USENIX Journal of Election Technology and Systems (JETS)
Volume 1, Number 1 • August 2013

Improved Support for Machine-Assisted Ballot-Level Audits

Eric Kim, University of California, Berkeley
Nicholas Carlini, University of California, Berkeley
Andrew Chang, University of California, Berkeley
George Yiu, University of California, Berkeley
Kai Wang, University of California, San Diego
David Wagner, University of California, Berkeley

This paper studies how to provide support for ballot-level post-election audits. Informed by our work supporting pilots of these audits in several California counties, we identify gaps in current technology in tools for this task: we need better ways to count voted ballots (from scanned images) without access to scans of blank, unmarked ballots; and we need improvements to existing techniques that help them scale better to large, complex elections. We show how to meet these needs and use our system to successfully process ballots from 11 California counties, in support of the pilot audit program. Our new techniques yield order-of-magnitude speedups compared to the previous system, and enable us to successfully process some elections that would not have reasonably feasible without these techniques.

1. Introduction

Post-election audits form one of the most compelling tools for providing transparency and securing elections that use electronic technology. Recent research has shown that ballot-level machine-assisted audits [Calandrino et al. 2007; Benaloh et al. 2011] can offer significant improvements over current practice: significantly better assurance, at lower cost. Unfortunately, the voting systems currently deployed in the US do not support ballot-level post-election audits.

In this paper, we develop tools to facilitate ballot-level machine-assisted audits of elections conducted using current voting systems. We have been working with the State of California and various California counties to pilot new methods for ballot-level machine-assisted election audits. The approach involves re-tabulating the election using a second system that *was* designed from the start with support for ballot-level audits, checking that the second system selects the same winners as the official results, and then conducting an efficient ballot-level audit of the results from the second system [Lindeman et al. 2013]. This work focuses on the design of such a second system, called OpenCount, intended for this task.

One might wonder, why build a new system to re-tabulate the election from scratch? An alternative approach would be to extend deployed voting systems with the support needed for ballot-level machine-assisted audits. However, many of the major commercial vendors are focusing their development efforts primarily on their next-generation systems rather than on upgrading deployed systems; the deployed systems are proprietary, so it is not easy for third parties to develop extensions without vendor assistance; and updates to legacy systems may require that the entire system be first certified under new EAC standards, which may be impossible, as those systems were not designed to meet the new EAC standards. More fundamentally, many existing systems cannot be retrofitted in this way due to hardware limitations: in many deployed systems, the precinct-count optical scanners do not have the hardware capacity to record scanned images of all ballots, and there is no way to link each individual ballot to its scanned record. Therefore, pragmatically it may be easier to deploy ballot-level audits by re-tabulating the ballots using a second system that was designed with machine-assisted auditing in mind. That is the path we explore in this work. Of course, our work may be of direct relevance to future generations of voting systems that wish to provide support for efficient audits.

This work extends OpenCount [Wang et al. 2012] to provide better support for ballot-level audits, based upon our experience using it to support pilots in several California counties. This experience has helped us gain a better understanding of what is needed to allow ballot-level audits to be used in

This work was supported by National Science Foundation under grants CNS-0524745 and CCF-0424422.

practice. In particular, we identified two major shortcomings of the previous version of OpenCount. First, the previous version required election officials to compile a collection of all blank ballots (one of each possible ballot style). Due to the number of ballot types, this process proved to be labor-intensive for election officials and was a hurdle to deployment. Second, we learned that the previous version of OpenCount did not scale to large, complex elections. When there are many ballot styles, operator data entry of contest titles, candidate names, and ballot attributes (e.g., language, precinct, tally group) became extremely tedious and time-consuming. Each of these two is an independent barrier to being able to use OpenCount in large elections; either alone would be a showstopper, so we must solve both problems.

In this paper, we show how to solve both of these two problems. First, we develop new techniques to eliminate the need for scanned blank ballots. This allows us to re-tabulate an election given only scans of the voted ballots (and without access to the election database from the official voting system). Second, we develop new methods to reduce the human effort so that OpenCount will scale to large elections with many ballot types. We implement these improvements in OpenCount.

We found that these improvements are enough that we can now successfully handle large, complex elections and meet the needs of the California pilots. We evaluate the improved OpenCount on over 560,000 ballots from 12 different elections in 11 California counties and 1 Florida county. Our experiments show that these new methods enable order-of-magnitude speedups on medium-sized elections ($\sim$ 30k ballots) with many ballot styles, and enable us to process large elections ($\sim$ 120k double-sided ballots) that could not reasonably have been processed without them.

This paper makes the following contributions:

— We develop new methods to analyze and decode ballot styles, from scans of only the voted ballots (without needing scans of blank ballots or other external information). We show how to rapidly identify the ballot style of each voted ballot, how to reverse-engineer the structure of contests on the ballot, and how to recognize precinct numbers and decode barcodes that identify the ballot style of each voted ballot.

— We develop new methods to reduce the amount of human effort needed to re-tabulate an election. We show how to robustly identify multiple instances of the same contest on different ballot styles (so that the operator only needs to enter in candidate names once), how to associate voting targets with contests, how to infer the bounding box of each contest robustly, and how a human operator can verify that the results of these automated methods are accurate.

— We build a tabulation system that election officials can use to conduct ballot-level audits of their elections, and that researchers can use as a basis for future research. The system is open source and is publicly available at `https://code.google.com/p/opencount/`.

2. Related Work

OpenCount was inspired by the pioneering work of TEVS [Trachtenberg 2008] and Votoscope [Hursti 2005], which aim to solve the same problem. BallotTool [Lopresti et al. 2008] is a related system which assists an operator in retabulating a set of scanned ballots. Our work distinguishes itself in our tight integration of computer vision techniques with focused operator interaction. In [Smith et al. 2008], the authors develop a system that segments voter marks from the ballot form. However, no attempt is made to classify the marks as filled or unfilled. [Xiu et al. 2009] introduces specialized classifiers that fully automate the voter intent classification task. As future work, we intend to explore applying classifiers to further improve the ballot interpretation stages.

Document analysis is a field that considers many of the same challenges as ballot analysis. In document analysis, researchers develop systems that automatically infer the structure of documents such as forms or articles. The X-Y Cut algorithm [Nagy et al. 1992], shape-directed cover algorithm [Baird et al. 1990], and whitespace cover algorithm [Breuel 2002] are methods which excel at segmenting documents with white backgrounds and rectangular layouts. See [Mao et al. 2003; Namboodiri and Jain 2007] for a survey of the document analysis field.

Ballot analysis distinguishes itself in that it requires near-perfect accuracy in its interpretation of voted ballots. Thus, any approach must be designed with this requirement in mind - towards that

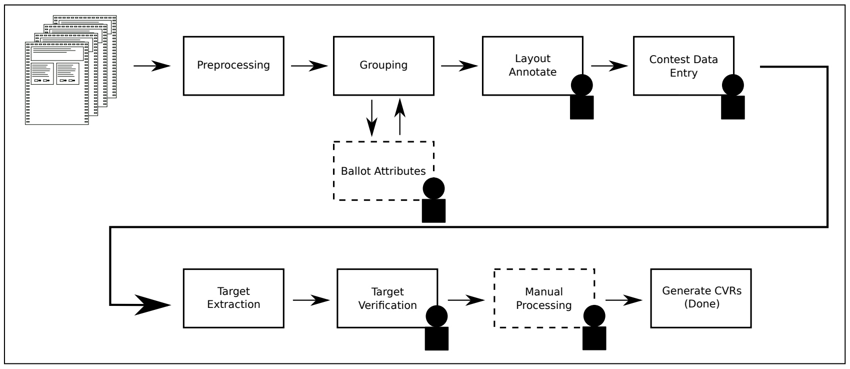

Fig. 1. Overview of the OpenCount architecture. The human outline indicates each steps that involve the operator. Steps with a dotted-line border may or may not be required.

end, OpenCount combines automated analysis with operator interaction to accurately and efficiently process ballots.

3. Overview of the OpenCount Architecture

Audit Process. OpenCount is designed to support a *transitive audit* [Lindeman and Stark 2012], where one re-tabulates the election using a second system (in this case, OpenCount), checks that the second system declares the same winners as the official results, and then audits the second system. We focus on elections conducted using paper ballots.

The audit process works as follows. After an election, election workers collect all the cast ballots and scan them all using an ordinary document scanner. Then, OpenCount processes those scanned images to extract cast vote records (CVRs) that describe the votes present on each ballot. Election officials tally the cast vote records, check that they declare the same winners as the official results, and then commit to the OpenCount CVRs. Finally, election officials conduct a public, risk-limiting audit of the ballots. During this audit process, officials repeatedly select a random ballot, pull the corresponding paper ballot, visually compare the marks on the paper ballot to the electronic CVR produced by OpenCount, and confirm that they match. Because the paper ballots are retained in the same order they were scanned, it is possible to uniquely associate each paper ballot to its corresponding CVR, which enables the ballot-level audit. Standard methods can be used to calculate how many ballots to examine and compute the level of confidence thus attained.

For instance, this process was successfully used in Napa County to audit their June 2012 primary election [Farivar 2012]. Election officials scanned 6,809 ballots, and OpenCount was used to process those scanned ballot images. Then, election officials used the CVRs produced by OpenCount to audit the election in a very close contest. Napa County officials audited 559 ballots, and found that in every case the votes on the ballots exactly matched the CVRs produced by OpenCount.

In this paper, we focus on the design of OpenCount and the algorithmic techniques needed to successfully analyze scanned images of ballots and produce CVRs. OpenCount is designed to avoid relying upon election definition files from the official voting system or other external information; we would like OpenCount to rely only on scanned images, with no further external information. The need to re-derive this information, the scale of elections, the diversity of ballot formats and voter marks, and the imperfections in scanned images make this a challenging image-processing task.

Architecture. There are four main stages to OpenCount's processing (Figure 1). First, *grouping* divides the ballots into groups, where all ballots within a group share the same ballot style (that is, the same layout and same contests). We present improvements to the grouping state that allows OpenCount to scale to larger elections. Second, in *layout annotation*, a human operator assists OpenCount in identifying the structure of the ballot (location of contests, candidates, and voting targets) and enters the titles of contests and the names of candidates. The operator only must annotate one ballot from each group. We develop novel methods to reduce the workload of layout annotation on the operator; this is crucial to enabling us to scale. One challenge here is to analyze the structure of the ballot robustly despite the presence of voter marks. Third, *ballot interpretation* involves automated processing to identify voter marks and associate them with the corresponding contest and candidate. Fourth, and finally, in *target verification*, a human operator checks OpenCount's interpretation of voter marks and inspects ambiguous or marginal marks. The output is a cast vote record for each ballot that identifies all votes found on that ballot. Ballot interpretation and target verification remain mostly unchanged, compared to the previous version of OpenCount [Wang et al. 2012]; our new work primarily affects the first two stages.

4. Design

4.1. Terminology

A *voted ballot* is the ballot after a voter has marked his/her ballot and cast it. Each ballot contains a set of *contests*. A *contest* includes a list of *candidates*, with one voting target per candidate. A *voting target* is an empty oval, broken arrow, or other location on the ballot where the voter should mark her ballot, if she wants to indicate a vote for the associated candidate. A *cast vote record* (CVR) is a record of all selections made by the voter on a single voted ballot.

The *ballot style* is the set of contests found on the ballot as well as the visual organization and location of these contests on the ballot.[1] For example, an English-language ballot may have the same set of contests as a Spanish-language ballot, but because their text is different, we consider them as two different ballot styles. Similarly, two ballots may contain identical contests, but the order of candidates in the contests may not be the same; in this case, we consider them as distinct ballot styles. Ballots may also contain a precinct number or a tally group (e.g., absentee vs. polling-place) for accumulation.

Min/Max Overlay Verification. OpenCount uses *overlays* [Cordero et al. 2010] to help the human operator verify the correctness of automated computations. Overlays help the operator quickly verify that a set of images is identical, or carries the same meaning. For instance, in Figure 3(b) the operator can immediately verify that all the images contain the word "Democratic".

Let S be a set of grayscale images of uniform dimension. The *min*-overlay of S is the image S_{min} where the intensity value at (x, y) is the *lowest* intensity value at (x, y) out of every image in S. The *max*-overlay of S is the image S_{max} where the intensity value at (x, y) is the *highest* intensity value at (x, y) out of every image in S. Intuitively, if any image in S has a black pixel at (x, y), then so does S_{min}. Similarly, if S_{max} has a white pixel at (x, y), then at least one image in S does.

If the min and max overlays of the images in S suggest to the operator that not all the images in S match, then the operator can choose to *split* S into two smaller sets. The split operation uses the k-means algorithm [MacQueen 1967] (with $k = 2$) on the images in S. The feature representation of each image is simply the pixel intensities arranged row-by-row into a vector, with the L2 norm as the distance metric. This method works well, provided there are two image classes present that exhibit different spatial-visual distributions. If there are more than two image classes present in the set S, the operator may need to perform several consecutive splits to further refine the results until each cluster is homogeneous. See Figure 2(b,c) for an illustration.

4.2. Grouping

In the first step of the pipeline, OpenCount separates the input set of voted ballots into groups, so that all ballots within a given group have the same ballot style. This stage reduces the effort required in later stages: rather than asking a human operator to annotate each individual ballot, we ask the operator to annotate only one ballot from each group.

Table I summarizes the number of groups (the number of distinct styles) for a range of elections that we processed. Notice that the number of groups is orders of magnitude smaller than the number

[1]Previous versions of OpenCount required one (unmarked) *blank ballot* per ballot style.

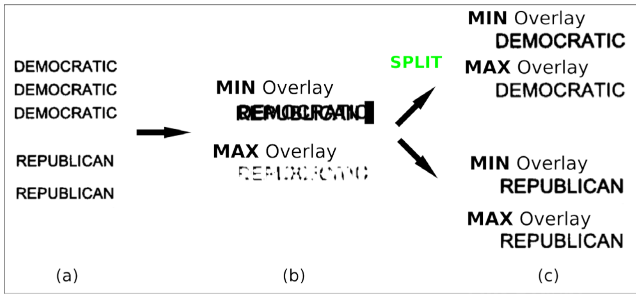

Fig. 2. The min/max overlay-based verification process. (a) The set of images. (b) The min and max overlays. (c) A single "split" operation, and the resulting overlays.

of voted ballots. Also, the number of groups does not necessarily scale linearly with the size of an election. Instead, the number of groups scales with the number of possible precincts, political party affiliations, languages, and tally groups. For instance, although Leon County has four times more voted ballots than Marin County, Leon actually has *fewer* groups than Marin.

In the previous version of OpenCount [Wang et al. 2012], we grouped ballots by matching each voted ballot to a corresponding blank ballot. However, this required election officials to gather and scan one of every possible kind of blank ballot, which we discovered is very burdensome. Therefore, we cannot rely on knowing all ballot styles a priori. Instead, our approach is to decode the barcodes present on optical scan ballots, supplemented by additional information when necessary.

4.2.1. Vendor-Specific Barcodes In deployed optical scan systems, the scanners rely on the presence of specialized, vendor-specific barcodes to determine ballot metadata such as the ballot style. These barcodes can range from simple binary bit-strings to more complex encodings (Figure 7). In most cases, the barcodes present on each ballot fully determine the ballot style, so we can group ballots by decoding the barcode and then put ballots with the same barcode into the same group. (Section 4.2.3 describes how to handle exceptional cases where the barcode alone is not enough.)

OpenCount does not require knowledge of the semantic meaning of the barcodes; we merely need to be able to decode them to bit-strings. That said, we have reverse-engineered the semantic meaning of most of the barcodes[2]. For instance, several vendors incorporate a checksum in the barcode; when possible, we verify that the checksum is valid during the decoding process.

OpenCount currently has built-in support for paper ballots from four major election vendors: Hart InterCivic, Premier Election Solutions (formerly Diebold Election Systems), Sequoia Voting Systems, and Election System & Software (ES&S). It would not be difficult to add more vendors to OpenCount in the future.

4.2.2. Ballot Attributes To facilitate comparison of OpenCount's results to the official results, OpenCount includes support to identify the precinct number and tally group of each ballot. This information is not always encoded in the barcode, but it is always printed in human-readable form on the ballot. Therefore, we provide support to enable a human operator to identify these features on the ballot; OpenCount then uses this information to automatically decode these ballot attributes. The previous system [Wang et al. 2012] did include a mechanism for this purpose. However, there were two major shortcomings present in the previous design that we address here. First, we extend the automated annotation process to make it more robust to variations in visual appearance, such as varying background colors and patterns. Second, we introduce a novel approach to decoding precinct number stamps on ballots. Decoding ballot attributes is a two-step process: attribute definition, and then (if necessary) exemplar detection.

[2]For details, see: https://code.google.com/p/opencount/wiki/Barcodes

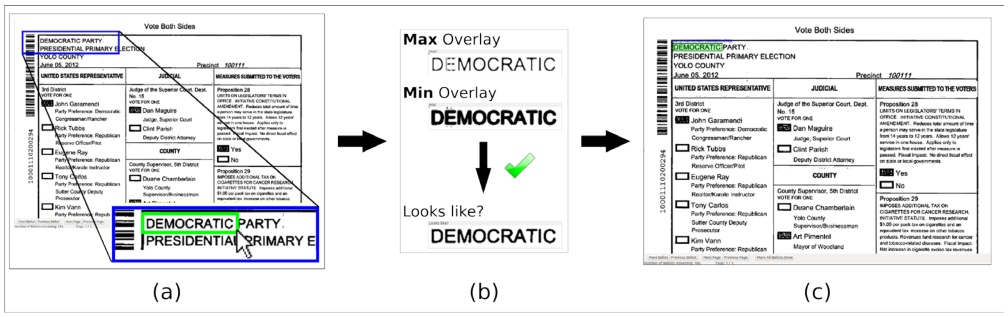

Fig. 3. Defining the “party” ballot attribute. (a) The operator draws a bounding box around the region where the party affiliation is printed on the ballot. (b) OpenCount finds other ballots that contain the same image patch and shows the min and max overlays of all matches to the operator. The operator uses the overlays to verify the matches. (c) The operator continues this labeling process until all ballots have been labeled.

Attribute Definition. Within OpenCount, the operator declares a number of “ballot attributes” for each desired property (precinct number, tally group, etc.), and then defines the possible attribute values. Additionally, the operator associates each attribute value with an image patch where this value is present in human-readable form on the ballot.

Workflow. The operator workflow is as follows. At any given moment, there is a set of ballots that have not been fully labeled. First, an unlabeled ballot is displayed. The operator draws a bounding box around the location of the desired attribute (Figure 3(a)), and specifies an attribute type and value (e.g., “party: independent”). OpenCount then searches the remaining unlabeled ballots to find ballots containing a matching image patch. Matches are identified using normalized cross-correlation (NCC) and a fixed threshold.

OpenCount then displays an overlay for all matches, so the operator can confirm that all detected matches are indeed a true match (Figure 3(b)). This process is repeated until all ballots have a label for every defined attribute type. Precinct-number attributes are handled separately (see Section 4.2.4 for details).

When defining an attribute, the operator indicates whether the value of the attribute is known to be consistent within each group. If the attribute is known to be *group-consistent*, then OpenCount will select one ballot from each group, perform the above process, and apply the discovered labels to all ballots automatically; no further processing is needed. Otherwise, to allow for the possibility that an attribute value may vary within a single group, OpenCount randomly selects 20% of the ballots from each group, and applies the above process to each of them; then, OpenCount applies exemplar detection and automatic annotation, detailed next.

Robust Exemplar Detection. At this point, only one image patch is defined for each attribute value. Depending on the election, a single image patch may not be sufficient to annotate the ballots. In Figure 4, the difference in background color (white vs. grey) overwhelms the difference in printed number (“005” vs. “006”). The image patch in Figure 4(c) is misclassified as “006” because it shares the same background as the “006” representative.

More generally, we would like visual distractions such as the background style to not affect the matching process. The previous system [Wang et al. 2012] did not have a sufficient mechanism in place to handle such variations, leading to failure cases such as Figure 4. Thus, we introduce the following extension. Rather than use only one image patch for each attribute value, OpenCount instead selects multiple representative image patches for each attribute value. These representative image patches are chosen such that they capture the possible diversity present in the dataset. The algorithm is a fixed-point iteration that takes advantage of the fact that, at this point, the software already has a set of image patches labeled with their correct attribute values (done by the operator during attribute definition).

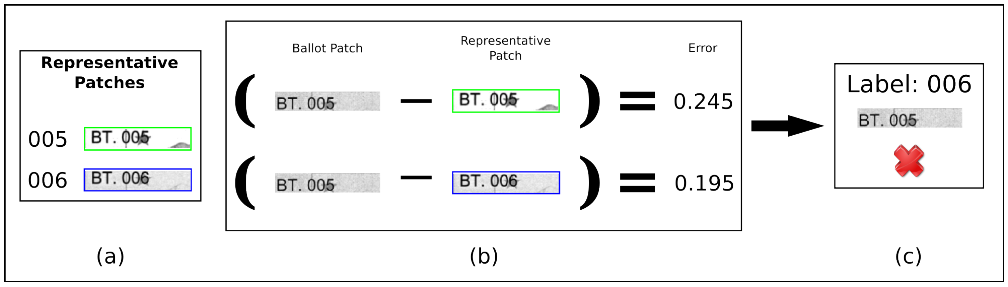

Fig. 4. (a) The representative image patches for this attribute. (b) Matching an image patch to one of the representative patches. Error is computed with the SSD metric. (c) Due to background variation, the patch was misclassified as "006".

For each attribute value, we initially choose a single image patch as the first representative patch. The algorithm then classifies all image patches with the current set of representative image patches. If an image patch is misclassified (i.e., a "democratic" patch was mistaken for a "republican" patch), then it is likely that this image patch contains some significant variation. Thus, we add this image patch to the set of representatives for that attribute value, and repeat the process.

To classify a new patch, we compare it to each representative and choose the attribute value whose sum-of-squared-difference (SSD) between the patch and the representative patch is minimized. To allow for translation and rotation invariance, we first align the image patch to the representative patch prior to computing each error score. The algorithm terminates when all image patches are classified correctly by the current set of representative image patches. OpenCount chooses the initial representative values by selecting the image patch with the highest average pixel intensity.

Automatic Attribute Annotation. Once the operator has defined the desired attributes and representative patches, OpenCount determines the attribute values of each voted ballot. For brevity, we refer the reader to Section 4.5 of [Wang et al. 2012], as the core approach remains unchanged.

Once the automated annotation is complete, OpenCount asks the user to verify the resulting labels using overlays (see Figure 2(b–c)). The operator is able to correct any errors by manually relabeling any misclassified image patches.

4.2.3. Additional Grouping In some cases, the ballot style may not be completely encoded within the barcodes. For instance, the choice of language on Sequoia-style ballots affects the location of voting targets. However, the language is not encoded in the barcodes—thus, in this case grouping by the barcodes is insufficient. In this scenario, one may use attributes as an additional grouping criterion after the barcode-based grouping has been performed. For instance, in the Sequoia example the operator should define a ballot attribute for the language[3].

4.2.4. Precinct Number Recognition We added support for OpenCount to determine the precinct of each ballot, based upon the precinct number printed on the ballot. The previous version of OpenCount used the attribute decoding process described above for this purpose. However, while processing ballots from the June 2012 primary, we found that this approach was inadequate: the number of different precincts is large enough that this imposes an overwhelming burden on the operator. Additionally, without precinct number recognition we would require one attribute per precinct; since grouping runs linear in the number of different groups, this becomes prohibitively slow.

Therefore, we designed an alternative method for decoding precinct numbers. After experimenting with off-the-shelf OCR software, we found it was not accurate enough for our purposes. Instead,

[3]We discovered that Sequoia election machines interpret voting targets via a vertical scan line through the middle of the complete-the-arrow targets, allowing invariance to vertical translation. This is in contrast to other optical scan systems, where the location of voting targets within a ballot style is fixed—hence why this additional grouping step is required. For instance, in the Alameda election, the Chinese-language ballots sometimes have voting targets at a different location than the corresponding Spanish-language ballot, because the Chinese-language candidate names used more vertical space.

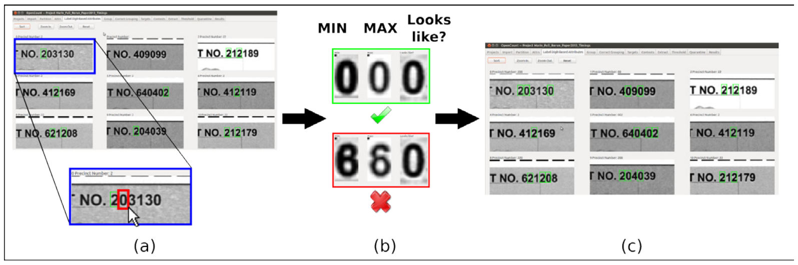

Fig. 5. The user interface for selecting digit templates. (a) The user selects one example of a "0" digit. (b) The overlay of all matches is presented to the user. After splitting once, we obtain two groups, shown stacked vertically. The top overlay shows that all digits in that group are clearly a "0". In contrast, the bottom overlay reveals that the group contains some mismatches, e.g., "6" or "8". (c) The accepted matches are labeled with their value ("0").

we designed a novel solution to this problem, which works by recognizing each individual digit and taking advantage of the fact that all digits within the same election are printed in the same font. The algorithm works in two steps: digit exemplar selection, and digit parsing.

Digit Exemplar Selection. In this step, we obtain an example of each digit (0–9) directly from the ballot images themselves. Once OpenCount knows what each digit looks like for this particular election, we can decode a precinct number by visually comparing the digits in the precinct region to its set of example digits.

To obtain examples of each digit, OpenCount displays all the precinct numbers in a grid[4], one patch from each group. See Figure 5(a). The operator selects an unlabeled digit and draws a bounding box around that digit, and enters the correct digit value (e.g., "0"). OpenCount then uses template matching across all precinct patches to find all matches for the selected digit. Any matches whose NCC (normalized cross-correlation) score exceeds a fixed threshold is retained as a potential candidate, and the operator confirms all candidate matches using an overlay verification step (Figure 5(b)). After the verification is complete, the operator-accepted matches can be labeled with their value (as shown in Figure 5(c)). The task is finished once all digits in all precinct patches have been labeled. To account for possible background pattern variations, OpenCount employs the same robust exemplar patch detection technique from Section 4.2.2 for each digit.

Digit Parsing. Once OpenCount has example image patches for each digit, it can begin to interpret the precinct patches (i.e., given an image containing a precinct number, recover the decimal string). Intuitively, each unknown digit in the precinct patch could be decoded by comparing it against all exemplars to find its closest match. However, we would like our approach to not require an initial segmentation of the unknown precinct patch into its individual digits. Ballots may not be scanned at very high resolution, and coupled with scanner noise, segmentation algorithms are likely to output incorrect results. This is exacerbated by the fact that precinct stamps are often printed in a small font. Instead, we implemented a dynamic programming algorithm to simultaneously identify the location of all digits in the unknown patch and find each digit's best match among the exemplars.

The core of our approach is a scoring algorithm that compares a representative digit against a particular precinct number through a combination of the individual digit match confidence scores and spatial relationships between adjacent pairs of digits. More formally, let $L = (l_1, \ldots, l_n)$ represent a configuration of an n-digit precinct number on a ballot where l_i is the (x, y) coordinate of the i^{th} digit. Let $m(l_i^{(c)})$ be the match cost of a digit, c at location l_i. We compute this as the NCC score between the template image of that digit and the ballot at location l_i. Now let $M(l_i) = \max_c l_i^{(c)}$ store the best match score of all digits at a given location and $Q(l_i) = \operatorname{argmax}_c l_i^{(c)}$ store the identities of the digits at those locations. Finally, let $d(l_i, l_j)$ represent the pairwise penalty of two adjacent digits

[4]The operator identifies the location where the precinct number is printed on the ballot during attribute definition.

placed at l_i and l_j. In our case, the ideal location for l_j given l_i is to be one character's width away on the x-axis while being on the same y-axis. We set $d(l_i, l_j)$ to be a quadratic cost for deviations from the ideal location. Our algorithm solves for the optimal configuration L^*, using the function:

$$L^* = \underset{L}{\operatorname{argmin}} \left(\sum_{i=1}^{n} M_i(l_i) + \sum_{(v_i,v_j)\in E} d_{ij}(l_i, l_j) \right) \tag{1}$$

The cost of any configuration of digits L is the sum of the cost of their individual match scores and sum of pairwise spatial costs. Our formulation draws from the Pictorial Structures work of [Fischler and Elschlager 1973; Felzenszwalb and Huttenlocher 2005] and a solution can be found efficiently using dynamic programming. This algorithm allows us to efficiently and accurately decode precinct numbers on all ballots, using the exemplars selected by the operator in the previous stage. Once the precinct decoding is complete, OpenCount asks the user to verify the labeling results via overlay verification. The user can correct any misclassified digits here, if necessary.

4.3. Layout Annotation

After OpenCount has grouped the ballots, the operator must then annotate the layout of one ballot from each group. In the previous version of OpenCount [Wang et al. 2012], annotation of contests and candidates in very large elections required a lot of operator effort: directly proportional to the number of contests per ballot, times the number of different ballot styles. We designed several new methods to greatly reduce the workload on the operator in complex elections.

Detecting Voting Targets. OpenCount assists the operator in identifying the location of all voting targets in every group. When blank ballots are available, this is easy: the operator can identify one example of an empty voting target, and the system can find all matches on all blank ballots. Since the blank ballots do not have any voter marks, all matches should be clean. However, when we do not have blank ballots, more sophisticated methods are necessary.

Our improved procedure works as follows. For each group, one ballot is arbitrarily selected and displayed to the operator. The operator then draws a bounding box around one of the voting targets. Once a voting target is selected, OpenCount automatically tries to detect as many matching voting targets as possible using template matching on all remaining unannotated groups. Any match whose NCC score is above a fixed threshold is accepted as a candidate voting target. To prevent multiple overlapping matches, once a match is processed, all other NCC scores within a fixed region around the match are suppressed. We apply smoothing with a Gaussian kernel to the template and ballot images prior to template matching to improve detection rates.

OpenCount applies an additional technique to further reduce operator effort. We select N representative ballots from each group, template match against all representatives, and union the results together[5]. This is intended to solve the problem that voter marks often interfere with the template matching search. For instance, if the operator draws a bounding box around an empty voting target, filled targets will not be identified as matches during the template matching. However, the same voting target will be present on the other $N-1$ representatives in that group, and it is likely that it will be empty in at least one of these cases; thus the union trick allows us to detect the location of that voting target. We set $N = 5$, which offers a good balance between convenience and performance.

Detecting Contest Bounding Boxes. We implement a method to identify the "bounding boxes" of contests on a ballot—a rectangular region that surrounds all of the voting targets in a contest. These bounding boxes help recognize which voting targets belong to the same contest, which is used for tabulation purposes. The contest bounding boxes also enable us to identify redundant contests by performing pair-wise comparisons between the set of detected contests; this enables a major reduction in the amount of data entry required.

[5]Note that this requires all N representatives within a single group to be aligned to each other.

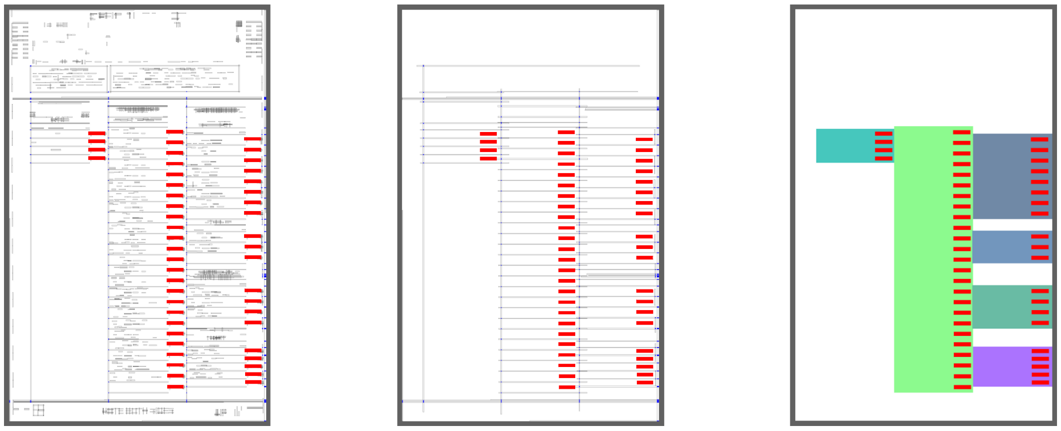
Fig. 6. The three stages of bounding box inference. First, all the lines on the ballot are identified. Second, lines are extended and extraneous lines are removed. Third, boxes are formed.

The previous version of OpenCount implemented a more naive approach. Instead of attempting to identify the actual contest bounding box, voting targets were instead grouped together by distance. However, as we experimented with large elections with many ballot styles, we discovered that this naive heuristic was both incorrect and could not find a box surrounding all candidate names.

We developed a new algorithm which is much more accurate, and also provides the full bounding boxes. We find it only makes errors the ballot has significant stray voter marks, and in those cases we provide a mechanism to select an alternate ballot in the same group without such marks. Our algorithm takes advantage of the fact that, on all ballot styles we have encountered, each contest is at least partially surrounded by horizontal and/or vertical lines that demarcate it from other contests.

First, OpenCount identifies all vertical and horizontal lines on the ballot. This is done by scanning over the ballot for pixels darker than some threshold, and attempting to identify if this is part of a long vertical or horizontal line segment. Only segments of a large enough size are retained. Line segments are identified by continuous segments of pixels darker than some threshold.

We then reduce all the scattered line segments through several intermediate steps. First, line segments are joined together if they extend in the same direction and are overlapping. Second, line segments are removed if they do not intersect with any perpendicular segments. Third, segments are removed if no voting target is located in the area it could possibly be extended to.

Next, we process the ballot to determine if there is some constant C where, if all lines were extended by a factor of C, significantly more intersections are found. This allows for cases where vertical and/or horizontal lines are present on the ballot, but do not fully intersect each other, as happens on several styles.

Finally, OpenCount searches over all candidate rectangles that can be formed by these lines (from smallest rectangle to largest), looking for rectangles that contain voting targets. When a rectangle does contain at least one voting target, all of the targets in its interior are removed and the process is repeated. For some vendors, it is necessary to combine multiple bounding boxes together to form the final set of bounding boxes; in these ballot styles, contests are finally merged. For example, Figure 6 shows that each contest is contained within its own bounding box in the second image, but in the third image many are merged together to form the correct contest bounding boxes.

Once OpenCount infers all bounding boxes, it presents them to the operator, who can manually correct any errors. OpenCount can optionally detect bounding boxes on the other four representative ballots and warn the operator if the bounding boxes are not identical on all five.

Contest Data Entry. The final step in ballot annotation is to label all detected contests with the contest title and the candidate names. This is crucial information to allow OpenCount to output human-readable election results. In principle, this is not a difficult task—one can simply ask the operator to enter the relevant text on a representative ballot from each group. In fact, this is what was done in the previous version of OpenCount.

However, in many elections there are several hundred different groups, and manually annotating every contest on every group would take many hours of manual effort. To address this challenge, we developed new methods to detect contests that are duplicates of each other. While there may be hundreds of groups, each having several contests, there are typically only a few distinct contests in the entire election. If duplicates can be detected, the operator only needs to label one instance of each contest. In practice, we find that this speeds up the contest data entry process from between a factor of ten to a hundred for large elections, not counting computation time.

Unfortunately, detecting duplicate contests is a difficult task. We experimented with many strategies for recognizing when two contests are the same by comparing them as images, but all failed. There are three key challenges. First, contests that are *semantically* equivalent may not necessarily be *visually* identical. Examples of such visual discrepancies include varying image dimensions, text lines wrapped at different locations, and inconsistent spacing between the words in candidate names. Second, the order of candidates on a given contest may be different on two different instances, due to ballot rotation. And third, some contests do not fit on a single column of a ballot, but continue onto the next column.

We resolve the first challenge by using an off-the-shelf Optical Character Recognition (OCR) software as a preprocessor to extract the text from the contest images[6]. We extract the title and the candidate names independently. To compare the similarity of two contests, we then use the Levenshtein (edit) distance [Levenshtein 1966] between the extracted text of the two contests. By comparing the OCR outputs between two contests, rather than the contest images themselves, we remain invariant to the visual confounding factors mentioned previously. Crucially, this approach allows us to tolerate inaccuracies in the OCR output. For example, if the OCR were only accurate on 50% of the characters, we would expect the Levenshtein distance to be 25% smaller for two identical contests than for two different contests (of similar length). In practice, we found that Tesseract makes many errors, but it is still over 90% accurate on most ballot styles and languages.

Solving the second challenge requires more work. At a high level, whenever we test a pair of contests for equivalence, we search over all possible rotations of the candidate names and find the best match. For all elections we are aware of, candidate names are not permuted arbitrarily. Instead, they are simply rotated some number of positions, with one modification: the write-in candidate(s) always appear at the end. Therefore, our algorithm takes advantage of this fact. Once we have extracted both the contest title and the candidate names for each voting target, we attempt to determine (a) the number of write-in candidates and (b) the rotation amount. In particular, we search over all possible number of write-ins, and all possible rotations. We then treat two contests as duplicates if they have both the same number of write-in contests, and if they have a small Levenshtein distance between the names of corresponding candidates.

We solve the third challenge by letting the operator mark contests that span multiple columns while labeling the voted ballots: if the operator finds such a contest while entering text on the ballots, she clicks a button marking it as such, and continues on.[7] OpenCount then performs another search over all consecutive pair of contests to attempt to identify other contests that also are split across two bounding boxes by comparing their similarity against the pair the operator marked.

We implement several optimizations to improve performance. First, contests are only compared if they have the same language and same number of targets; and second, we sort candidates by length and perform an initial linear scan where we compare consecutive contests, to detect some duplicates quickly. These two optimizations typically reduce computation time by a factor of a hundred on large elections.

[6]We use Tesseract [Smith 2007] for this task. OpenCount does support multiple languages for this task. As Tesseract is able to interpret a range of languages, if the operator defines a "language" ballot attribute (Section 4.2.2), then OpenCount will interpret each ballot with the correct language.
[7]The user interface displays the ballot image during the data entry process, making it easy for the user to notice such contests.

Table I. General information on elections processed with OpenCount.

County	Number of Ballots	Number of Styles	Number of Sides	Image Resolution
Alameda	1,374	8	1	1460x2100
Merced	7,120	1	1	1272x2100
San Luis Obispo	10,689	27	1	1700x2200
Stanislaus	3,151	1	1	1700x2800
Ventura	17,301	1	1	1403x3000
Madera	3,757	1	1	652x1480
Marin	29,121	398	1	1280x2104
Napa	6,809	11	2	1968x3530
Santa Cruz	34,004	136	2	2400x3840
Yolo	35,532	623	1	1744x2878
Leon	124,200	216	2	1400x2328
Orange	294,402	1,839	1-3	1715x2847

Finally, the we present operator with an overlay of the contests. We align candidates independently, and merge them together to form a single overlay. The user then verifies that the resulting overlays indeed correspond to the same contest. See Figure 8 (Right) for an example overlay.

4.4. Ballot Interpretation and Result Generation

Once the operator has annotated a representative of each group, OpenCount uses this information to interpret the remaining ballots. We did not need to make major improvements or changes to this stage from the previous version of OpenCount [Wang et al. 2012], though we did take the opportunity to improve performance in several places. For convenience, we summarize this stage:

— **Extract voting targets.** In each group, OpenCount takes a representative and globally aligns all other ballot to it. OpenCount then locally aligns each voting target to the corresponding voting target on the representative, and extracts it. The current version runs on average three times faster.

— **Classify voting targets.** OpenCount then presents the operator with a grid of targets sorted by average intensity, who then identifies which targets are filled. We added the capability for the operator to set filters to only show overvotes and undervotes or only targets from a given contest, or to run a second classifier over the voting targets and display only disagreements.

— **Handle quarantined ballots.** Finally, the operator handles all ballots that were quarantined. The operator manually indicates the votes for each contest, as well as the ballot attribute values. Partial information about each ballot is automatically populated when possible. For instance, if a ballot was quarantined after contests were inferred, this information is automatically filled. The previous version of OpenCount did not have this feature, making it much more labor-intensive.

— **Generate results.** OpenCount generates CVRs and cumulative vote tallies. Each CVR specifies the candidates that the voter voted for. Overvotes and undervotes are identified as such. The cumulative vote tallies show the vote totals for each candidate in each contest. These totals are further broken down by precinct and tally group if those attributes are available, which assists in comparing with the official results at a finer granularity.

5. Evaluation

5.1. Methodology

We ran OpenCount on the 12 elections listed in Table I, and also ran the previous version of OpenCount [Wang et al. 2012] on a subset of the elections. We timed how long each took, on a six-core machine Intel i7-3930K processor with 16GB of RAM. Table II records the results.

Different operators ran different elections, so comparisons on timing statistics from election to election may not be meaningful. However, whenever we ran both versions of OpenCount on an election, the same operator ran both versions. On elections where both versions of OpenCount were run, the results between election runs match on $\geq 99.9\%$ of ballots.[8] The only differences were

[8]Exception: The previous version of OpenCount produced inaccurate results in one election, Napa, as described below.

Table II. Election timing data with OpenCount. Steps that require operator interaction are labeled with (H), and steps that only perform computation are labeled with (C). The second column identifies the version of OpenCount used; the 2013 version contains the improvements described in this paper, whereas the 2012 version does not. Times are rounded to the nearest minute, except for the "Total" column. Entries tagged with a * are extrapolated (see text for details).

County	Version	Decode (C)	Ballot Attrs. (H)	Group (C)	Group Check (H)	Layout Annotate (H)	Label Contests (H)	Target Extract (C)	Target Check (C)	Avg. per ballot	Total
Stanislaus	2013	0m	-	-	-	1m	1m	5m	1m	0.14s	7m 18s
	2012	-	-	-	-	1m	2m	11m	4m	0.33s	17m 30s
Merced	2013	0m	-	-	-	0m	1m	8m	3m	0.11s	12m 31s
	2012	-	-	-	-	1m	1m	22m	2m	0.22s	25m 32s
Ventura	2013	0m	-	-	-	2m	1m	17m	3m	0.08s	23m 6s
	2012	-	-	-	-	1m	1m	39m	3m	0.15s	43m 8s
Alameda	2013	1m	6m	2m	1m	3m	3m	1m	1m	0.75s	17m 6s
	2012	-	3m	4m	0m	2m	3m	6m	3m	0.96s	22m 1s
SLO	2013	10m	3m	0m	0m	4m	1m	11m	1m	0.17s	30m 35s
	2012	-	3m	32m	9m	2m	2m	37m	1m	0.48s	1h 25m
Madera	2013	0m	-	-	-	1m	0m	3m	2m	0.11s	6m 38s
	2012	-	-	-	-	1m	1m	5m	2m	0.14s	8m 30s
Napa	2013	20m	11m	26m	1m	3m	7m	33m	22m	1.02s	1h 56 9s
	2012	-	10m	1h 2m	42m	4m	13m	2h 30m	42m	2.85s	5h 23m
Marin	2013	52m	6m	0m	0m	2h 47m	2h 20m	5h 17m	32m	1.47s	11h 53m
Santa Cruz	2013	3h 38m	-	-	-	40m	1h 2m	9h 46m	3h 45m	2.00s	18h 50m
Yolo	2013	25m	7m	0m	0m	52m	26m	1h 40m	6m	0.37s	3h 36m
	2012	-	16m	10h 5m	40h*	25m	52m*	6h 42m	9m	5.92s	58h 27m
Leon	2013	3h 40m	4m	0m	0m	29m	44m	8h 27m	36m	0.41s	14h 2s
Orange	2013	4h 9m	2h 19m	0m	0m	8h 48m	8h 13m*	2d 9h 27m	13h 5m*	1.149s	3d 22h 39s

Table III. Quarantined ballots requiring manual processing. For entries marked with a (*), we processed 10% and extrapolated the required time.

County	Version	Number of Quarantined Ballots	Fraction of Election Size	Total Time
Stanislaus	2013	1	0.03%	29s
Merced	2013	1	0.01%	48s
Ventura	2013	1	0.01%	8s
Alameda	2013	191	13.90%	1h 4m 22s*
SLO	2013	14	0.13%	5m 46s
Madera	2013	0	0.00%	0s
Napa	2013	16	0.23%	7m 23s
Marin	2013	129	0.44%	1h 46m 10s*
Santa Cruz	2013	163	0.48%	2h 16m 30s*
Yolo	2013	60	0.17%	1h 35m 0s*
Leon	2013	602	0.48%	8h 11m 34s*
Orange	2013	192	0.00065%	4h 14m*

almost always the result of the operator interpreting ambiguous marks differently (i.e., a mark that may either be caused by an erasure, or may be a lightly-filled in mark). OpenCount developers processed each elections.

We also invited Tom Stanionis, an election official from Yolo county, to evaluate the OpenCount software. Mr. Stanionis was able to successfully process an election subset without any significant intervention from the developer conducting the evaluation. We gathered valuable feedback and constructive criticism about the software from a usability standpoint.

5.2. 2011 Elections

We analyzed five special elections from 2011 (the same ones evaluated in [Wang et al. 2012]): Stanislaus, Merced, Ventura, Alameda, and San Luis Obispo (SLO). All of these have only a few contests. Three have only a single ballot style. Alameda has 8 ballot styles, but is the smallest election we analyzed, with only 1,374 ballots. SLO has 27 ballot styles, but all styles contain the same two contests, differing only in precinct and tally group.

The results are given in Table I. Because these elections are so small and contain so few contests and so few ballot styles, the 2011 elections do not stress-test the ability of OpenCount to handle complex elections with many ballot styles. For instance, Alameda shows the least benefit from our improvements: with so few ballots, very few of the additions have an impact on total running time. Instead, the total time spent is dominated by initial setup. As it happens, many ballots in the Alameda election were quarantined because the scanned ballot was missing the top portion of the ballot, which included the barcode, see Table III for full quarantine statistics.

5.3. 2012 Elections

We also analyzed ballots from the June 2012 primary in five counties. These elections were more complex. As a primary election, they contained many more ballot styles. Madera was by far the smallest, with only one ballot style, and we encountered no difficulties.

Napa. Napa had 28 ballot styles and we were able to process all 6,809 ballots using our improved version of OpenCount without incident. The previous version of OpenCount made serious errors during target extraction and extracted the wrong regions on nearly 50% of ballots. On ballots where targets were extracted correctly, the two versions matched with over 99.9% accuracy. On the remaining 50%, we randomly sampled 20 ballots and, in every case, the current version of the software was completely correct, and the previous version was wrong. Our improvements led to a modest (2.8$\times$) speedup; however, with so few ballot styles, many of our improvements do not take effect.

Marin. The Marin election had 398 ballot styles, so it was a good stress test of our improvements to reduce operator effort. The improvements were crucial to our ability to successfully analyze this election. However, target extraction failed for 115 ballots, due to poor image alignment, and these ballots had to be flagged for manual processing. This dominated the time it took for us to process the Marin ballots.

Santa Cruz. Santa Cruz had fewer ballot styles, which made processing go quickly. However, verifying the interpretation of marks was made more challenging because Santa Cruz uses a Sequoia complete-the-arrow style ballot, and OpenCount's visualization tool is not as effective for these ballots. This is a good opportunity for further improvements. Also, target extraction failed for 84 ballots, which had to be flagged for manually processing. In each case, the initial image alignment between the ballot and the reference image was incorrect, causing a wildly inaccurate alignment to occur. See Section 6.2 for additional discussion about these alignment issues.

Yolo. Yolo was our second most complex and third-largest election. The time to label contests includes the 13 minutes of computation OpenCount spent detecting equivalent contests, which massively reduced the amount of data-entry required, from 4,603 contests to fewer than 50.

The absence of blank ballots made it significantly more challenging to identify all voting targets: this task took two times longer than on the previous version, when blank ballots were available. This is due both to the increase in the number of groups—there are 117 different blank ballots, but this increases to 623 groups when we do not have blank ballots—and to the increase in operator workload due to the presence of voter marks—the operator must repeatedly identify empty voting targets, to ensure all are found. As an experiment, we ran the ballot annotation process of the current version of OpenCount on the 117 blank ballots, and found that it ran six times faster.

We did not complete grouping verification in its entirety on the previous version of OpenCount. After finishing 5% of grouping verification (in two hours), we extrapolated that grouping verification would take about 40 hours. We used data from the current version of OpenCount to fill in the remaining data.

Leon. With 124,200 ballots, the November 2008 general election in Leon County, Florida was our second largest election processed. We only ran Leon on the most recent version of OpenCount;

processing it with the previous version would have taken an unreasonable amount of time. The contest labeling step of Leon took much longer than that of Yolo, because Leon had several contests of the form "District court of appeals. Shall Justice __ be retained in office" for each of five different justices. This caused the contest equivalence class computation to erroneously infer that all of these contests were identical; we had to reject and manually label these contests, which took extra time.

Orange. Finally, we processed the Orange county June 2012 Presidential Primary Election, which consisted of 294,402 voted ballots with a variable number of sides per ballot, ranging from one to three. This election has as many ballots as all other elections combined. Similar to Leon County, we only processed the ballots with the most recent version of OpenCount.

This election dataset posed several significant challenges. First, the scan quality of these images is noticeably poorer than that of any of the other elections. Rather than rescanning the paper ballots with commercial scanners to obtain high-quality image scans, these images were output directly from the Hart voting systems. This is both good and bad; while no additional scanning effort was necessary, we had to modify OpenCount to handle much more challenging images due to scanning conditions outside of our control. Additionally, the images are binary images, thresholded by the Hart system, which further degraded the image quality through the loss of information. Second, the complexity of the election is by far the greatest: there are 1,839 distinct ballot styles [9].

While completing the "Target Check" stage, we discovered that the target sorting metric currently used is inadequate for elections of this size, and requires the operator to inspect an overwhelming number of voting targets. The average-intensity heuristic used for sorting targets breaks down when voter marks are small in comparison to the voting target itself. Image noise then dominates the sorting metric, resulting in empty voting targets being mixed with filled-in voting targets, requiring many hours of manual corrections. It is worth noting that these sparsely filled-in voting targets are typically cases where the voter drew in an "X" or a line to indicate his or her vote. Improving this step is a focus for future work, and may draw upon ideas from the style-based mark classifier of [Xiu et al. 2009]. Thus, we completed a small portion of the "Target Check" stage and extrapolated the total time required. The timing data for the "Label Contests" stage was similarly extrapolated.

Interestingly, Orange County had the lowest percentage of quarantined ballots (Table III).

6. Discussion

One question we must consider is how well OpenCount will generalize. For this system to be useful, it must minimize the restrictive assumptions it makes on specific ballot details. We consider the major assumptions that OpenCount makes in this section.

6.1. Ballot Assumptions

We assume each ballot image contains a barcode-like structure that can be used to group ballots by ballot style. We do not assume that the barcode uniquely determines ballot style.

Also, contests must be roughly box-shaped and be at least partially surrounded by lines on at least three sides. Furthermore, each contest must contain the contest title followed by a list of candidate names, with one voting target corresponding to each candidate (including write-in candidates[10]). OpenCount does support ballot rotation: the ordering of the candidate names displayed within a contest is allowed to vary from ballot to ballot.[11]

Voting targets may be any consistent structure that the voter is asked to mark. This includes the common "fill-in-the-bubble" and "fill-in-the-arrow" styles.

[9]We used a spreadsheet file output by the Hart voting system that mapped precinct number to "ballot type" to do the grouping - otherwise there would be 22,025 ballot styles. This was a pragmatic decision on our part, as we would prefer not to rely on files generated by the voting system.

[10]OpenCount does not attempt to interpret handwriting for write-in votes—it only records the presence of write-in votes.

[11]Our current implementation assumes that only rotations of the candidates are possible, not arbitrary permutations. So far, we have not observed an election that violates this assumption. However, if necessary, it would be trivial to extend OpenCount to support arbitrary permutations using maximum-weighted matching.

Consequences and Relaxation. OpenCount is currently able to support optical scan ballots from Diebold (Premier), ES&S, Hart, and Sequoia systems, which accounts for the overwhelming majority of opscan ballots in the US. Of the requirements listed above, the most stringent is the requirement that the ballot contain a barcode that can be used for grouping. Fortunately, all ballots that we have encountered to date meet this requirement. To relax this requirement, one could imagine applying algorithms that group ballots on visual appearance. However, we have not investigated the challenges associated with doing so.

Finally, while OpenCount currently does not support voting schemes such as straight-party voting or fusion voting, it would be simple to modify the logic to accommodate such voting schemes.

6.2. Scanner Assumptions

In several stages of the pipeline, ballots are aligned to a representative ballot. The image alignment method used in OpenCount assumes a rigid transformation model, which allows translation and rotation variation. The simpler rigid model performed better than more complex models such as the affine model, as the rigid model has fewer parameters to solve for, typically leading to higher-quality solutions. Additionally, in practice, almost all images can be aligned with only translations and rotations, and do not require other affine warps (scaling, shearing, etc.).

Our alignment scheme is able to tolerate a moderate amount of non-rigid warping, due to the fact that it performs two alignment steps: a coarse "global" alignment of the entire image, followed by a finer "local" alignment applied to small regions of the image. Both alignments still use rigid transformations. If an image contains non-uniform warping (e.g., the bottom-half of the image is stretched/skewed more than the top), then no precise global alignment will exist between this warped image and its reference image. However, a good local alignment can often still be recovered, allowing our scheme to tolerate a moderate amount of non-uniform warping. If ballots contain too much non-rigid warping, then mis-alignment errors will cause issues during target extraction. Such issues affected 0.2% of Santa Cruz ballots, as mentioned in Section 5.3.

6.3. Scalability

The improvements made to the OpenCount system in this paper have enabled the processing of significantly larger and more complex elections. The previous system was not able to reasonably process the Yolo county election, let alone the Leon or Orange datasets. The current proposed system, on the other hand, is able to process all three without trouble.

However, it remains to be seen how the system will scale to even larger elections. For instance, about 3.2 million ballots were cast in November 2012 in Los Angeles county [LA-Registrar 2012]. This is roughly $11\times$ larger than the Orange county election. Future work is needed to investigate whether OpenCount is able to efficiently handle elections of that magnitude.

7. Conclusion

We have introduced several significant improvements to the original OpenCount system that enables OpenCount to scale to significantly larger elections, and process them an order of magnitude faster. The new system does not require the collection of blank ballots, which reduces the barrier on election officials; we also greatly reduce the amount of data entry required, which speeds the re-tabulation process. These improvements enabled OpenCount to successfully support audit pilots in 11 California counties and advance the state of the art in ballot image analysis.

ACKNOWLEDGMENTS

We thank the California Secretary of State's office and election officials at Stanislaus, Merced, Ventura, Alameda, San Luis Obispo, Madera, Marin, Napa, Santa Cruz, Yolo, Orange, and Leon counties for sharing scanned ballots, and Clear Ballot for their assistance with obtaining ballot scans. This research would not have been possible without their generous assistance. We thank Philip Stark for helpful guidance and Tom Stanionis for providing feedback on OpenCount.

REFERENCES

H.S. Baird, S.E. Jones, and S.J. Fortune. 1990. Image segmentation by shape-directed covers. In *Proceedings of ICPR*, Vol. i. 820–825 vol.1.

Josh Benaloh, Douglas Jones, Eric L. Lazarus, Mark Lindeman, and Philip B. Stark. 2011. SOBA: Secrecy-preserving Observable Ballot-level Audits. In *Proceedings of EVT/WOTE*.

Thomas M. Breuel. 2002. Two Geometric Algorithms for Layout Analysis. In *Proceedings of DAS*. 188–199.

Joseph A. Calandrino, J. Alex Halderman, and Edward W. Felten. 2007. Machine-Assisted Election Auditing. In *Proceedings of EVT*.

Arel Cordero, Theron Ji, Alan Tsai, Keaton Mowery, and David Wagner. 2010. Efficient user-guided ballot image verification. In *Proceedings of EVT/WOTE*.

Cyrus Farivar. 2012. Saving throw: securing democracy with stats, spreadsheets, and 10-sided dice. http://arstechnica.com/tech-policy/2012/07/saving-american-elections-with-10-sided-dice-one-stats-profs-quest/. (24 July 2012).

Pedro F. Felzenszwalb and Daniel P. Huttenlocher. 2005. Pictorial Structures for Object Recognition. *International Journal of Computer Vision* 61, 1 (Jan. 2005), 55–79.

Martin A. Fischler and R.A. Elschlager. 1973. The Representation and Matching of Pictorial Structures. *Computers, IEEE Transactions on* C-22, 1 (1973), 67–92.

Harri Hursti. 2005. Votoscope Software. (2 October 2005). http://vote.nist.gov/comment_harri_hursti.pdf.

LA-Registrar. 2012. Los Angeles County: Statement of Votes Cast Election Results (Nov. 2012 General Election). http://www.lavote.net/VOTER/PDFS/STATEMENT_VOTES_CAST/11062012_SVC.pdf. (2012).

VI Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966).

Mark Lindeman, Ronald L. Rivest, and Philip B. Stark. 2013. Retabulations, Machine-Assisted Audits, and Election Verification. (2013). http://www.stat.berkeley.edu/~stark/Preprints/retabulation13.htm.

Mark Lindeman and Philip B. Stark. 2012. A Gentle Introduction to Risk-limiting Audits. *IEEE Security and Privacy* 10, 5 (2012). Special Issue on Electronic Voting.

Daniel Lopresti, George Nagy, and Elisa Barney Smith. 2008. A Document Analysis System for Supporting Electronic Voting Research. In *Proceedings of DAS*. 167–174.

J. B. MacQueen. 1967. Some Methods for Classification and Analysis of MultiVariate Observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, Vol. 1. 281–297.

Song Mao, Azriel Rosenfeld, and Tapas Kanungo. 2003. Document structure analysis algorithms: a literature survey. In *Proceedings of SPIE*, Vol. 5010. 197–207.

George Nagy, Sharad Seth, and Mahesh Viswanathan. 1992. A Prototype Document Image Analysis System for Technical Journals. *Computer* 25, 7 (July 1992), 10–22.

Anoop M. Namboodiri and Anil K. Jain. 2007. Document Structure and Layout Analysis. (2007).

Elisa H. Barney Smith, Daniel Lopresti, and George Nagy. 2008. Ballot Mark Detection. In *Proceedings of ICPR*.

R. Smith. 2007. An Overview of the Tesseract OCR Engine. In *Proceedings of ICDAR*, Vol. 02. 629–633.

Mitch Trachtenberg. 2008. Trachtenberg Election Verification System (TEVS). (2008). https://code.google.com/p/tevs/.

Kai Wang, Eric Kim, Nicholas Carlini, Ivan Motyashov, Daniel Nguyen, and David Wagner. 2012. Operator-assisted tabulation of optical scan ballots. In *Proceedings of EVT/WOTE*.

Pingping Xiu, Daniel Lopresti, Henry Baird, George Nagy, and Elisa Barney Smith. 2009. Style-Based Ballot Mark Recognition. In *Proceedings of ICDAR*.

A. Appendix

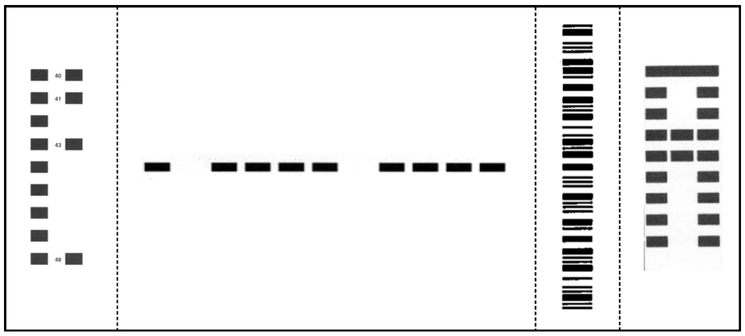

Fig. 7. Examples of barcodes from major vendors. From left to right: ES&S, Diebold, Hart, Sequoia.

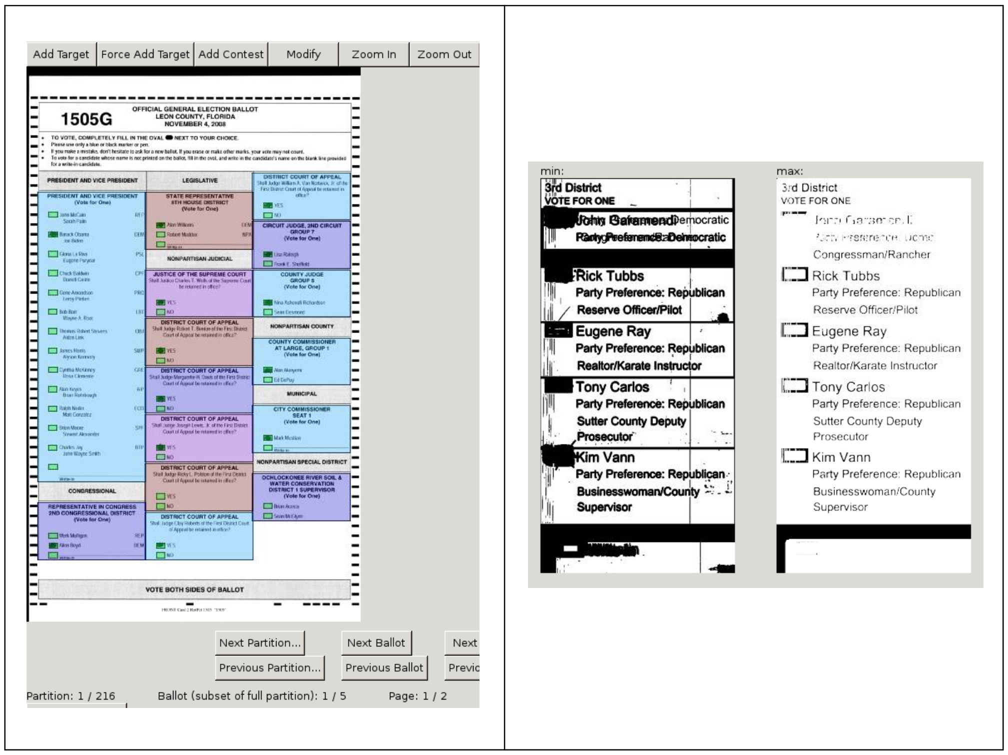

Fig. 8. (Left) The interface where the operator annotates the ballot style for each group. Here, both the voting targets and contest bounding boxes have already been found. (Right) An example of overlay-based verification. Note that there is an error on the first contest.