author has copied a subset of the data repository, so that the experimental dependencies can be found.

3 Review of the Experiments

In an application of machine-learning for pattern recognition, the original paper demonstrated that character recognition performance could be improved on difficult prob-

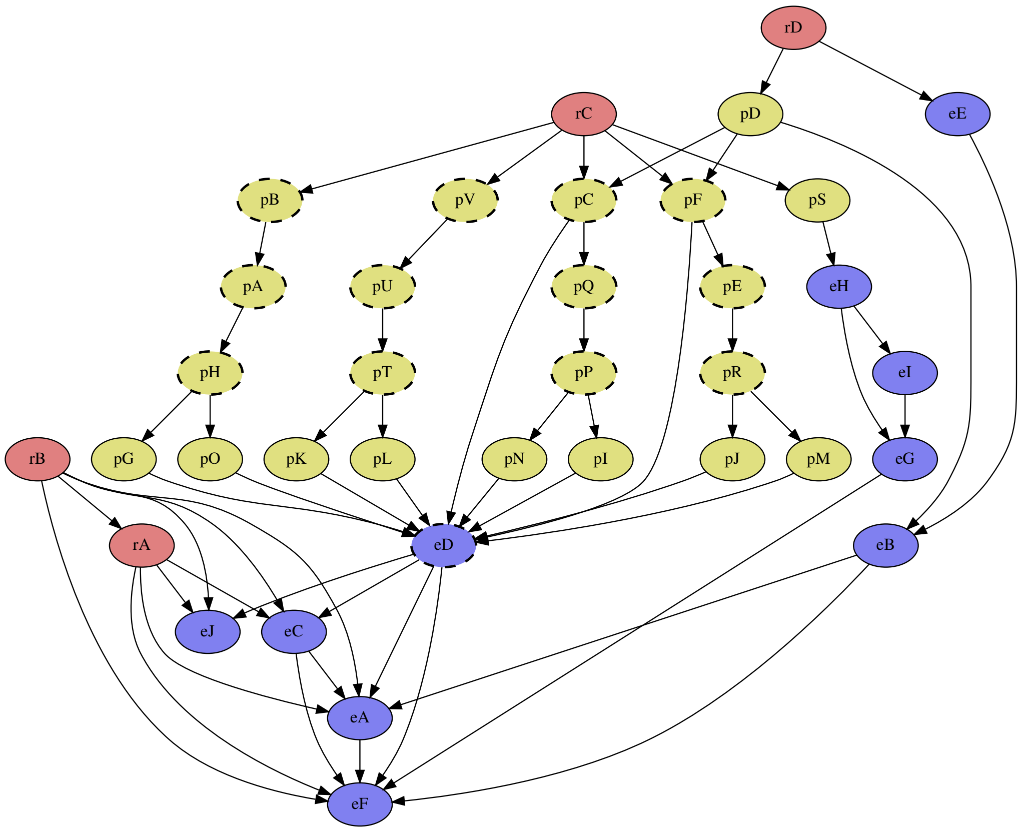

Figure 1. Dependency graph of the cached computations involved in reproducing the original paper [1]. Each node (arbitrarily lettered) represents a data collection (cf. Section 2), with red indicating *raw* collection ("*r*"), yellow *processed* ("*p*") and blue *experiments* ("*e*"). All results are generally replicated, but dashed borders indicate a failure to reproduce identically. Detailed descriptions of each collection appear in Table 3.

lems of scene text recognition by leveraging font-specific correlations between character identity and width [1]. For example, although the width of the character "e" varies widely across fonts, knowing (or hypothesizing) that information strongly informs beliefs about the width of other characters in the same font.
The paper required a deep pipeline of data and a wide array of tools and programming environments. This section provides a somewhat archeological and forensic view of everything that needed to be run and briefly indicates the dependencies on external libraries.

3.1 Experimental Structure

Figure 1 illustrates the dependencency graph of all data repository collections (cf. Section 2) utilized in the paper. This section sketches the basic purpose and tool dependencies of each collection, with further details in Table 3.

Raw Collections — Raw collections *rB* and *rA* represent the benchmark data used for evaluation; *rB* contains the cropped images of street and storefront signs recognized by the system, while *rA* contains the ground truth character annotations (the dependency arising from the interactive script used to acquire the annotations). Although the entire data repository itself was not created until 2009, the sign data collections are artificially timestamped to their original dates of creation in 2003 and 2005, respectively. Raw collection *rC* contains synthetic images of individual characters rendered in over 1800 fonts, which forms the basis of training data used for learning the appearance-based module of the recognizer as well as the intra-font character width correlations. Finally, *rD* contains the raw ASCII text of 85 English-language books downloaded from Project Gutenberg, used for training the bigram-based language module.

Processed Collections — The four isomorphic processing chains in the center of the graph represent rendered and processed variations of characters made to appear as they would in photographed storefronts, rather than the clean glyphs of a font library (*rC*). The recognizer was trained to discriminate not only among characters and numbers (the dependency chain rooted at *pF*), but also among inter-word spaces (*pV*) and intra-word gaps between characters (*pB*), so synthetic exemplars from each of these categories are rendered, in addition to horizontally stretched and scaled versions of the original characters (*pC*). These root collections render the text with occasional randomly-placed borders. Collection *pD* simply counts the bigram frequencies in the books of *rD*; chains *pF* and *pC* utilize these bigram statistics to synthesize plausible neighboring characters that appear alongside the target character in the image. Collection *pS* measures the width of each character in each font for later use in the width bigram module that represents the paper's scientific and technical contribution.
The roots of these chains—*pB, pV, pF, pC,* and *pS*—all use MATLAB as the computational engine with a small dependency on the author's support library (stored in a version control system) of MATLAB tools. The bigrams of *pD* were counted using a simple standard C program housed and compiled within the collection.
Deeper in the data processing chains, additional image transformations are applied before extracting image features used by the recognizer. Collections *pA, pU, pE,* and *pQ* add random contrasts, brightness, and linear bias fields to the images, while *pH, pT, pR,* and *pP* downsample the images by 1/4. These collections also use MATLAB with minimal assistance from a local support library. Collections *pO, pK, pM,* and *pI* binarize the resulting images in the same way the test data will be; run within MATLAB, the Niblack binarization routine from the author's library uses a `mex` implementation for speed (the `mex` interface allows programs written in C to be compiled and used from within MATLAB). The sibling collections *pG, pL, pJ,* and *pN* apply a wavelet transform to the images that produces the representation used by the recognition module. The implementation of this transform—the steerable pyramid [5], which localizes image edges, orientations, and scales—is a `mex` implementation by its original creator (Simoncelli) dating to 1996; although updated versions now exist on GitHub, the version in the author's local library (used in these experiments) dates to 28 March 2001.

Experiment Collections — In practice, distinguishing processed from experiment collections is not always as clear as the nomenclature might indicate. For instance, collection *eE* is simply a ten-way partition of the books in *rD*, used for cross-validation experiments; it should probably be considered processed. Likewise, collection *eH* captures the intra-font character width bigrams statistics in a frequency table (and plots), making it analogous in role to *pD* (character bigram counts). Experimental collection *eI* fits the parameters of a regularized exponential probability model over these character width bigrams, using cross-validation to determine the amount of statistical regularization (which limits overfitting). Collection *eG* adds spaces as one of the categories for the width bigram model. Collection *eB* does the same thing, but for actual character identity bigrams of the usual sort used in language modeling (with ten-fold cross validation based on the partition in *eE*).
These "learned model" collections (*eE, eI,* and *eG*) effectively train what is variously called a maximum entropy (MaxEnt) classifier [6] or multinomial logistic regression [7] (in this case with no feature inputs; only class bias weights are learned). The MATLAB implementation of the classifier and the L-BFGS optimizer needed [8] to train it were

Table 1. Main code libraries used in the experiments of Figure 1. Statistics generated by `cloc`* and count only actual code lines, omitting blank lines and comments.

Name	Purpose	Language	Files	Lines of Code
VIDI	Viterbi parsers	Java	10	1,565
	Data processing	MATLAB	160	7,075
L-BFGS	Optimization	MATLAB	2	416
PyrTools [5]	Image features	MATLAB	83	2,926
		C/`mex`	15	2,143
MaxEnt	Discriminative classifier	MATLAB	35	1,907
CUDA MaxEnt	GPU-accelerated training	MATLAB	11	834
		C/CUDA	202	33,749

* `cloc v1.85 , https://github.com/AlDanial/cloc`

created by the author (housed in the local, version-controlled support library), and have been publicly shared under the GPL since 2010.
The hub experimental collection *eD* trains the character recognition model, a discriminative MaxEnt classifier. The implementation is a combination of the MATLAB MaxEnt class and L-BFGS code above, as well as a CUDA-backed `mex` kernel to rapidly accelerate the computations needed for each stage of the batch gradient descent algorithm. CUDA is the language framework supporting general purpose computation on a GPU (graphics processing unit). The so-called `cudamaxent` software was published (GPLv3) and described in a publication by Weinman, Lidaka, and Aggarwal in 2010 [9].
The first result published in the paper is finally computed in collection *eJ*, where the test images' binarizations and steerable pyramid features are calculated the on the fly, put through the MaxEnt model, and finally parsed by a Java-based Viterbi (dynamic programming) algorithm. Intermediate collection *eC* fine tunes the relative weights of a module that selectively penalizes certain overlaps or gaps among the parse segments as scored within the Viterbi calculations. Collections *eA* and *eF* create the last two results in the paper, *eA* adding the character identity bigram score within the Viterbi parser, and *eF* adding the character width bigram score on top of that. With the addition of the width-based model, the dynamic programming table for the semi-Markov model for is larger and the experiment uses a different Java subclass within the same hierarchy.

3.2 Code and Data Footprint

In addition to highlighting the structure of inter-experimental *data* dependencies (cf. Section 3.1 and Figure 1), it is useful to comprehend the magnitude of code dependencies and the resulting data footprints.
Table 1 enumerates the primary external code dependencies used in the experiments. These do not include the obvious need for the built-in MATLAB tools and toolboxes, which included the Image Processing, Statistics (now Statistics and Machine Learning), Optimization, and Parallel Computing Toolboxes. All of the other local software libraries listed in the table were managed through an institutional Subversion source control system. Revision numbers for each experiment were manually recorded by the author in the text manifest for each collection. Bug fixes and improvements to the software have of course been recorded since the original experiments were performed for the publication. Most of the author's software—L-BFGS, MaxEnt, and CUDA MaxEnt libraries—was publicly distributed with those improvements on the author's home page at around the time or shortly after the publication; all were published under the GPLv3 free software license. Simoncelli's PyrTools was obtained under the MIT license. The author's VIDI tool was not previously published.
Table 3 also quantifies the foot print of the collections themselves. On average, each collection relies on approximately three scripts. For example, this might be the master,

argument-free "run" script, which invokes a parameter-driven sub-script, which may itself be aided by some instance-level helper. Although there is substantial variation among individual collections, the total amount of stored data involved in producing the work is 127 GiB, even though the "raw" data is only 320 MiB.
MATLAB was "chosen" as the primary environment for hosting the experiments because the author had, in a sense, "grown up" working in MATLAB a decade earlier in undergraduate image processing and computer vision courses. By the time these experiments were crafted, the author had developed quite a library of operational tools through his dissertation work [3]. Moreover, the model in the paper was an extension of that earlier dissertation work, so several bits of that code would be leveraged (although, as observed above, the version control throughout the dissertation work was not as strong or prevalent; RCS *was* used sparingly).
Java became involved because a robust and stateful class/object capability was necessary to support the parser efficiently with a broad standard library. At the time, the author had used Java for over a decade (since its 1.0 days) and was happened to be far more familiar with it than an alternative such as C++. Moreover, MATLAB had just began to support native Java class integration with its environment.

4 Reproducing the Work

This section documents the reproduction experience on an updated computing platform, including minor changes to the code that were necessary.

4.1 Compute Environments

Original Environment — The host computer system on which the original experiments were run is still operational in the author's research laboratory. However, like the proverbial Ship of Thesus, enough of its hardware and software have been upgraded in the last decade to question whether it is truly the same.

Hardware The CPUs are dual quad-core 64-bit Intel Xeons (E5520), with hyperthreading thta present to the OS as a total of 16 compute cores. The system has 48GiB of host (CPU) memory. The GPU used for the original experiments was an NVIDIA Tesla C1060, featuring 240 compute cores and 4GiB of memory.

Software At the time, the host operating system was Ubuntu 8.04 LTS, though precisely which version is not known. This distribution and version was intentionally chosen for the ease of installation, maintenance, stability, and most notably the duration of support. Operating system upgrades can be fraught with difficulty and the preference was to avoid them for as long as possible. None of the distribution package details are known and there was no regular practice of updating them. Hence, important details such as the precise C compiler used cannot be known (though it would have been whatever the default of the distribution was). In order to use CUDA and the NVIDIA GPU, both the CUDA library and the NVIDIA driver were necessary. The versions installed by the author on the platform and used in the experiments are not known. However, considering the timing of the experiments it was most likely CUDA 2.2 or 2.3. A `Makefile` path indicates MATLAB R2009a compiled the CUDA MaxEnt code; given that the system administrator at the time updated software infrequently, it is almost certain the same version was used for all the experiments in the original paper as well. A now deprecated `nvmex` tool was used to bridge the CUDA/MATLAB divide; the `Makefile` pointed to a user's home directory where the file still exists. Although the version of Java is not known precisely, the the Ant `build.xml` file for VIDI indicates Java version 1.5 was to be used.

Reproduction Environment — Restoring the original machine to its state at the time of the original experiments is within the realm of possibility. The author retains all the original hardware (the GPU included), and archived versions of the OS and library software are available for download. However, rather than pursue that rather rewardless task, it seemed preferable to embark upon a replication study to verify the stability of standard tools (i.e., bash, C, Java, Matlab, and CUDA) .

Hardware An entirely different host was used. Its dual 14-core Intel Xeon (E5-2695) CPUs present as having 56 cores, with 512 GiB of available host memory. The system houses several GPUs, but experiments relied on only one Titan RTX GPU, with 23.6 GiB of RAM and 4,608 cores.

Software The host operating system is Ubuntu 18.04.3 LTS. MATLAB is R2018a. CUDA 10.1 sits atop NVIDIA drivers with version 418.67. The default C compiler is gcc 7.4.0. The Java is OpenJDK 1.8.0_222.

4.2 Software Preparation

The Subversion server hosted by the author's institution was taken down about one year before embarking upon reproducing this work. Although local copies of the most recent code exist, specific versions of the code could not be checked out without the Subversion server. Fortunately, the system administrator was able to quickly restore a server and source repositories to their previous working order. This proved pivotal to the precise reproduction of at least some experimental results.
MATLAB has been maintained on the system, including all the requisite toolboxes. However, licensing could have been an issue, and it is unlikely Octave would have produced similar, let alone identical results.
Compiling and running the C code required for calculating text bigrams was effortless, as it was all standard C. Checking out and building the Java files for the author's VIDI library also worked perfectly (there were no external library dependencies). Whereas the author had already adapted the Steerable Pyramid toolbox to the Ubuntu ecosystem, copies were on hand that were compiled most recently in 2016 using MATLAB R2015a. However, these were easily recompiled and verified to produce identical results (cf. Section 5.3). In addition, the author's C/`mex` code for an efficient 2D convolutional box filter compiled without trouble for these experiments with MATLAB R2018a.
By far the most challenging part of the process was in resurrecting the CUDA MaxEnt code and restoring it to a runnable state. One expects fairly significant differences to emerge over the evolution from version 2.x to 10.1 of the CUDA library. Surprisingly, to get the code compiled required only modest changes to the test harness and virtually none in the core computational code. The undergraduate students who primarily authored the code wrote a plethora of unit tests, accompanied by Matlab scripts that generate the underlying data used for regression tests against native Matlab routines. (This accounts for the very large number of files listed for the CUDA MaxEnt library in Table 1.) Without the `nvmex` script to unite the CUDA compilation and Matlab-object linking, some tinkering with the `Makefile` was required. The resulting strategy was to invoke `nvcc` (the NVIDIA compiler for CUDA code) to create the appropriate object files and then the MATLAB `mex` command could be used to link everything into a mex file for invocation from within the MATLAB environment. Before the build could complete, however, the latest version of CUnit, the requisite unit test framework, needed to be installed (easily accomplished with the `apt-get` command, which installed version 2.1-3). This version was clearly newer than the version that the software relied on, because compile errors indicated the API had changed. A quick review of the updated API indicated only few additional (unused `NULL`) parameters needed to be inserted into a test suite array. In addition, the return type for the `cudaMalloc` function

from the CUDA library may have evolved in way that now resulted errors, so four of these needed to be changed. Although several deprecation warnings were given for the use of `cudaThreadSynchronize()`, the resulting code all compiled. (A newer replacement exists, but the correctness of the code does not seem to be affected.)
In the revision of the code documented as the one used in the paper, the CUDA MaxEnt testing breaks down into 7 test suites consisting of 185 tests having a grand total of 309 assertions. Using the Titan RTX, only one of these tests failed—a `cudaMalloc` call, but the resulting kernel failed to produce any useful results in the experiments. Forging ahead, the last version of the repository was used instead (dating to just one year later, Feb 2011). It involved the same basic changes (enumerated above), passed the unit tests, and produced plausible, though slightly different results when experiments were re-run. Without a more careful examination of the failure of the original code to run effectively, it is impossible to attribute the differences in results more specifically to the different CUDA library, the slightly different MaxEnt implementation, or the different hardware.
Although somewhat tedious, recreating the 36 experimental collections (diagrammed in Figure 1) was a straightforward matter of invoking the author's `make-collection` and `copy-collection` utility scripts for each. Then, the dependency files (`DEPS`) needed to be updated for each so that it would point to any newly recreated parent collection.
Some detail work was necessary to get a few collections that utilize MATLAB scripts working with the current version of the software and the general data repository setup. Common to virtually every collection was the mechanism for intializing the MATLAB path to point to the appropriate Subversion repositories (since the system-wide version generally utilized was too recent). Two other small changes were necessary in a few places. The first updated the method for seeding the random number generator (though it is doubtful this had any useful effect, then or now), and the second updated the call for opening a thread pool to parallelize computations using the MATLAB Parallel Computing Toolbox. All in all, these were relatively small changes that required no intimate knowledge of the scripts, only an awareness of how the MATLAB API had evolved.
It was discovered that one substantive element was not achieved (as it should have been) through the collection `src/` folder, though it was properly archived in the `data/` folder. The learned character classifier was hand-modified to exclude the narrowest and widest "space" characters. Because the file was named `precog_no_extreme_spaces.mat`, recreating the effect in the reproduced experiments was fairly straight forward (load weight matrices, set two entries to `-Inf`, store modified weight matrices and save the file), but the need to do so surely would not have been obvious to anyone beside the original author.
Although recreating and rerunning the various configurations took a non-trivial amount of time, it is still somewhat remarkable that this rather complex web of inter-dependent experiments could be re-created at all.

5 Results

To isolate possible sources of variation, computations were re-run in phases, from the bottom up.

5.1 Parsing

The earliest/easiest experiments are those lowest in the computational graph (Figure 1), which were re-computed by preserving the dependencies on the original collections, thus utilizing the original data. The parsing experiments were indeed re-produced exactly. This is remarkable for a few reasons. First, there are several inner optimizations using Matlab's `fminbnd` procedure that is "based on golden section search and parabolic

Table 2. Reproduction of "Recognition results with increasing amounts of information." [1] Numbers are the recognition error rate for the $N = 1145$ characters (measured by total edit distance) over the 95 image dataset. Extra columns indicate the collections that were re-run with data from existing (2010) parent collection dependencies. The last column indicates the complete reproduction.

Colls. [Re]Run / Information	Original [1]	Parser eJ, eA, eF	+ Training $e\star$	+ Data $p\star, e\star$
Appearance	22.20	22.20	22.38	23.51
+ Char. Bigram	17.40	17.40	18.71	17.31
+ Char. and Width Bigrams	16.87	16.87	17.13	17.05

interpolation" according to the documentation;[1] the reproduced experiments produced a bitwise equivalent double-precision floating point value in all cases. Second, only the raw test images are provided to the collections; upon reproduction these images had to be fed through the `mex`-based box filter for binarization and steerable pyramid tools library for feature calculation. Because those results are not cached, it is unknown whether they are identical, but they must be close enough so that the Viterbi parse produced the same path (if perhaps not precisely the same score).

One difference in runtime behavior on the reproduced experiment is worth noting. Because the Matlab Parallel Compute toolbox automatically spawns a thread pool according to the local host configuration, four times as many threads (48) were used for accelerating the runtime of the reproduced experiments as compared to the original (16). Because the parses are entirely independent of one another (and the underlying Java and C/`mex` libraries are thread-safe), the results proved to be the same regardless of the parallel speedup.

5.2 Statistical Model Training

Next *all* of experimental results downstream of the pre-processed training data were re-run, without regenerating the training data itself. Unsurprisingly, the raw character bigram counts matched exactly. The weights of the learned character bigram model in *eB* (which rely on a Matlab-only MaxEnt model and the L-BFGS optimizer) match very closely. For weights with an absolute value mean/median/max of 0.9794/0.2233/7.3773, the absolute differences between original and new are 2.4e–7/1.7e–7/1.1e–6. The character width bigram weights in *eI* and *eG* match as well, being 3–4 orders of magnitude closer. (Results in *eH* and *eE* are identical, being mostly bookkeeping or simple pixel measurements.)

The biggest difference arises from retraining the character classifier in *eD*, which relied on the CUDA MaxEnt library. The experimental procedure for training the model incrementally relaxed a regularization penalty, designed to avoid over-fitting; a held-out validation set was used to identify the best performing model. The left panel of Figure 2 shows the training objective reducing with each step of gradient descent. As the minimization levels off and converges for each regularizing coefficient, the process then continues with a more relaxed penalty (signalled by a color shift). The optimization trace (not published in the paper, but extracted from a log file in the original 2010 collection) progresses as would be expected with rapid decreases as the penalty is eased, followed by gradual further improvements to the model weights. The updated version of the code run on the same data produces the same general curve, but generally achieves superior minima with fewer iterations. However, the validation data show that while the results *are* indeed different, they are not as different as the training curves might show. Importantly, the same model regularization amount is chosen across all runs (whether

[1] MathWorks. Find minimum of single-variable function on fixed interval. https://www.mathworks.com/help/matlab/ref/fminbnd.html

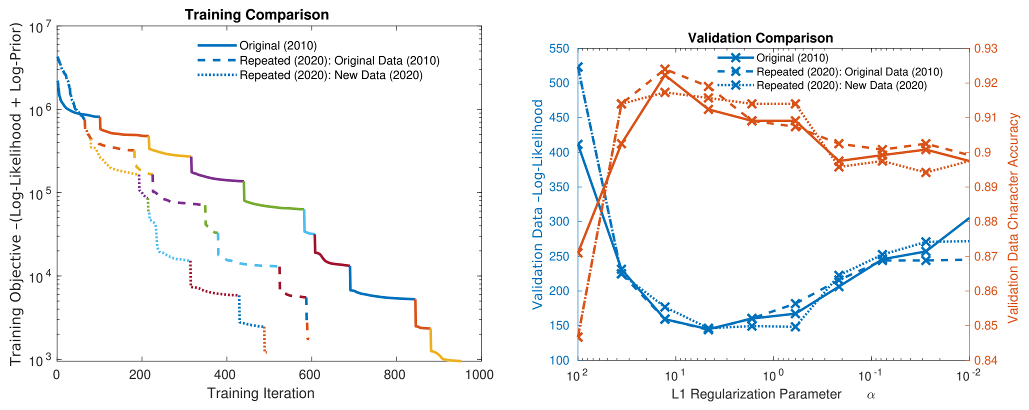

Figure 2. Training and testing results comparisons across the original run, a repetition of the training using the original data, and a second repetition using regenerated data. LEFT: Training objective function value; the color shifts with each successive relaxation of the L1 regularization parameter α. Right: Held-out validation data objective function (log-likelihood) and simple character classification accuracy. The classifier with the lowest validation data objective (L1 smoothing penalty $\alpha = 10^{2/3}$) was used in all subsequent experiments.

with the original result, a re-run on the original data, or a re-run with newly generated data). Thus, the learned models—which each have 11,315,642 parameters—appear to have fairly similar characteristics.

The end results of these intermediate experiments appear in Table 2 under column "+ Training", as in "Parser + Training" because the parser experiments were re-run with the new models as well. Although the results are not precisely the same, the relative performance among the modules tested by the original question is the same (as one would hope!).

5.3 Synthetic Data Generation

The final test was to attempt reproducing all of the original synthetic training data, and then re-run the model training and final parse experiments. Although the MATLAB code was unchanged but for the syntax of seeding the random number generator, it is not evident this was done correctly in the first place. Thus, the randomly sampled factors (neighboring characters, where synthetic sign borders appear—if at all, brightness/contrast, and noise) are all distinct. However, the code operates without fail and the general visual effects are the same in the processing chains (i.e., *pF*–*pE*–*pR*). In sum, the data appears repeatable, if not reproducible.

As an intermediate test of reproducibility, the cached image features used for training the character classifier were checked. The binarized images regenerated in *pM* from the original *pR* were identical (a result of the Niblack binarization algorithm, which uses thresholded local image statistics calculated with the recompiled `mex` box filter). The wavelet-transformed images (using the recompiled C/`mex` PyrTools) in *pJ* from the original *pR* were bitwise identical in most cases (median absolute difference of precisely zero), and the maximum absolute differences are 2ϵ, where $\epsilon = 2^{-52}$, the granularity of a double-precision floating point number. Thus, the feature calculations were fully reproducible.

The complete reproduction results appear in Table 2 under column "+ Data", as in "Parser + Training + Data", meaning all data (i.e., collections $p*$) and intermediate or final results (i.e., collections $e*$) were recomputed from the previously stored raw primary data (collections $r*$). As expected from the differences in model training described above, the final results do differ from the original. However, the relative performance ranking is the same. The absolute performances are comparable, though it might be noted that with the relatively small (by today's standards) test data set of $N = 1144$ characters, the gap between the standard character bigram model and the character plus width bigram model featured in the paper shrinks from six to three characters.

6 Conclusions

The original 2010 work has been successfully repeated ten years later, if not reproduced exactly in all cases. Although many nodes in the experimental chain could be reproduced exactly, the most critical nodes involving the fitting of statistical models were only approximately reproduced. All of the straightforward data-processing tasks (i.e., image features, bigram counts) were identical; notably even the output of the statistical models (using previously learned parameters) applied to recalculated image features and the resulting predictions matched precisely.
The entire compute chain from raw data to three numbers has been stored in an experimental data repository spanning 36 collections whose 70 unique source code files having 3,500 lines of code generated 127GiB of data. This repository facilitated inspection and ease of reproduction, ensuring collections could be re-run against the correct versions of local (Subversion) source code repositories involving over 500 files and 50,000 lines of code. Java and standard C code required no modifications, and only two Matlab calls required updating. Surprisingly, the intervening decade brought very few changes to the CUDA library so that only minimal updates to the core machine learning code used in the paper were required. However, these failed to produce satisfactory results, so that a slightly newer (six months after the original publication) source version of the author's library was used. This brought comparable, though not identical results (perhaps due to differing hardware, newer libraries, or the updated client software, if not all three).
When a scientific work requires a long chain of data transformations that cannot be efficiently recomputed on the fly, caching the results and the method of computation is essential to both understanding and reproducing the work. Although not as easily portable or universal as modern-day containers such as docker or Peng and Eckel's Cacher add-on package for R [4], the transparency and simplicity of the author's filesystem-based data repository [2] has proven invaluable for a small research lab with minimal external collaborations. The repository and its function has also scaled; a more recent work by the author [10] incorporated 170 collections in its results chain whose 513 unique source files (19,400 lines of code)—not including library dependencies—generated 857 GiB of data and span nearly six years. Although the provenance of the calculations and the intermediate data are easily traceable, only time would tell whether the computations will be truly reproducible. This article indicates that using long-lived tools along with well-tracked dependencies increase the chances of generating reproducible results.

References

1. J. J. Weinman. "Typographical Features for Scene Text Recognition." In: **Proc. IAPR Intl. Conf. on Pattern Recognition**. Istanbul, Turkey, Aug. 2010, pp. 3987–3990.
2. J. Weinman. **Data Repository for Reproducible Research**. Tech. rep. DOI:11084/10001. Grinnell, Iowa: Grinnell College, 2014.
3. J. J. Weinman. "Unified Detection and Recognition for Reading Text in Scene Images." PhD thesis. University of Massachusetts Amherst, 2008.
4. R. D. Peng and S. P. Eckel. "Distributed Reproducible Research Using Cached Computations." In: **Computing in Science Engineering** 11.1 (2009), pp. 28–34.
5. E. P. Simoncelli and W. T. Freeman. "The Steerable Pyramid: A Flexible Architecture for Multi-Scale Derivative Computation." In: **Proc. Intl. Conf. on Image Processing**. Vol. 3. 1995, pp. 444–447.
6. A. L. Berger, S. A. Della Pietra, and V. J. Della Pietra. "A Maximum Entropy Approach to Natural Language Processing." In: **Computational Linguistics** 22.1 (1996), pp. 39–71.

Table 3. Details of computational experiments, automatically extracted from `INFO` file, `src/` subdirectory (using `cloc`), and `data/` subdirectory (using `du`) of each collection.

Name	Date	Src Files	Code Lines	Data Size	Description
eA	2010 01 01	3	172	30K	Minimization of error for bigram compatibility term
eB	2009 07 08	3	235	18K	Train a case-sensitive bigram maxent model with numbers tied
eC	2010 01 01	3	167	48K	Minimization of error for overlap compatibility term
eD	2009 10 29	11	523	707M	L1 regularized maxent classifier over all character and space widths plus a gap
eE	2009 07 08	2	21	4.0K	Ten-fold split of the book corpus for cross-validation
eF	2010 01 01	3	178	30K	Minimization of error for width bigram compatibility term
eG	2010 01 01	3	172	538K	Train classifier for pairwise character identity and energy, including spaces
eH	2009 06 18	2	43	204M	Scatter plots of different characters' widths in the same font
eI	2009 06 25	10	570	15M	Test of classification on pairwise character width
eJ	2010 01 01	3	162	4.0K	Test of maxent classifier(s) on parsing scene text
pA	2009 07 27	3	164	45M	Full-sized gaps in context with borders, random contrast/brightness and bias
pB	2009 07 27	3	182	23M	Full-sized gaps in context with borders
pC	2009 10 13	3	271	2.8G	Full-sized characters in context with borders and horizontal scaling
pD	2009 06 18	3	121	1.3M	Character/digit bigram counts gathered from an English corpus
pE	2009 07 14	3	163	949M	Full-sized characters in context with borders, random contrast/brightness, and bias
pF	2009 07 14	3	241	717M	Full-sized characters in context with borders
pG	2009 07 27	3	160	1.1G	Downsized steerable pyramid filtered gap images, rectified and normalized
pH	2009 07 27	3	156	55M	Down-sized gaps in context with random contrast/brightness/bias and noise
pI	2009 10 15	3	162	1.6G	Down-sized binarized characters in context
pJ	2009 07 14	3	163	22G	Downsized steerable pyramid filtered character images, rectified and normalized
pK	2009 07 28	3	155	46M	Down-sized binarized spaces in context
pL	2009 07 14	3	160	2.4G	Downsized steerable pyramid filtered space images, rectified and normalized
pM	2009 07 28	3	155	417M	Down-sized binarized characters in context
pN	2009 10 15	3	171	85G	Downsized steerable pyramid filtered character images, rectified and normalized
pO	2009 07 28	3	155	21M	Down-sized gaps binarized in context
pP	2009 10 14	3	166	4.5G	Down-sized characters in context with random contrast/brightness/bias and noise
pQ	2009 10 13	3	171	3.7G	Full-sized characters in context with borders, random contrast/brightness, and bias
pR	2009 07 14	3	158	1.2G	Down-sized characters in context with random contrast/brightness/bias and noise
pS	2009 06 18	3	159	232M	Character bounding-box widths measured in raw pixels
pT	2009 07 14	3	156	129M	Down-sized spaces in context with random contrast/brightness/bias and noise
pU	2009 07 14	3	164	104M	Full-sized spaces in context with borders, random contrast/brightness and bias
pV	2009 07 14	3	182	53M	Full-sized spaces in context with borders
rA	2005 11 18	4	266	118K	Meta-data about the 95 scene text sign images
rB	2003 09 12	N/A	N/A	21M	Size normalized, grayscale, fronto-parallel images of signs from outdoor scenes
rC	2005 08 05	2	144	232M	Raw normal letter images (no symbols or all-caps fonts)
rD	N/A	N/A	N/A	66M	85 Novels from Project Gutenberg
	Total	70*	3489*	127G	

* Total does not double-count duplicate files (defined as having the same MD5 digest sum) across collections

7. B. Krishnapuram, L. Carin, M. A. Figueiredo, and A. J. Hartemink. "Sparse Multinomial Logistic Regression: Fast Algorithms and Generalization Bounds." In: **IEEE Trans. on Pattern Anal. Mach. Intell.** 27 (2005), pp. 957–968.
8. R. H. Byrd, J. Nocedal, and R. B. Schnabel. "Representations of quasi-Newton matrices and their use in limited memory methods." In: **Mathematical Programming** 63 (1994), pp. 129–156.
9. J. J. Weinman, A. Lidaka, and S. Aggarwal. "Large Scale Machine Learning." In: **GPU Computing Gems**. Ed. by W.-m. W. Hwu. Morgan Kaufmann, 2010. Chap. 19, pp. 277–291.
10. J. Weinman. "Geographic and Style Models for Historical Map Alignment and Toponym Recognition." In: **Proc. IAPR International Conference on Document Analysis and Recognition**. Kyoto, Japan, Nov. 2017, pp. 957–964.