Algorithms for the Automated Correction of Vertical Drift in Eye Tracking Data

Jon W. Carr, Valentina N. Pescuma, Michele Furlan, Maria Ktori, Davide Crepaldi
Cognitive Neuroscience, International School for Advanced Studies, Trieste, Italy

Abstract

A common problem in eye tracking research is vertical drift—the progressive displacement of fixation registrations on the vertical axis that results from a gradual loss of eye tracker calibration over time. This is particularly problematic in experiments that involve the reading of multiline passages, where it is critical that fixations on one line are not erroneously recorded on an adjacent line. Correction is often performed manually by the researcher, but this process is tedious, time-consuming, and prone to error and inconsistency. Various methods have previously been proposed for the automated, post-hoc correction of vertical drift in reading data, but these methods vary greatly, not just in terms of the algorithmic principles on which they are based, but also in terms of their availability, documentation, implementation languages, and so forth. Furthermore, these methods have largely been developed in isolation with little attempt to systematically evaluate them, meaning that drift correction techniques are moving forward blindly. We document nine major algorithms, including two that are novel to this paper, and evaluate them using both simulated and natural eye tracking data. Our results suggest that a method based on dynamic time warping offers great promise, but we also find that some algorithms are better suited than others to particular types of drift phenomena and reading behavior, allowing us to offer evidence-based advice on algorithm selection.

Keywords: algorithms; dynamic time warping; eye tracking; reading; vertical drift

Introduction

Reading is a fundamental skill for navigating modern society and, as such, is subject to intense study in the cognitive and language sciences. Among the many tools that researchers use to investigate reading in the laboratory, eye tracking occupies a prominent

Cite as: Carr, J. W., Pescuma, V. N., Furlan, M., Ktori, M., & Crepaldi, D. (under review). Algorithms for the automated correction of vertical drift in eye tracking data. https://doi.org/10.31219/osf.io/jg3nc

position. Using this technique, participants' eye movements may be recorded as they read written material, providing a window into the relevant cognitive processes as they unfold. Technological advancements in eye tracking, particularly from the 1970s (see e.g., Rayner, 1998), have allowed scholars to collect increasingly accurate measures of eye movements during reading tasks, leading to great improvements in the investigation of the cognitive processes underlying reading and reading acquisition.

Many eye tracking studies involve the reading of single words or sentences. For example, researchers may embed target words into different sentence contexts and manipulate predictability (e.g., Rayner et al., 2001), display isolated words to gain insight into how a reader's eye moves when processing a word (e.g., Vitu et al., 2004), or reveal parts of words in a gaze-contingent fashion to investigate parafoveal processing (e.g., Schotter et al., 2012). Sentence reading experiments have also been essential in revealing the cognitive processes behind different levels of written language processing, from the width of the perceptual span (e.g., Blythe et al., 2009; Rayner, 1986) to the effects that word length and frequency have on eye movements (e.g., Joseph et al., 2009; Tiffin-Richards & Schroeder, 2015), as well as the effects of syntactic (e.g., Frazier & Rayner, 1982; Pickering & Traxler, 1998) and lexical (e.g., Sereno et al., 2006) ambiguity.

In our everyday experience, however, we often do not encounter sentences in isolation; a good part of our reading experience involves connected text that is distributed over multiple lines. Therefore, experiments based on paragraph reading also provide insight into the reading experience, while allowing us to address levels of processing that are simply not available when one reads a single sentence, such as the role of broader context or the integration of syntactic relations across sentence boundaries (Jarodzka & Brand-Gruwel, 2017). Indeed, studies of multiline reading have become more prevalent in recent years, with researchers using passage reading tasks to investigate, for example, the effect of text- and participant-level characteristics on eye movements (Kuperman et al., 2018) or of contextual facilitation on developing readers' eye movements (Tiffin-Richards & Schroeder, 2020). Eye tracking databases of multiline reading, such as GECO (Cop et al., 2017) and Provo (Luke & Christianson, 2018), have also been made available.

A technical issue that arises from the particular circumstances of multiline reading is so-called "vertical drift," which we define as the progressive displacement of fixation registrations on the vertical axis *over time*. In other words, fixations may be recorded above or below the line of text that the participant was actually reading, and the amount of measurement error may fluctuate over time, making it nontrivial to eliminate. Fig. 1A depicts a reading trial exhibiting vertical drift phenomena; in this case, fixations—especially those on the left hand side—are recorded around one line higher than where the reader was actually fixating, but they also tend to slope down to the right such that fixations on the right hand side seem to be better aligned.

Vertical drift can occur quite unpredictably, even following good quality calibration, and it is likely caused by subtle movements in head position or changes in pupil dilation, which can be difficult to control for, even in a laboratory setting. Such sources of measurement error are often exacerbated in the context of multiline reading because the reading of a passage of text necessarily takes longer than single words or sentences, during which time calibration may begin to degrade. In addition, there are less frequent opportunities to recalibrate the device, since calibration can only be performed between trials and not

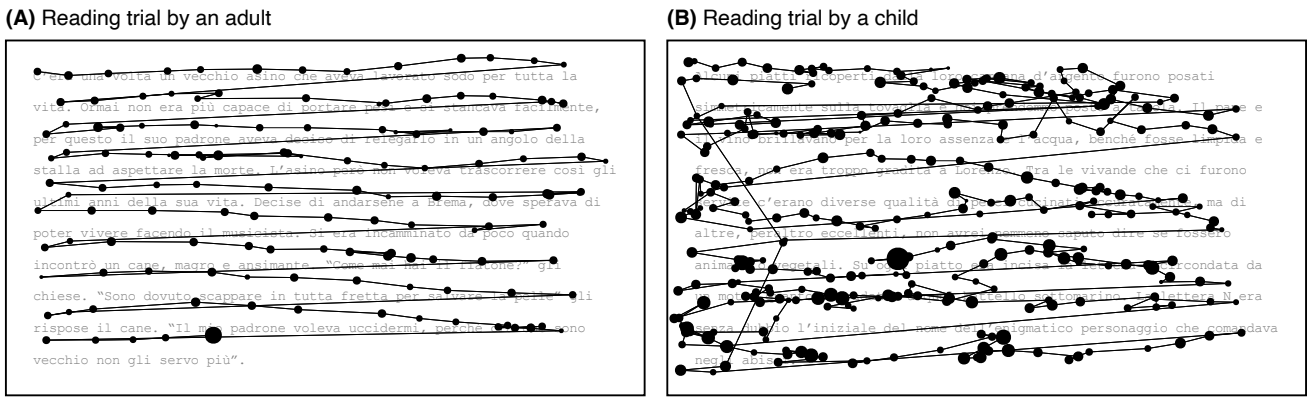

Figure 1
Example reading trials from an adult participant (left) and a child participant (right) taken from Pescuma et al. (in prep.). Each dot represents a fixation and the size of the dots represents duration. The adult trial exhibits upward shift, especially in the lower left part of the passage. The child trial is extremely noisy and exhibits not just vertical drift issues but also many natural reading phenomena that will pose challenges to the algorithms.

during the reading of a passage.

Whatever the cause and however it manifests itself, vertical drift will ultimately have a negative impact on the analysis of eye tracking data because fixations will be mapped to words that were not actually being fixated at a given point in time (as we can see in Fig. 1A). It is therefore incumbent on the researcher to recognize such issues when they occur and to take corrective measures. Specialized software packages, such as EyeLink Data Viewer (SR Research, Toronto, Canada) or EyeDoctor (UMass Eyetracking Lab, Amherst, USA), provide the ability to manually move fixations, either individually or in small batches. However, manual realignment can be very time-consuming and is likely to be error-prone. In particular, the realignment process can be greatly complicated by other sources of noise or idiosyncratic reading behaviors. For example, Fig. 1B depicts a reading trial by a child reader; in this case, not only is the data affected by drift issues, but there are also various natural reading behaviors, such as within- and between-line regressions, which add an additional layer of complexity to the task of realignment, not to mention the baseline level of noise and unusual features such as the arching sequence of fixations targeting line 4.

A number of methods have previously been developed to automate post-hoc vertical drift correction. `FixAlign`, an R package developed by Cohen (2013), is currently the most well-established method in the experimental psychology community, although other methods have recently been proposed by Schroeder (2019) and Špakov et al. (2019). In addition, there is a disparate body of work from several subfields of computer science, such as biometrics (Abdulin & Komogortsev, 2015), educational technology (Hyrskykari, 2006), and user-interface design (Beymer & Russell, 2005), in which various ad-hoc algorithms have been reported (see also Carl, 2013; Lima Sanches et al., 2015; Martinez-Gomez et al., 2012; Mishra et al., 2012).

These reported methods can be difficult to evaluate and use because they vary widely in terms of their availability, design choices, implementation languages, usability, level of

documentation, expected input data, and the extent to which they rely on project-specific heuristics or particular eye tracker hardware. Furthermore, these methods have largely been developed in isolation from each other, and there has been little attempt to systematically evaluate them, so drift correction software is moving forward blindly without an evidence base to support new directions. In this paper, we attempt to classify the reported methods into nine major approaches, which we formalize as nine simple algorithms that adopt a consistent design model. In other words, we do not attempt to evaluate existing software implementations; rather, we explore the spectrum of drift correction algorithms by isolating and evaluating the core principles on which previous methods have been based. Our goal is to provide a systematic comparison of these algorithms in order to guide researchers' choices about the most suitable methods and to lay a solid foundation for future drift correction software.

To be clear, the algorithms we consider in this paper are restricted to one specific type of problem. Firstly, we only consider algorithms intended for use on passages of text; other uses of eye tracking, such as visual search and scene perception, can also undergo drift correction, but the methods required are different (see e.g., Zhang & Hornof, 2011, 2014). Secondly, we only consider the problem of post-hoc correction; vertical drift can also be corrected in real time, but this imposes a more restrictive set of constraints that are better handled by other types of algorithm (see e.g., Hyrskykari, 2006; Palmer & Sharif, 2016). Thirdly, we only consider fully automated algorithms that do not require human supervision; an alternative approach would make greater use of human insight.

The paper proceeds in four main sections. First, we outline the algorithms. Second, we test the algorithms on simulated fixation sequences afflicted with various types of measurement error. Third, we test the algorithms on an eye tracking dataset (two examples from which are presented in Fig. 1). And finally, we discuss the major properties of the algorithms, provide guidance to researchers about their use, and suggest ways in which they can be improved further. All code and data required to reproduce the analyses reported in this paper, as well as Matlab/Octave, Python, and R implementations of the algorithms, are available from the public data archive associated with this paper: https://doi.org/10.17605/OSF.IO/7SRKG

Algorithms

In this section, we describe nine algorithms for the automated, post-hoc correction of vertical drift. The reader may also wish to refer to the Appendix where the algorithms are presented in pseudocode alongside other technical details.

Attach

The `attach` algorithm is the simplest of the algorithms considered in this paper. The algorithm simply attaches each fixation to its closest line, as illustrated in Fig. 2A. While this has the benefit of being extremely simple, it is generally not resilient to the kinds of drift phenomena described above. However, `attach` serves as a useful baseline algorithm, since it essentially corresponds to an eye tracking analysis in which no correction was performed—a standard analysis of eye tracking data would simply map fixations to the closest words or other areas of interest. We return to this point later in the paper.

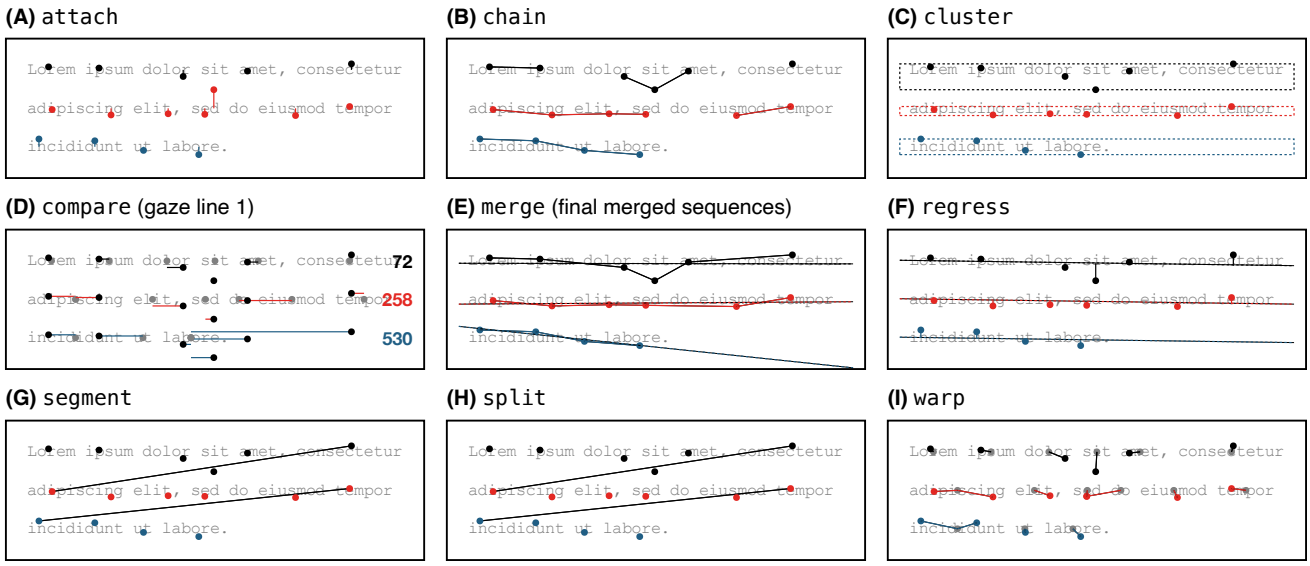

Figure 2
Illustrations of the core principle of each algorithm. The colors represent which line each fixation was assigned to (black indicates line one, etc.). `attach` maps the fixation on "sit" to line 2, while all other algorithm correctly map it to line 1.

Chain

The `chain` algorithm is based closely on one of the methods implemented in the R package `popEye` (Schroeder, 2019) and can be seen as an extension of `attach`. Fixations are first linked together into "chains"—sequences of consecutive fixations that are within a specified x and y distance of each other (see Fig. 2B for an example). Fixations within a chain are then attached to whichever line is closest to the mean of their y values. This procedure is similar to the methods reported by Hyrskykari (2006) and Mishra et al. (2012), so we consider these to be special cases of `chain`.

The `chain` algorithm generally provides better performance over `attach` by exploiting the sequence's order information, as illustrated in Fig. 2B: The fixation on "sit" on line 1 is chained to two surrounding fixations and is correctly mapped to line 1; while `attach` assigns this fixation to line 2 to which it is technically closer. A disadvantage of the method, however, is that it is necessary to specify appropriate thresholds that determine when a new chain begins. If these thresholds are set too low, `chain` becomes equivalent to `attach`; if they are set too high, `chain` will group large numbers of fixations together and force them onto a single inappropriate line. By default, `popEye` sets the x threshold to 20 $\times$ the font height and the y threshold to 2 $\times$ the font height. It is not exactly clear how these defaults were chosen, but we would tentatively suggest that the x threshold should be set to approximately one long saccade length (we use 192 px), and the y threshold to around half a line height (we use 32 px).

Cluster

The `cluster` algorithm is also based on one of the methods implemented in `popEye` (Schroeder, 2019). `cluster` applies k-means clustering[1] to the y values of all fixations in order to group the fixations into m clusters, where m is the number of lines in the passage. Once each fixation has been assigned to a cluster, clusters are mapped to lines based on the mean y values of their constituent fixations: The cluster with the smallest mean y value is assigned to line one and so forth.

Unlike `attach` and `chain`, `cluster` does not assign fixations to the closest line in absolute terms; instead, it operates on the principle that fixations with similar y values must belong to the same line regardless of how far away that line might be. As such, the algorithm generally handles drift issues quite well. However, `cluster` will often not perform well if there is even quite mild overlap between fixations from different lines. In addition, since k-means clustering is not guaranteed to converge on the same set of clusters on every run, the `cluster` algorithm is nondeterministic and can be somewhat unpredictable across multiple runs on the same reading trial. This is an important consideration from the point of view of reproducible research output.

Compare

The `compare` algorithm is based on the method reported by Lima Sanches et al. (2015). The fixation sequence is first segmented into "gaze lines" by identifying the return sweeps—long saccades that move the eye from the end of one line to the start of the next. The algorithm considers any saccade that moves from right to left by more than some threshold value (we use 512 px) to be a return sweep. Gaze lines are then matched to text lines based on a measure of similarity between them.

Lima Sanches et al. (2015) considered three measures of similarity and found dynamic time warping (DTW; Sakoe & Chiba, 1978) to be the best method. DTW returns a measure of how dissimilar two sequences are, so the `compare` algorithm simply matches each gaze line to whichever text line results in the smallest DTW distance. In the example in Fig. 2D, we only depict the first gaze line (black dots), which is compared to each line of text; the best match is with the first text line, as reflected in its low cost (i.e., the sum of the mapping lines is minimized).

The gaze lines and text lines are only compared in terms of their x values under the assumption that the fixations in a gaze line should have a good horizontal alignment with the centers of the words in the corresponding text line. Relying only on the x values helps the algorithm overcome vertical drift issues, but it is also problematic because the lines of text in a passage tend to be horizontally similar to each other; each line tends to contain a similar number of words that are of a similar length, resulting in potential ambiguity about how gaze lines and text lines should be matched up. To alleviate this issue, Lima Sanches et al. (2015) only compare the gaze line to a certain number of nearby text lines, controlled by a free parameter (we use 3).

[1] We used the `KMeans` function from the Python library `scikit-learn` (Pedregosa et al., 2011). There are many variations of k-means clustering, which we have not systematically compared.

Merge

The `merge` algorithm is closely based on the post-hoc correction method described by Špakov et al. (2019). The algorithm begins by creating "progressive sequences"—consecutive fixations that are sufficiently close together. This is similar to `chain`, except that the sequences are strictly progressive (they only move forward), so a regression will initiate a new progressive sequence. The original method uses several parameters to define what constitutes "sufficiently close together," but here we simplify this down to a single parameter, the `y_thresh` which determines how close the y values of two consecutive fixations must be to be considered part of the same progressive sequence (we use 32 px).

Once these sequences have been created, they are repeatedly merged into larger and larger sequences until the number of sequences is reduced to m—one for each line of text. On each iteration of the merge process, the algorithm fits a regression line to every possible pair of sequences (with the proviso that the two sequences must contain some minimum number of fixations). If the absolute gradient of the regression line or its error (root-mean-square deviation) is above a threshold (we use 0.1 and 20 respectively), the candidate merger is abandoned. The intuition here is that, if two sequences belong to the same text line, the regression line fit to their combined fixations will have a gradient close to 0 and low regression error. Of the candidate mergers that remain, the pair of sequences with the lowest error are merged and added to the pool of sequences, replacing the original two sequences and reducing their number by one. This process is repeated until no further mergers are possible.

The algorithm then enters the next "phase" of the process, in which the criteria are slightly relaxed, allowing more mergers to occur. These phases could in principle be defined by the user, but we follow the four-phase model reported by Špakov et al. (2019), which effectively builds a set of heuristics into the algorithm. In Phase 1, the first and second sequences must each contain a minimum of 3 fixations to be considered for merging; in Phase 2, only the second sequence must contain a minimum of 3 fixations; in Phase 3, there is no minimum number of fixations; and in Phase 4, the gradient and regression error criteria are also entirely removed. Of course, as soon as the number of sequences is reduced to m the algorithm exits the merge process, so not all four phases will necessarily be required. Finally, the set of m sequences is matched to the set of text lines in positional order: The sequence with the smallest mean y value is mapped to line one and so forth.

A similar sounding method is reported by Beymer and Russell (2005) whose technique is based on "growing" a gaze line by incrementally adding fixations until this results in a poor fit to a regression line, at which point a new gaze line is begun. However, the description of the method lacked sufficient detail for us to consider it further.

Regress

The `regress` algorithm, which is closely based on Cohen's (2013) R package `FixAlign`, treats the fixations as a cloud of unordered points and fits m regression lines to this cloud. These regression lines are parameterized by a slope, vertical offset, and standard deviation, and the best parameters are obtained by minimizing[2] an objective function that

[2]We used the `minimize` function from the Python library `SciPy` (Virtanen et al., 2020). We have not systematically compared the choice of optimizer settings.

determines the overall fit of the lines through the fixations. The algorithm has six free parameters which are used to specify the lower and upper bounds of the slope, offset, and standard deviation. Here we directly adopt `FixAlign`'s defaults: $[-0.1, 0.1]$, $[-50, 50]$, and $[1, 20]$ respectively. Once the m best-fitting regression lines are obtained (e.g., the dashed lines in Fig. 2F), `regress` assigns each fixation to the highest-likelihood regression line, which in turn is associated with a text line.

`regress` has some conceptual similarities with `merge` but differs in several important respects. Notably, `regress` takes a top-down approach, where `merge` is more bottom-up, and the regression lines that `regress` fits to the fixations cannot take independent values—it is assumed that all fixations are sloping in the same direction, with the same vertical offset, and with the same amount of within-line variance. In addition, unlike `merge`, `regress` does not utilize the order information; instead, like `cluster`, it views the fixations as a collection of unordered points.

Segment

The `segment` algorithm is a slight simplification of the method described by Abdulin and Komogortsev (2015). The fixation sequence is first segmented into m disjoint subsequences based on the $m - 1$ most extreme backward saccades along the x-axis (i.e., the saccades that are most likely to be return sweeps). These subsequences are then mapped to the lines of text chronologically, under the assumption that the lines of text will be read in order. Abdulin and Komogortsev (2015) do not state precisely how they identify the return sweeps, but it seems they potentially allow for more than m subsequences to be identified, in which case, rereadings of a previous line, based on a threshold level of similarity, are discarded. The version of the algorithm considered here does not discard any fixations and instead always identifies exactly m subsequences.

The advantage of this general approach, as emphasized by Abdulin and Komogortsev (2015), is that the y values of the fixations are completely ignored, rendering any vertical drift entirely invisible to the algorithm. However, the approach does not allow for the possibility that the lines of text might be read out of order or that a line of text might be read more than once, which is not uncommon in normal reading behavior. Therefore, the great strength of `segment`—its identification of m consecutive subsequences, permitting a chronological, as opposed to positional, mapping—is also its great weakness: If a large regression is mistakenly identified as a return sweep, this will lead to a catastrophic off-by-one error in subsequent line assignments.

Split

As far as we know, the `split` algorithm takes an approach that is distinct from anything previously reported, although it is conceptually similar to `segment`. Like `segment`, the `split` algorithm works on the principle of splitting the fixation sequence into subsequences by first identifying the return sweeps. However, `split` is not restricted to finding exactly $m - 1$ return sweeps; instead, it identifies the most likely set of return sweeps, however many that turns out to be. There are various ways of approaching this classification problem, but here we use k-means clustering to partition the set of saccades into exactly two clusters. Since return sweeps are usually highly divergent from normal saccades, one of the

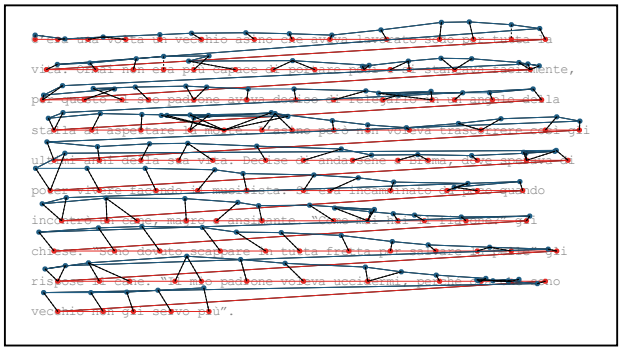

Figure 3
Illustration of the `warp` algorithm. The veridical fixation sequence is represented in blue, and the expected fixation sequence (the sequence of word centers) is represented in red. The dashed black lines show the DTW warping path—the optimal way to align the two sequences, such that the sum of the Euclidean distances between matched points (i.e., the sum of the dashed lines) is minimized.

two clusters will invariably contain the return sweeps, which can then be used to split the fixation sequence into subsequences. However, since this is not guaranteed to produce $m-1$ return sweeps (and therefore m subsequences) an order-based mapping is not possible, so `split` must use absolute position: Subsequences are mapped to the closest text lines in absolute terms. `split` has the advantage of generally finding all true return sweeps, and even if it identifies some false positives, the resulting subsequences can still be mapped to the appropriate lines by absolute position. However, this also means the algorithm is less resilient to vertical drift issues.

Warp

The final algorithm we consider, `warp`, is also novel to this paper, although it is mostly a wrapper around a preexisting algorithm—dynamic time warping. DTW was used by Lima Sanches et al. (2015) to provide a measure of dissimilarity between a gaze line and a text line in the `compare` algorithm. To our knowledge, however, there have been no previous reports of DTW being used for its primary purpose: finding the best alignment between two time sequences. This is somewhat surprising because DTW is the natural computational choice for tackling vertical drift. Two methods have previously been described that share some conceptual similarities. Carl (2013) uses a basket of reading-related measures to place a cost on different paths through a lattice of fixation-to-character mappings and selects the path with minimal cost. Martinez-Gomez et al. (2012) convert the fixation sequence into a low resolution image and find the best transformation of this image to fit the overall shape of the passage. Both of these methods are quite convoluted, however, and we consider them to be special cases of `warp`, which is a direct application of the standard DTW algorithm to eye tracking data.

In order to use DTW to realign the fixation sequence to the text, we first need to specify an expected fixation sequence. Since we expect the reader to traverse the passage from left to right and from top to bottom, we can use the series of word centers as the expected sequence, under the assumption that readers will target the centers of words

(O'Regan et al., 1984). Given the expected and veridical sequences as inputs, the DTW algorithm finds the optimal way to nonlinearly warp the sequences on the time dimension such that the overall Euclidean distance between matched points across the two sequences is minimized, while maintaining a monotonically increasing mapping.[3] In the "warping path" that results from this process, every fixation is mapped to one or more words and every word is mapped to one or more fixations (see Fig. 3 for an example). It is then simply a case of assigning each fixation to whichever line its mapped word(s) belong(s) to. In the unlikely event that the mapped words belong to different lines, the majority line wins or an arbitrary choice is made in the case of ties.

If the final fixation on line i were mapped to the first word on line $i+1$, this would result in a large increase in the overall cost of the mapping, so line changes act as major clues about the best alignment. The upshot of this is that `warp` effectively segments the fixation sequence into exactly m subsequences, which are mapped to the lines of text in chronological order. In this sense, `warp` behaves very much like `segment`. However, the additional benefit of `warp` is that it can simultaneously consider different possibilities about which saccades are the return sweeps, selecting only those that result in the best fit to the passage at a global level. Nevertheless, `warp` is ultimately limited by the veracity of the expected fixation sequence, which encodes one particular way of reading the passage—line by line from start to end. If the reader deviates from this assumption, `warp` can fail to correctly assign fixations to lines.

Summary

In this section we have described nine algorithms for aligning a fixation sequence to a multiline text, each of which takes a fundamentally different approach. A summary of the information utilized by the algorithms is provided in Table 1; each algorithm uses at least one piece of information about the fixations and at least one piece of information about the passage, and some also rely on additional parameters set by the user or built-in heuristics.

Broadly speaking, the algorithms proceed in three stages: grouping, assignment, and update. In the grouping stage, the fixations are, in some sense, placed into groups, the one exception being `attach` which has no grouping stage. The rational for how a group is formed varies by algorithm, but in general the algorithms can be categorized into those that are guaranteed to produce m groups (i.e., one per text line; `cluster`, `merge`, `regress`, `segment`, and `warp`) and those that are not (`attach`, `chain`, `compare`, and `split`).

In the assignment stage, the previously identified groups are assigned to text lines. If the grouping stage is *not* guaranteed to produce m groups, then assignment must be based on absolute position (or similarity in the case of `compare`, although it still uses absolute position to select neighboring lines to compare to). If the grouping stage *is* guaranteed to produce m groups, then they can be assigned to text lines based on order; this generally allows for better handling of vertical drift because absolute position is ignored. In the case of `cluster`, `merge`, and `regress`, which produce unordered groups at the grouping stage, groups are matched to text lines based on the order in which they are positioned vertically

[3] Specifically, the first fixation must be mapped to (at least) the first word; the last fixation must be mapped to (at least) the last word; every other fixation must be mapped to at least one word; and, if fixation i is mapped to word j, then fixation $i+1$ must be mapped to word(s) $\geq j$. And vice versa for the mapping from words to fixations.

Table 1
Information Utilized by the Algorithms

	Fixation Info.			Passage Information			Other Information	
Algorithm	X	Y	Order	No. Lines	Line Y	Word X	Params.	Heuristics
`attach`		✓			✓			
`chain`	✓	✓	✓		✓		2	
`cluster`		✓		✓				
`compare`	✓	✓	✓		✓	✓	2	
`merge`	✓	✓	✓	✓			3	✓
`regress`	✓	✓		✓	✓		6	
`segment`	✓		✓	✓				
`split`	✓	✓	✓		✓			
`warp`	✓	✓	✓		✓	✓		

Table 2
Summary of the Grouping and Assignment Stages of each Algorithm

Algorithm	Grouping Stage	Assignment Stage
`attach`	N/A	Assign fixations to closest text lines
`chain`	Chain consecutive fixations that are sufficiently close to each other	Assign chains to closest text lines
`cluster`	Classify fixations into m clusters based on their y values	Assign clusters to text lines in positional order
`compare`	Split fixation sequence into subsequences based on saccades that are longer than a threshold	Assign subsequences to text lines by measuring horizontal similarity with the words in neighboring text lines
`merge`	Form a set of progressive sequences and then reduce the set to m by repeatedly merging those that appear to be on the same line	Assign merged sequences to text lines in positional order
`regress`	Find m regression lines that best fit the fixations and group fixations according to best fit regression lines	Assign groups to text lines in positional order
`segment`	Segment fixation sequence into m subsequences based on $m-1$ most-likely return sweeps	Assign subsequences to text lines in chronological order
`split`	Split fixation sequence into subsequences based on best candidate return sweeps	Assign subsequences to closest text lines
`warp`	Map fixations to word centers by finding a monotonically increasing mapping with minimal cost, effectively resulting in m subsequences	Assign fixations to the lines that their mapped words belong to, effectively assigning subsequences to text lines in chronological order

(i.e., mean y value). In the case of `segment` and `warp`, the groups are assigned to text lines in chronological order, which is only possible because these two algorithms produce subsequences that inherit the order of the original fixation sequence. An overview of the grouping and assignment methods is provided in Table 2 for quick reference.

Finally, in the update stage, the original fixation sequence is modified to reflect the line assignments identified in the previous stage. In the versions of the algorithms reported in this paper, we always use the same update approach: The y values of the fixations are adjusted to the y values of the assigned lines, while the x values and the order of the fixations are always left untouched. In principle, however, there are other ways of performing the update stage (e.g., retaining the original y-axis variance or discarding ambiguous fixations).

Performance on Simulated Data

We now test the ability of each algorithm to correctly recover the intended lines from simulated fixation sequences. These fixation sequences are simulated with particular characteristics, allowing us to understand how the algorithms respond to specific, isolated phenomena.

Method

In each simulation, we generate a passage of "Lorem ipsum" dummy text consisting of between 8 and 12 lines with up to 80 characters per line and 64 px of line spacing. We then generate a fixation sequence consisting of one fixation for every word in the passage: The x value of a fixation (f_x) is set randomly within the word; the y value of a fixation (f_y) is calculated according to:

$$f_y = \mathcal{N}(l_y, d_{\text{noise}}) + f_x d_{\text{slope}} + l_y d_{\text{shift}}, \tag{1}$$

where l_y is the vertical center point of the intended line—the y value that the reader is targeting. This models three types of distortion: noise, slope, and shift. Additionally, we simulate two types of regression that are characteristic of normal reading behavior and can disrupt algorithmic correction. Together, these five phenomena are illustrated in Fig. 4 and described below.

Noise Distortion

The noise distortion parameter, d_{noise}, controls the standard deviation of the normally distributed noise around the intended line and represents imperfect targeting by the reader and/or measurement error. In our exploration of this parameter, we use values of $d_{\text{noise}} = 0$, representing no noise, through $d_{\text{noise}} = 40$, representing extreme noise. The noise parameter is also a proxy for line spacing (raising the noise level effectively corresponds to tightening the line spacing), so this parameter also provides an indication of how the algorithms will perform under different degrees of line spacing.

Slope Distortion

The slope distortion parameter, d_{slope}, controls the extent to which fixations progressively move downward as the reader moves from left to right across the passage; fixations

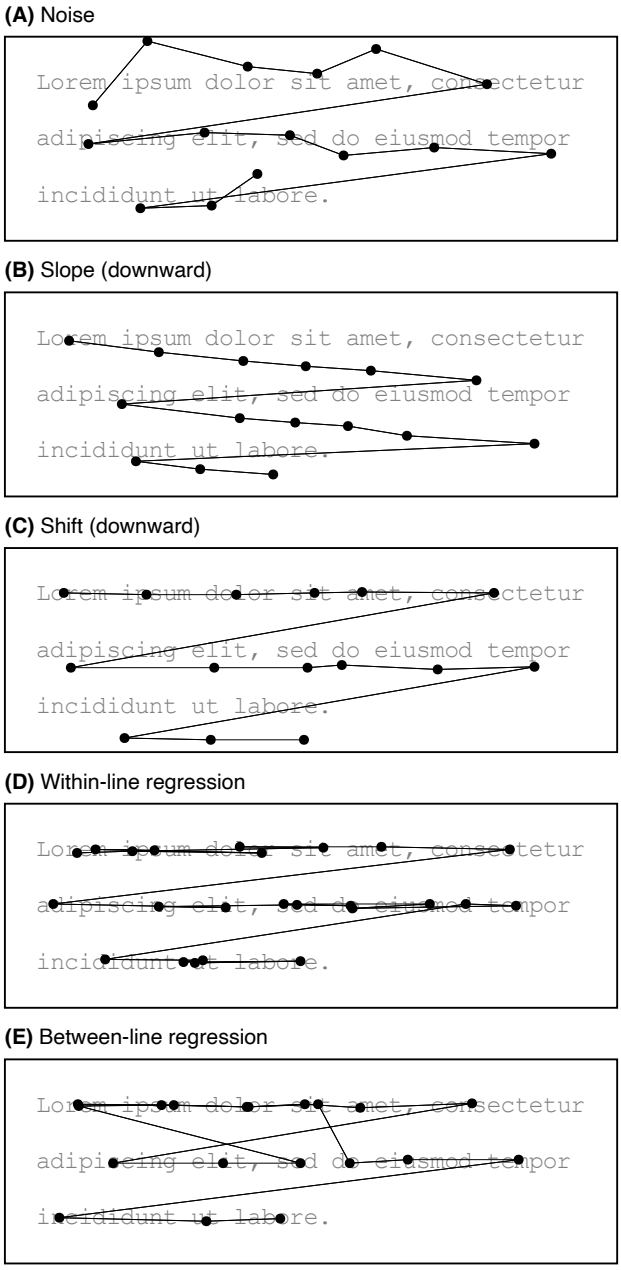

Figure 4
Example simulated fixation sequences under five phenomena considered in this paper. The algorithms must overcome these phenomena in order to correctly infer the intended line of each fixation.

on the left edge of the passage will be correctly located, but for every one pixel the reader moves to the right, the fixations will drift downward by d_{slope} pixels. Unlike noise, this is solely attributable to measurement error. In our exploration of this parameter, we use values of $d_{\text{slope}} = -0.1$, representing extreme upward slope, through $d_{\text{slope}} = 0.1$, representing extreme downward slope.

Shift Distortion

The shift distortion parameter, d_{shift}, controls the extent to which fixations progressively move downward as the reader moves from one line to the next; fixations on the first line will be correctly located, but for every one pixel of intentional downward movement, the fixations will drift downward by a further d_{shift} pixels. Like slope, this represents systematic measurement error. Our exploration of this parameter uses values of $d_{\text{shift}} = -0.2$, representing extreme upward shift, through $d_{\text{shift}} = 0.2$, representing extreme downward shift.

Within-Line Regression

As mentioned above, we also consider the effects of two types of regression. The first of these is within-line regressions, which is where the reader momentarily jumps back to a previous point in the current line. The extent to which the reader performs within-line regressions is formalized by a probability. If this probability is set to 1, the reader will perform a regressed fixation after every normal fixation, doubling the number of fixations on the line; if the parameter is set to 0, the reader will never perform a regression within the line. The x position of the regressed fixation is located randomly between the start of the line and the current fixation with longer regressions being linearly less probable than shorter regressions. The y value of the regressed fixation follows Equation 1.

Between-Line Regression

The second type of regression, between-line regressions, is where the reader rereads text from a previous line. Between-line regressions are expressed in terms of the probability that the reader will go back to a previous line at some point during reading of the current line. Once the regression is completed, the reader returns to the point in the passage before the regression occurred. If the parameter is set to 1, the reader will reread part of a previous line for every line they read; if it is set to 0, the reader will never perform a regression to a previous line. When a between-line regression occurs, the previous line is determined randomly, with more recent lines linearly more probable than less recent lines; the section of the previous line is determined randomly by two uniformly distributed x values. The y values of the regressed fixations follow Equation 1.

Results

For each phenomenon, we ran 100 simulations for each of 50 gradations in the parameter space, and each of these 5000 simulated reading scenarios was corrected by all nine algorithms. Accuracy is measured as the percentage of fixations that were correctly mapped back to the target line. Before describing the results, there are two important things to note. Firstly, the extreme values we have chosen for each phenomenon are arbitrary, so

the algorithms should only be compared within, and not across, phenomena.[4] Secondly, for the algorithms that have free parameters (`chain`, `compare`, `merge`, and `regress`), we use the default parameter settings defined above. We have not systematically manipulated the parameter settings because (a) this would result in an explosion in the number of algorithm/parameter combinations that we must consider, (b) manipulating a parameter to deal with one phenomenon can have unexpected consequences for other phenomena,[5] and (c), in a sense, these algorithms ought to incur a penalty for not being parameter-free.

Performance on Noise

Results for the noise distortion parameter are shown in Fig. 5A. Under zero noise, all algorithms perform at 100% accuracy, but five of the algorithms are adversely affected by noise when it reaches a sufficiently high level of around 10: Of these, `chain` performs best, closely followed by `attach`, then `cluster` and `regress`, and finally `merge`. Of the remaining algorithms, `compare` and `split` are highly resilient to noise, while `segment` and `warp` are entirely invariant.

Performance on Slope

In terms of slope distortion (Fig. 5B), when the parameter is set to zero, all algorithms perform perfectly, but as the slope becomes more extreme (in either the upward or the downward direction), four of the algorithms experience a sustained loss in accuracy. Of these, `cluster` generally performs best and, initially at least, `attach` performs worst; `chain` and `split` initially perform better than `attach`, but are eventually outperformed. Interestingly, although `regress` is mostly resilient to slope, it has two weak spots around the values of -0.03 and 0.03. When the slope takes one of these values, `regress` struggles to disambiguate between (a) zero offset combined with the appropriate slope and (b) a large offset combined with slope in the opposite direction; if it selects the wrong option, fixations on one half of the passage will be misaligned, causing a substantial drop in accuracy. This reveals a hidden weakness of the `regress` algorithm, and we will see an example of it later. Of the remaining algorithms, `compare` and `merge` are highly resilient to slope, while `segment` and `warp` are invariant.

Performance on Shift

In terms of shift (Fig. 5C), when the parameter is zero, all algorithms perform perfectly, but as it becomes more extreme, four of the algorithms—`attach`, `chain`, `regress`, and `split`—drop in accuracy. In fact, `attach`, `chain`, and `split` produce identical results in the case of shift because they are all fundamentally reliant on absolute position. `compare` is mostly resilient to shift, while the remaining four algorithms—`cluster`, `merge`, `segment`, and `warp`—are invariant.

[4] For example, `regress` appears to have worse performance on shift compared to slope; however, if we had simulated a narrower range of shift values, the results might have led us to the opposite conclusion.

[5] For example, if the standard deviation bounds of `regress` are widened, it may be possible to improve performance on noise, but the algorithm will be less capable of dealing with slope.

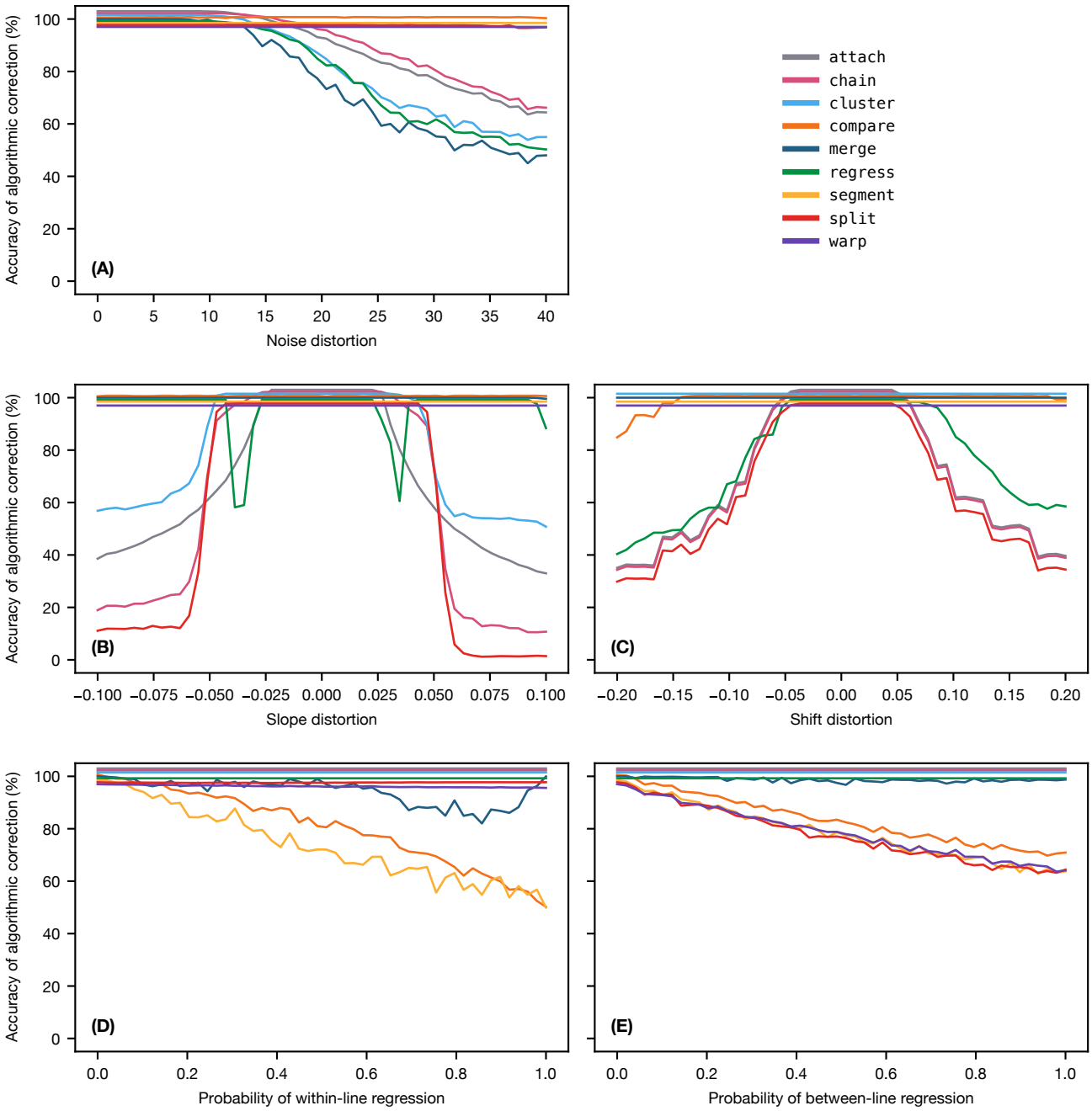

Figure 5
Mean accuracy of the nine algorithms in response to the five eye tracking phenomena. For example, some algorithms (`attach`, `chain`, `cluster`, `regress`, and `merge`) are adversely affected as the noise level is increased, while the other algorithms are either resilient to noise (`compare` and `split`) or entirely invariant to noise (`segment` and `warp`). The plotted lines have been vertically staggered to aid visualization.

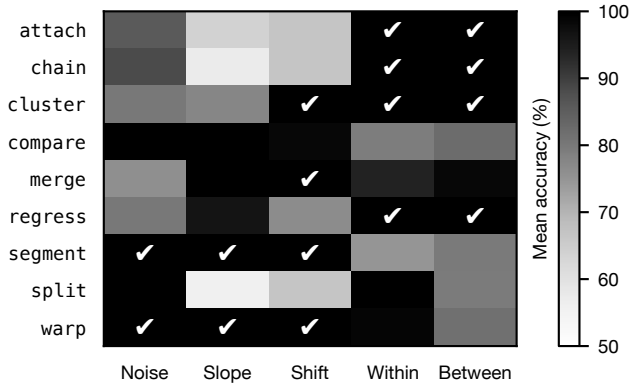

Figure 6
Mean accuracy of the algorithms for each of the eye tracking phenomena. Darker cells indicate phenomena that an algorithm performs well on. A checkmark indicates that the algorithm is entirely invariant to the phenomenon in question, scoring 100% in all 5000 simulations.

Performance on Within-Line Regression

Results for within-line regressions are shown in Fig. 5D. When there are no within-line regressions, all algorithms perform at 100%, but three of the algorithms drop off as the probability of within-line regression is increased. Of these, `compare` and `segment` track each other quite closely because they rely on identifying the return sweeps; `merge` is generally quite resilient, except when the parameter is around 0.7–0.9 because these values cause a large number of progressive sequences to be generated which cannot then be merged very freely, so the merge process tends to get trapped in local minima (i.e., bad mergers that happen early on cannot later be reverted). Of the remaining algorithms, `split`[6] and `warp` are highly resilient, while `attach`, `chain`, `cluster`, and `regress` are invariant.

Performance on Between-Line Regression

In terms of between-line regressions (Fig. 5E), four algorithms are negatively impacted by increases in this parameter. Of these, `compare` and `split` can in principle find more than m gaze lines, but they have difficulties identifying when a between-line regression occurs, while `segment` and `warp` are limited to identifying exactly m gaze lines in strictly sequential order, so they fundamentally cannot handle between-line regressions. Of the remaining algorithms, `merge` is resilient to between-line regressions, while `attach`, `chain`, `cluster`, and `regress` are entirely invariant.

Summary

In this section, we have simulated five eye tracking phenomena that are particularly relevant to understanding the performance characteristics of the algorithms. Fig. 6 summarizes how accurately the algorithms perform on each phenomenon. No single algorithm

[6]Unlike `compare` and `segment`, even if `split` misidentifies a regression as a return sweep, it will still be able to map the resulting gaze line to the appropriate text line because it assigns based on position rather than order.

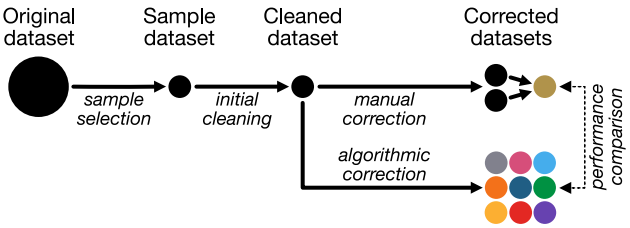

Figure 7
Pipeline for testing the algorithms on a natural dataset. The original dataset was first reduced to a smaller sample, which then underwent some initial cleaning steps. This cleaned sample dataset was then corrected by the nine algorithms and two human correctors (whose combined correction constitutes the gold standard). Performance is measured by how closely the algorithmic corrections match the gold standard correction.

is invariant—or even resilient—to all phenomena, although `merge` and `warp` come quite close: `merge` is only weak on noise, while `warp` is only weak on between-line regressions. In general, there tends to be a tradeoff between how well an algorithm can handle distortion and how well it can handle regressions; the ability to deal with one tends to come at the cost of the other. Nevertheless, in real world scenarios, performance will very much depend on the degree and relative prevalence of the phenomena. Furthermore, there may be other forms of measurement error and reading behavior that we have neglected to consider here, and those that we have considered are likely to interact in complex, unpredictable ways. It is therefore important to test the algorithms against natural eye tracking data to get a more holistic understanding of their performance.

Performance on Natural Data

In this section, we test the algorithms against an eye tracking dataset that has been manually corrected by human experts. Unlike the simulations, there is no ground truth, and we cannot isolate particular phenomena; however, the benefit of this approach is that the phenomena are combined in a realistic way, allowing us to estimate how well the algorithms are likely to perform in real-world scenarios.

Method

We tested the algorithms on an eye tracking dataset collected by Pescuma et al. (in prep.), which includes reading data for both adults and children, allowing us to test the algorithms on two distinct populations. Our general approach is illustrated in Fig. 7 and discussed over the following sections.

The Dataset

Pescuma et al. (in prep.) collected eye tracking data for 12 passages from Italian children's stories (e.g., a passage from *Goldilocks*). The passages were around 130 words in length, spanning 10–13 lines and were presented in 20-point Courier New (each character occupying around 0.45 degrees of visual angle). Either of two sets, each comprised of six passages, was administered for silent reading to a large sample of children aged 8–11

(N = 140) and a smaller sample of adult controls (N = 33) for a total of 877 reading trials. Eye movements were recorded using a tower mounted EyeLink 1000 Plus eye tracker (SR Research, Toronto, Canada) for which a typical accuracy of 0.25–0.50 degrees is reported by the manufacturer. Recording was monocular (right eye) with a 1000 Hz sampling rate.

Selection of the Sample for Manual Correction

Since is was impractical to manually correct all 877 trials (to do so would require months of work), we selected a sample for manual correction. For each of the 12 passages, we selected two reading trials by adult participants and two reading trials by child participants, for a total of 48 trials (5.5% of the full dataset). The reading trials were selected pseudorandomly such that no single participant was represented more than once. Additionally, we manually checked and adjusted the sample to ensure it contained an equal balance of easy and challenging cases, as well as examples of all the various eye tracking phenomena discussed previously.

Initial Data Cleaning

We performed two initial cleaning steps in order to isolate the core problem of line assignment from two extraneous issues. Firstly, any fixation that was located more than 100 px from any character in the passage was removed (i.e., out-of-bounds fixations that occur in the margins or off-screen). This is because the algorithms are not designed to detect and discard these fixations, and such cases can hinder their ability to match fixations to the appropriate lines. Secondly, prior to reading a passage and on its completion, a reader's fixations will typically jump around the text unpredictably; again, since the algorithms are not designed to automatically discard such fixations, we manually removed any such cases from the starts and ends of the fixation sequences, allowing the algorithms to concentrate on the core problem of assigning fixations to lines.

Manual Correction Procedure

The cleaned sample dataset was corrected independently by two human correctors (JWC and VNP). To perform the correction, each corrector studied plots of the participants' fixation sequences and recorded, fixation by fixation, which line each one belonged to, guided by fixation position, saccade trajectories, textual cues, and fixation duration, as well as general knowledge of eye tracking and reading behavior. Unlike the algorithms, the human correctors also had the option to discard fixations as they saw fit.

Across the 48 reading trials, the correctors initially disagreed on 299 of 10,245 fixations (2.9%). Of these 299 disagreements, only 15 related to which line a fixation was assigned to; on inspection, all 15 cases turned out to be human error on the part of one corrector or the other. The other 284 disagreements related to whether or not a fixation should be discarded; following discussion of these cases, the correctors reached consensus about how these fixations should be treated. This resulted in a single manual correction, which we consider to be the gold standard against which the algorithms can be evaluated. In this gold standard correction, a total of 255 fixations were discarded across all 48 trials (2.5%; 5.3 fixations per trial).

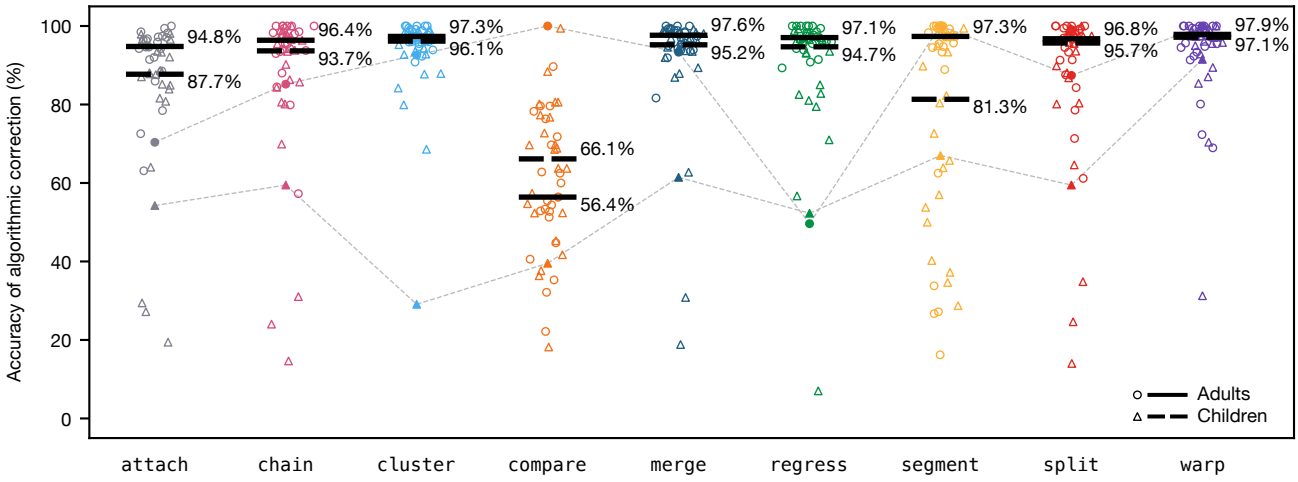

Figure 8
Accuracy of the algorithms on adult reading trials (circles) and child reading trials (triangles). The y-axis measures the percentage of fixations assigned to the correct line, as defined by the gold standard manual correction. The filled points, linked together by dashed lines, correspond to the two example trials illustrated in Figs. 9 and 10. The black bars show median accuracy for the adults (solid bars) and children (broken bars).

It is interesting to note that, although the two correctors had slightly different intuitions about when it was appropriate to discard a fixation, they essentially had perfect agreement about which line a fixation ought to be assigned to if it was retained. This suggests that the correction of vertical drift is actually quite objective—there is usually an unequivocally correct solution to any given trial, although that solution can still be difficult and time-consuming to obtain.

Results

We analyze the performance characteristics of the algorithms in three ways. Firstly, we look at how the algorithms fare against the gold standard manual correction; secondly, we look at how the algorithms perform in comparison to no drift correction at all; and thirdly, we look at how the algorithms relate to each other, regardless of their accuracy.

Accuracy against the Gold Standard

As with the simulations, accuracy is measured as the percentage of fixations that the algorithm mapped to the correct line; the ground truth is defined by the gold standard manual correction. In cases where the correctors chose to discard a fixation, the algorithm is automatically wrong. Fig. 8 plots accuracy on the 48 sample trials by algorithm. The most striking result is `compare` with overall median accuracy of 61.2%, substantially worse than all other algorithms.[7] This contrasts with our simulations, which indicated that `compare` should at least be relatively strong on distortion. The reason for this discrepancy is that the simulated fixation sequences were generated directly from the lines of text with one fixation

[7]This concurs with the 60% accuracy reported by Lima Sanches et al. (2015, p. 1231).

per word, so the artificial gaze lines that `compare` identified tended to be very horizontally similar to the artificial text lines. In the natural dataset, however, this is not the case; when the data contains a lot of natural noise and regressions, gaze lines cannot be reliably matched to text lines based on similarity, even if the set of candidate text lines is narrowed down to the three closest neighbors. Given that `compare` exhibited such poor performance, we consider it to be an algorithmic deadend and do not discuss it any further.

Of the remaining algorithms, median accuracy is typically around 95%, the worst performer being `attach` at 92% and the best performer being `warp` at 97.3%. Accuracy on child trials tends to be lower and more variable than accuracy on adult trials; however, the difference in medians was usually quite small. The major exception to this was `segment` for which median accuracy on adult trials was 97.3%, while median accuracy on child trials was 81.3%, making `segment` one of the best algorithms in terms of adult data but one of the worst in terms of child data. This may be because children tend to perform more regressions (e.g., Blythe & Joseph, 2011) and have more disfluent return sweeps (e.g., Parker et al., 2019), both of which create obstacles for the `segment` algorithm.

Median performance alone conceals the fact that accuracy is often highly variable and long tailed. In the best case scenario, an algorithm will produce a perfect correction that is identical to the gold standard—all algorithms (even `compare`) scored 100% in at least one trial. In the worst case scenario, an algorithm will perform as low as 10–30% accuracy. In addition, the algorithms often differ markedly on particular trials. We have highlighted this in Fig. 8 by singling out two trials, one by an adult and one by a child, which are represented by the filled data points that are linked together with dashed lines. Algorithmic corrections of the adult trial (filled circles) are depicted in Fig. 9 (see Fig. 1A for the original). In this particular case, `compare`, `segment`, and `warp` were able to correctly recover the intended line of every fixation. However, the trial presented problems for some of the other algorithms; in particular, `attach` failed to handle the upward shift in the lower left quadrant of the passage, and `regress` misinterpreted the situation as a case of upward slope, resulting in fixations on the right hand side of the passage being forced down by one line, a potential weakness highlighted by our simulations.

Fig. 10 depicts the algorithmic corrections of the child reading trial (see Fig. 1B for the original), which are represented by the filled triangles in Fig. 8. Performance on this trial is much worse due to the large amount of noise. `cluster`, for example, has struggled to correctly classify the fixations due to the large amount of overlap between fixations intended for adjacent lines, and `segment` has identified one particularly long within-line regression as a return sweep, resulting in some misalignment in the middle of the passage. Only `warp` was able to recover the intended lines for the majority of fixations, and the few errors it did make appear to be cases where the correctors chose to discard some fixations. Overall, these two example trials highlight that, although the algorithms have a similar level of performance on average, performance on a particular trial can be quite divergent depending on its particular characteristics. Corrections of all 48 trials can be found in the supplementary material. [FOR REVIEWERS: https://github.com/jwcarr/vertical_drift/blob/master/visuals/corrections.pdf]

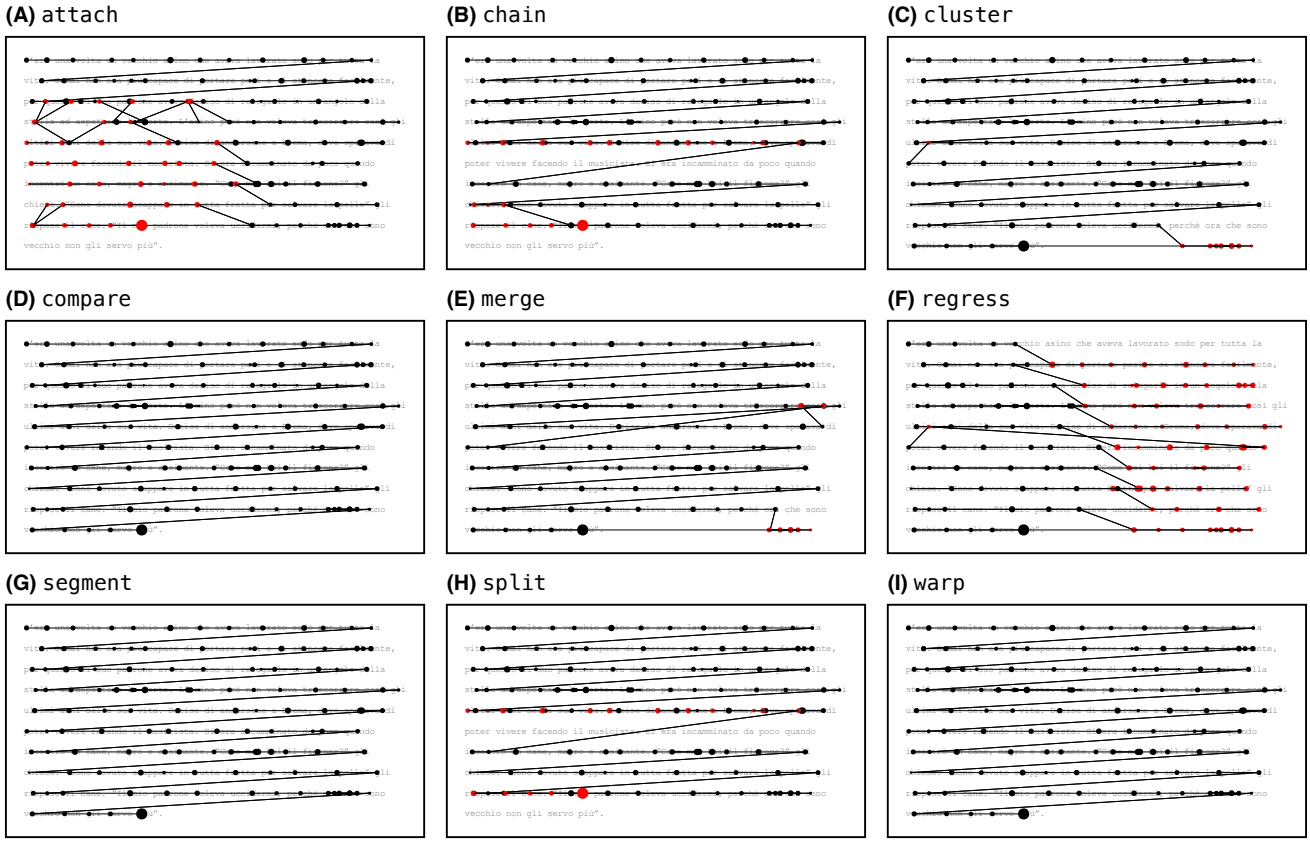

Figure 9
Algorithmic corrections for the adult trial depicted in Fig. 1A. Fixations in red have been assigned to the wrong line. These corrections correspond to the filled circles in Figs. 8 and 11.

Improvement over no Drift Correction

A second, perhaps more useful, way to measure the performance of the algorithms is in terms of how much of an improvement an algorithm provides in comparison to applying no drift collection at all. To estimate this, we first need to define a baseline level of accuracy. As mentioned previously, the `attach` algorithm essentially corresponds to a standard eye tracking analysis; it is equivalent to simply drawing maximal, nonoverlapping bounding boxes around the words in a passage and then mapping fixations to whichever bounding box they fall into (as would be the case in a standard analysis of eye tracking data using the widely adopted Area-of-Interest paradigm). Therefore, we can estimate the potential improvement that a given algorithm offers by comparing its accuracy to the accuracy of the `attach` algorithm.

The results of this analysis are plotted in Fig. 11. The y-axis shows the percentage point increase (or decrease) in accuracy that results from applying vertical drift correction. The zero line represents the baseline of no drift correction (equivalent to `attach`). As before, the datapoints themselves tell us a lot more than the medians. The `chain` and `split` algorithms tend to be quite conservative, while the others tend to have more extreme

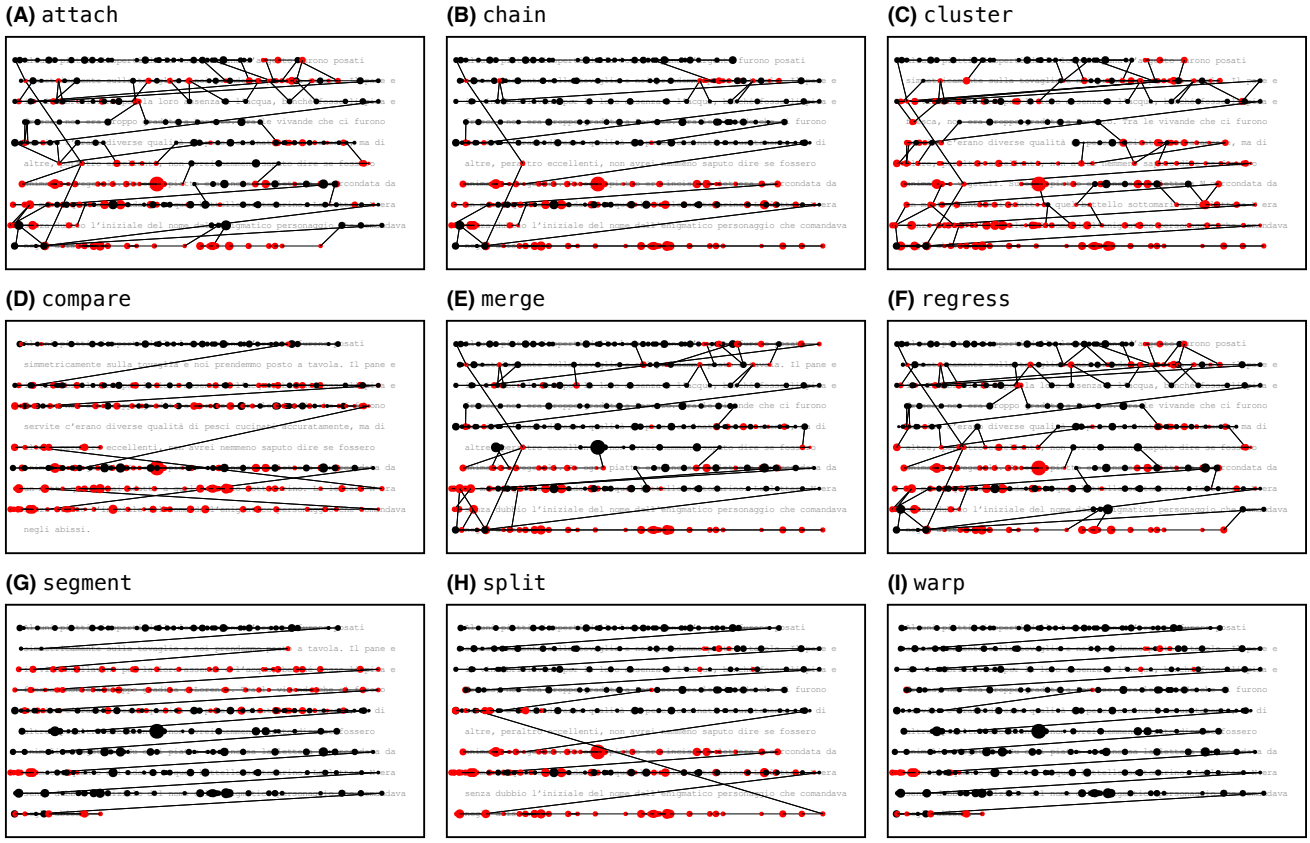

Figure 10
Algorithmic corrections for the child trial depicted in Fig. 1B. Fixations in red have been assigned to the wrong line. These corrections correspond to the filled triangles in Figs. 8 and 11.

effects. In the best case, `cluster` resulted in an 77 percentage point increase in accuracy in comparison to leaving the data uncorrected (i.e., `attach` = 19%, `cluster` = 96%); while in the worst case, `regress` resulted in an 81 percentage point drop in accuracy, badly corrupting the original input data (i.e., `attach` = 88%, `regress` = 7%).

These results highlight that, although in most cases the application of vertical drift correction can improve data quality, the process is not without risk. Furthermore, there is potentially more to gain from applying drift correction to child data, since the baseline level of accuracy tends to be lower to begin with; for example, `warp` offered a modest 2.1 percentage point increase in accuracy on adult data but an 8.2 percentage point increase on child data.

Relationships between Algorithms

As noted above, some algorithms tend to produce very similar output where others produce quite different output. This raises the issue of how the algorithms relate to each other regardless of their performance characteristics on real or simulated data. To investigate this, for each pair of algorithms, we measure the DTW distance between the

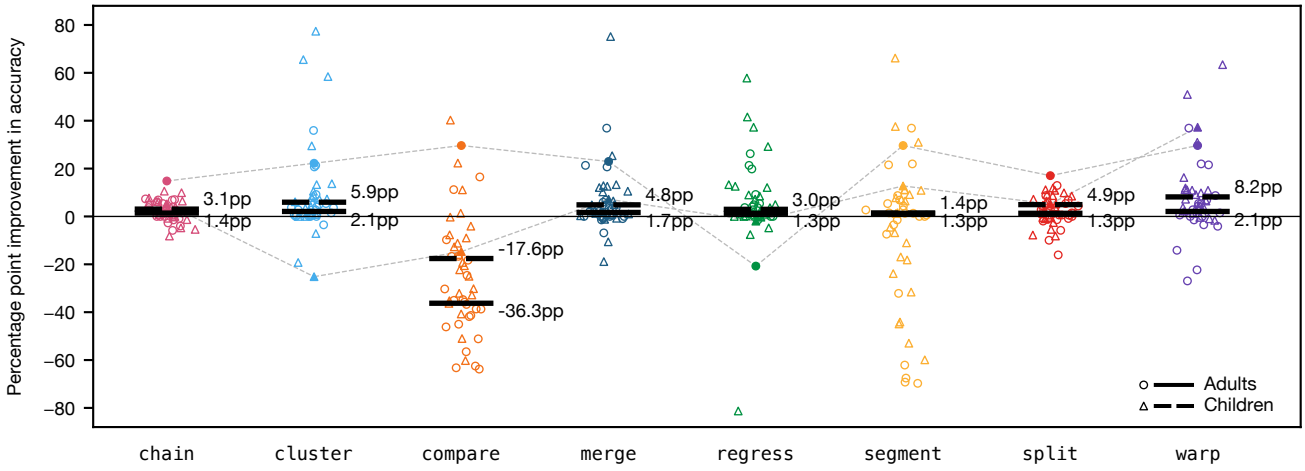

Figure 11
Improvement in accuracy in comparison to performing a standard eye tracking analysis with no drift correction. The y-axis measures the percentage point increase (or decrease) in accuracy beyond the baseline accuracy of the `attach` algorithm. The filled points, linked together by dashed lines, correspond to the two example trials illustrated in Figs. 9 and 10. The black bars show median improvement for the adults (solid bars) and children (broken bars).

two algorithmic corrections of each of the 48 sample trials and take the median distance as an estimate of how dissimilar those two algorithms are.[8] We then analyzed the pairwise distances in two ways.

Firstly, we used agglomerative hierarchical clustering to produce a dendrogram (see Fig. 12A), which provides a taxonomy of the algorithms based on their similarity. The root node represents all algorithms, which initially fork into two major groups. The "sequential algorithms," `segment` and `warp`, both operate on the principle of identifying the return sweeps and mapping the resulting subsequences to the lines of text in sequential order; in other words, their grouping stages can only produce groups consisting of fixations that were arranged consecutively in the original fixation sequence. This means they tend to produce similar outcomes—they both, for example, force fixations onto inappropriate lines in order to preserve sequentiality. Of the "positional algorithms," `split` is the first to branch off, perhaps because—like the sequential algorithms—it leans heavily on the return sweeps, and among the remaining algorithms, there is a clear dichotomy between those that assign based on relative position (`cluster`, `merge`, and `regress`) and those that assign based on absolute position (`attach` and `chain`).

Secondly, we used multidimensional scaling to locate the algorithms in some latent "algorithm space." Fig. 12B shows the output of this analysis projected into two hypothetical dimensions: Algorithms that are close together in this space tend to produce similar results, while algorithms that are far apart tend to produce dissimilar results. The two dimensions of the space appear to roughly correspond to the algorithms' grouping (x-axis) and assignment

[8] Since `compare` was highly divergent from all other algorithms due to its poor performance, it it not included in this analysis.

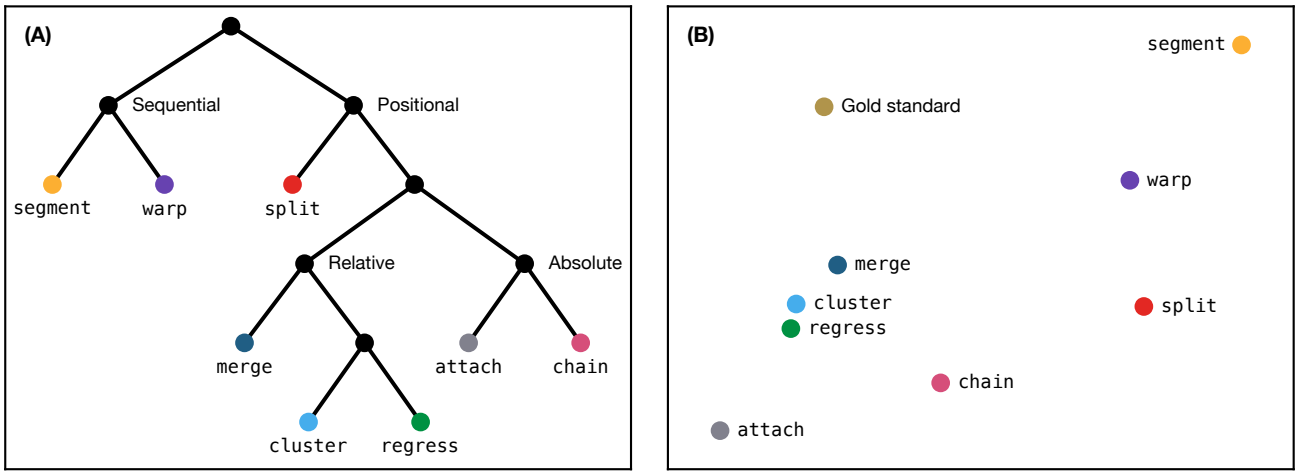

Figure 12
(A) Hierarchical clustering analysis of the algorithmic outputs, providing a taxonomy of the algorithms. (B) Multidimensional scaling analysis of the algorithmic outputs; the distance between two algorithms corresponds to how dissimilar their corrections tend to be, so the space as a whole approximates how the algorithms relate to each other on two hypothetical dimensions.

(y-axis) strategies. `split`, for example, shares its grouping strategy with `segment` (it groups based on return sweeps), but it has an assignment strategy that is more similar to `chain` (it assigns based on absolute position). We also see that `cluster`, `merge`, and `regress` tend to produce quite similar output and are therefore likely to be somewhat interchangeable. Interestingly, the human correctors—represented by the gold standard manual correction—are located in a relatively unexplored part of algorithm space: Their grouping strategy appears to be more similar to `chain` or `merge` (finding local linear clusters), while their assignment strategy seems to be more global and sequential, like `warp`. Anecdotally, this aligns with our experience of performing the manual corrections, and this observation is suggestive of fertile ground for the future development of correction algorithms.

Summary

In this section, we tested the nine algorithms on a real eye tracking dataset. Although `warp` was marginally the most performant algorithm, closely followed by `cluster` and then `merge`, our results indicate that the best algorithm will largely depend on the particular characteristics of a given trial, as well as the general characteristics of the dataset being corrected. All 48 reading trials could be improved by at least one of the algorithms; the difficulty for the researcher, of course, is in knowing which algorithm to apply to a given trial in the absence of a gold standard. We also used two clustering techniques to evaluate how the algorithms relate to each other, regardless of their performance against the gold standard. These analyses provide quantitative evidence regarding how the algorithms tend to behave—that `segment` and `warp` tend to produce similar output, for example.

Discussion

We have identified nine core approaches to the methodological problem of correcting vertical drift in eye tracking data. We instantiated each of these approaches as a simple algorithm that can be evaluated in a consistent and transparent way. Our first analysis using simulated data allowed us to identify which phenomena the algorithms are invariant to and to quantify how the algorithms respond to increasing levels of those phenomena. Our second analysis validated the algorithms on a real eye tracking dataset and allowed us to strengthen our qualitative intuitions about their similarity and main differences. In the remainder of the paper, we sum up what we learned about the algorithms, provide some practical guidance for researchers in the field, and conclude with some thoughts about how vertical drift correction can be improved going forward.

Major Properties of the Algorithms

The algorithms can be placed into three major categories. The **sequential algorithms**, `segment` and `warp`, hinge on their ability to correctly identify the return sweeps. If successful, these algorithms have excellent performance because any vertical drift in the data essentially becomes invisible. However, if the text is read nonlinearly, the premise on which the sequential algorithms are based breaks down. Therefore, one should apply these algorithms with great caution when the data are rich in regressions, either within-line (which `warp` seems to handle well) or between-line. The risk is particularly high with `segment`, which has good median performance on adult data, but can also lead to catastrophic errors if large regressions are mistakenly interpreted as return sweeps.

The **relative-positional algorithms**, `cluster`, `merge`, and `regress`, are mostly dependent on their ability to correctly classify the fixations into m groups, each using a slightly different technique. So long as the identified groups are sound, then the use of relative position to assign the fixations to lines is generally very resistant to vertical drift. `cluster`, in particular, emerged as particularly reliable; like `warp`, it yields small variability in accuracy against the human gold standard.

Particularly interesting is a comparison between the winners in their category. Contrary to `warp`, `cluster` is entirely invariant to between-line regressions. However, `warp` is much better than `cluster` at managing general noise and slope. This might seem like bad news for the community, because there is no all-around best solution. At the same time though, this also means that researchers will always have a good option available depending on what kind of data they have. For example, if the line spacing is quite tight, the eye tracking data is more likely be negatively impacted by distortion, making a sequential algorithm a better choice; conversely, if lines are spread far apart, a relative-positional algorithm might work better.

The **absolute-positional algorithms**, `attach`, `chain`, and `split`, generally tend to be the worst at dealing with vertical drift because they assign based on absolute position; this feature makes them generally weaker than other algorithms at dealing with noise, slope, and shift. On the other hand though, `attach` and `chain` are invariant to within- and between-line regressions. They also tend to be quite conservative and do not make dramatic changes to the data, which makes them a reasonable choice for researchers who would prefer a more minimalist data transformation (and whose data are not too affected

by vertical drift).

General Guidance for Researchers

The analyses presented in this paper clearly indicate that each algorithm performs best with data that are affected by one or more specific drift-related phenomena. None of the algorithms, however, handles all of them equally well, or well enough. Therefore, different sets of data will require different correction algorithms. A qualitative inspection of the data to detect the relative prevalence of noise, drift, and regression is in order to select one approach. This may not necessarily be a straightforward procedure. To help in the selection process, one option might be to hand-correct a subset of the sample trials in order to assess which algorithm performs best on those specific cases and then apply this algorithm to the entire dataset. However, there might very well be enough trial-by-trial (or participant-by-participant) variability, such that applying one correction algorithm to the whole dataset might not be the best approach. In this case, it may be preferable to create subsets of data exhibiting comparable patterns of eye tracking phenomena and deal with those subsets with different algorithms.

Recording children's eye movements poses extra challenges relative to adults', particularly due to the difficulty that younger participants often experience sitting still for relatively long periods of time (Blythe & Joseph, 2011), which can lead to a loss of calibration. Therefore, especially in the case of multiline reading, developing readers' eye movements are generally characterized by more noise, as well as by greater slope and shift, than adults'. This would suggest resorting to algorithms like `segment` and `warp`, which are entirely invariant to noise, slope, and shift (Fig. 6). However, children tend to generally make more regressions than adults (e.g., Blythe & Joseph, 2011; Reichle et al., 2003), which is exactly the phenomenon that affects `segment` and `warp` the most. The general tradeoff between the ability to handle distortion and the ability to handle regressions is at play here, and only an attentive, qualitative check of the data will tell the researcher which way to go. If there does not appear to be too much of an issue with between-line regressions, then `warp` is probably the best choice; otherwise, `cluster` or `merge` might be a better option.

Regarding the practicalities of the algorithm application pipeline, we suggest performing a few cleaning steps before drift correction, in order to isolate the line assignment problem from other issues that the algorithms considered here were not designed to deal with, and which would otherwise impair their performance. For example, researchers should discard any fixations that lie outside of the text area, such as out-of-bounds fixations. Only after these basic cleaning steps have been performed, can algorithmic correction be safely applied.

Another important aspect to consider is the presence of free parameters. The `chain`, `merge`, and `regress` algorithms take additional input parameters that must be set appropriately by the user. In practice, Špakov et al.'s (2019) and Cohen's (2013) suggested defaults for the `merge` and `regress` algorithms seemed to work well on our test dataset, but in the case of `chain`, it was somewhat unclear how to set the x and y thresholds appropriately; experimentation might be required to produce the best results. An advantage of the other algorithms is that they are parameter free, making drift correction easier to perform, document, and justify.

It is also worth considering the complexity of the algorithms. Some, such as `chain` and `segment`, are very simple and intuitive, while others, such as `merge` and `warp`, are quite complex. Although complexity is not an important consideration from a performance perspective (in general, we should prefer whichever algorithm works best), it is worth considering how complexity might impinge on real-world use. For example, users may be less inclined to use an algorithm if they cannot intuitively understand how it will manipulate their data, so algorithms should, where possible, be designed in a way that researchers find easy to understand and easy to convey to their readership. In that regard, we hope that this paper will give researchers more confidence in the algorithms, which we have validated and benchmarked.

Finally, it is worth noting that most of the algorithms have linear time complexity and can process a reading trial in fractions of a second, so runtime does not warrant any special consideration. The one exception to this is `merge` which scales quadratically with the number of fixations; in our testing, it took 100 ms for a trial consisting of around 100 fixations and up to 31 s for a trial of around 500 fixations.

Improvements on the Algorithms

We would not wish to claim that the algorithms, as presented here, are the only approaches one may take nor that they are the ultimate form of each core method; all nine of the algorithms can be improved in one way or another. The goal of this paper, however, was to evaluate the algorithms in their more abstract, idealized forms in order to make general recommendations and to provide a solid foundation for the future development of vertical drift correction software. Nevertheless, here we briefly note some of the most obvious ways in which the algorithms could be improved.

Chain

The main weakness of the `chain` algorithm is its reliance on threshold parameters that must be set by the user, but this situation could be improved if the parameters defaulted to sensible values based on reliable heuristics. For example, it may be the case that the parameters can reliably be estimated from the line and character spacing (as appears to be the case in Schroeder's (2019) implementation in `popEye`) or other known properties of the passage, language, or reader. Secondly, our simulations showed that `chain` does not respond well to slope distortion, performing worse than `attach` under extreme values. This can be alleviated by a y threshold that grows as the reader progresses over the line, as is the case in Hyrskykari's (2006) sticky lines algorithm.

Cluster

The biggest weakness of `cluster` was its ability to deal with general noise. One potential way to improve this would be to utilize the x values of the fixations. Unfortunately, it is not simply a case of performing a two-dimensional k-means clustering on the xy values because this leads to situations were clusters are identified that span multiple lines because they have similar x values. However, it might still be possible to utilize the x-axis information, perhaps by weighting the two axes differently in some way. Cluster analysis is a very

broad topic in data science, and there are likely to be many other candidate algorithms, beyond simple k-means clustering, that will be worth investigating.

Merge

The core principle of `merge` is to start with small groups of fixations and gradually build them up into gaze lines, guided by their fit to regression lines. The most extreme version of this algorithm would start with every fixation in an individual group, and the algorithm would consider every sequence in which mergers could be performed (i.e., the entire binary search tree). This would allow the algorithm to explore cases where it is first necessary to make a bad merger in order to make a great merger later on (i.e., it would avoid becoming stuck in local maxima). Such an algorithm would be intractable, however, due to a combinatorial explosion in the number of possible merge sequences. To avoid this, `merge` uses an initial `chain`-like strategy to seed the merge process with a reduced set of groups, and it then explores just one possible path through the search tree, selecting only the most promising merger at each step. One way to improve the algorithm, then, would be to use more advanced tree traversal techniques, such as beam search in which several of the most promising mergers are fully explored on each iteration. This would come at the cost of making an already slow algorithm even slower, but it would probably result in better solutions and might also allow for the removal of the thresholds and heuristics.

Regress

The main weakness of the `regress` algorithm is that the m regression lines it fits to the data cannot take independent slope or offset values, limiting its ability to handle complex cases, especially those involving shift. Thus, one obvious way to advance the algorithm would be to allow for such independent values. However, even the simplest case of having a single slope parameter, a single standard deviation parameter, and one offset parameter per line of text would result in an objective function with $m + 2$ parameters, which may become difficult or impossible to minimize, especially as the number of lines increases. Another avenue for improving `regress` would be to try some form of nonlinear regression. In Fig. 1B, for example, we see a case where a gaze line forms a nonlinear arc, which a linear regression line cannot fully capture (see Fig. 10D).

Segment

The performance of the `segment` algorithm hinges on its ability to identify the true return sweeps; when it works, it tends to work very well, but when it fails, it does so catastrophically. One way to improve the `segment` algorithm would therefore be to encode additional heuristics about how to distinguish true return sweeps from normal regressions. For example, a return sweep is not just an extreme movement to the left but also a movement downward by a relatively predictable amount (one line space), ultimately landing near the left edge of the passage. Introducing such heuristics would not be without caveats, however; in the case of downward slope, for example, return sweeps can appear quite flat (see e.g., the final sweep in Fig. 1A) and would therefore go unnoticed under this change.

Split

Like `segment`, `split` could also benefit from better sweep detection, as well as better detection of between-line regressions. There are likely to be many ways of approaching this classification problem, but one simple option would be to use both dimensions in the saccade clustering—the return sweeps would then be the cluster of saccades that have large negative change on the x-axis, as well as a small positive change on the y-axis. More generally, it might be possible to combine the `split` and `segment` algorithms, since they are quite closely related computationally. For example, the set of saccades that most resemble return sweeps could first be identified, and then the $m - 1$ most extreme of these could be treated as major segmentation points, allowing for a sequential assignment, while the remainder could be treated as minor segmentation points, allowing for the identification of between-line regressions.

Warp

DTW finds the best alignment between the expected and veridical fixation sequences, but the expected sequence cannot encode unpredictable reading behavior that might be present in the veridical sequence. Improving the `warp` algorithm will therefore involve relaxing DTW's requirement that matches between sequences increase monotonically, allowing the algorithm to find a mix of global and local sequence alignments. In this respect, the so-called "glocal" alignment algorithms could prove useful (Brudno et al., 2003), as well as many other sequence alignment algorithms that ought to be systematically investigated for the present purposes (e.g., Keogh & Pazzani, 2001; Tomasi et al., 2004; Tormene et al., 2009; Uchida, 2005). One simpler option—which could also be applied to `segment`—would be to use `attach` as a fallback method in cases where a fixation's revised y-axis coordinate is substantially different from its original y-axis coordinate.

Conclusion

Our intentions with this paper were twofold. Firstly, we wanted to systematically evaluate the various vertical drift correction algorithms that have been reported in the literature in order to provide guidance to researchers about how they work, when they should be used, and what their limits are. In this respect, our most important observation was that there is no one killer app; different datasets—and even different trials within a dataset—will require different solutions, so researchers should select their correction method carefully. We hope that the guidance we have provided herein will be helpful in this regard.

Secondly, we wanted to lay a solid foundation for future work on post-hoc vertical drift correction by delimiting the core algorithms, providing constraints that future work can operate inside, and offering new perspectives on how drift correction techniques can be improved going forward. In this respect, we have provided basic implementations of the nine algorithms in multiple languages, which can be used as a starting point for building new versions or, indeed, as a comparison group against which entirely new algorithms can be compared.

It is possible that our use of an Italian dataset may mean that our results will not generalize well to logographic writing systems, such as Chinese, right-to-left scripts, such as Hebrew, orthographically opaque languages, such as English, or agglutinating languages,

such as Turkish, where fixation patterns might differ in ways that the algorithms are sensitive to. Likewise, it is unclear to what extent our findings will generalize to other eye tracker hardware or alternative passage sizing and layout. We leave such exploration to future work.

Finally, we have introduced two novel methods in this paper that are distinct from those that have previously been presented. The `warp` algorithm, in particular, showed great promise and is likely to be especially useful to researchers working on reading development in children. We also hope that connecting the literature on vertical drift to sequence alignment techniques might also open new avenues for future algorithm development.

Appendix

For reference, we present the algorithms here in a pseudocode that should be clear to programmers of any high-level scientific computing language. Matlab/Octave, Python, and R implementations may be found at https://doi.org/10.17605/OSF.IO/7SRKG. We have generally emphasized readability over optimization, and we make very minimal assumptions about the input and output. Most of the algorithms take two inputs: `fixation_XY`, an array of size $n \times 2$ representing the xy positions of n fixations, and `line_Y`, an array of length m representing the y positions of the m lines of text. Some algorithms take slightly different input or additional arguments as detailed below. All algorithms return a modified `fixation_XY` as output, in which only the y values have been adjusted.

Attach

```
function attach(fixation_XY, line_Y)
|  n = length(fixation_XY)
|  for fixation_i in 1 : n
|  |  fixation_y = fixation_XY[fixation_i, 2]
|  |  line_i = argmin(abs(line_Y - fixation_y))
|  |  fixation_XY[fixation_i, 2] = line_Y[line_i]
|  return fixation_XY
```

Chain

`chain` takes two additional arguments, `x_thresh` and `y_thresh`, which determine how much change is required on the x- or y-axis to start a new chain of fixations.

```
function chain(fixation_XY, line_Y, x_thresh=192, y_thresh=32)
|  n = length(fixation_XY)
|  dist_X = abs(diff(fixation_XY[:, 1]))
|  dist_Y = abs(diff(fixation_XY[:, 2]))
|  end_chain_indices = where(dist_X > x_thresh or dist_Y > y_thresh)
|  end_chain_indices = append(end_chain_indices, n)
|  start_of_chain = 1
|  for end_of_chain in end_chain_indices
|  |  mean_y = mean(fixation_XY[start_of_chain:end_of_chain, 2])
|  |  line_i = argmin(abs(line_Y - mean_y))
```

```
|  |  fixation_XY[start_of_chain:end_of_chain, 2] = line_Y[line_i]
|  |  start_of_chain = end_of_chain + 1
|  return fixation_XY
```

Cluster

`cluster` calls on one external function, `kmeans`, which returns `clusters`, an array of length n that gives the cluster index of each fixation, and `centers`, an array of length m that gives the mean y value of each cluster.

```
function cluster(fixation_XY, line_Y)
|  n = length(fixation_XY)
|  m = length(line_Y)
|  fixation_Y = fixation_XY[:, 2]
|  clusters, centers = kmeans(fixation_Y, m)
|  ordered_cluster_indices = argsort(centers)
|  for fixation_i in 1 : n
|  |  cluster_i = clusters[fixation_i]
|  |  line_i = where(ordered_cluster_indices == cluster_i)
|  |  fixation_XY[fixation_i, 2] = line_Y[line_i]
|  return fixation_XY
```

Compare

Instead of `line_Y`, `compare` takes `word_XY` as its second argument, an array representing the xy positions of the centers of all words in the order in which they are expected to be read. It takes two additional arguments: `x_thresh`, which specifies the threshold for considering backward saccades to be return sweeps, and `n_nearest_lines`, which determines how many neighboring text lines a gaze line will be compared to. `compare` calls on one external function, `dynamic_time_warping`, which returns the DTW cost between a gaze line and text line.

```
function compare(fixation_XY, word_XY, x_thresh=512,  n_nearest_lines=3)
|  n = length(fixation_XY)
|  line_Y = unique(word_XY[:, 2])
|  diff_X = diff(fixation_XY[:, 1])
|  end_line_indices = where(diff_X < -x_thresh)
|  end_line_indices = append(end_line_indices, n)
|  start_of_line = 1
|  for end_of_line in end_line_indices
|  |  gaze_line = fixation_XY[start_of_line:end_of_line]
|  |  mean_y = mean(gaze_line[:, 2])
|  |  lines_ordered_by_proximity = argsort(abs(line_Y - mean_y))
|  |  nearest_line_I = lines_ordered_by_proximity[1:n_nearest_lines]
|  |  line_costs = zeros(n_nearest_lines)
|  |  for candidate_i in 1 : n_nearest_lines
|  |  |  candidate_line_i = nearest_line_I[candidate_i]
```

```
|  |  |  candidate_line_y = line_Y[candidate_line_i]
|  |  |  text_line = word_XY[word_XY[:, 2] == candidate_line_y]
|  |  |  cost, _ = dynamic_time_warping(gaze_line[:, 1], text_line[:, 1])
|  |  |  line_costs[candidate_i] = cost
|  |  line_i = nearest_line_I[argmin(line_costs)]
|  |  fixation_XY[start_of_line:end_of_line, 2] = line_Y[line_i]
|  |  start_of_line = end_of_line + 1
|  return fixation_XY
```

Merge

`merge` takes three additional arguments: `y_thresh` determines how much change is required on the y-axis to start a new sequence of progressive fixations; `g_thresh` determines the maximum absolute gradient of the fit regression lines; and `e_thresh` determines the maximum regression error. `merge` calls on one external function, `linear_model`, which fits a regression line to a candidate set of fixations and returns `g`, the absolute gradient, and `e`, the regression error (RMSD). The global variable `phases` defines three parameters per phase of the merge process: the minimum number of fixations in the first candidate sequence; the minimum number of fixations in the second candidate sequence; and a Boolean that removes the gradient and error constraints (this should be `TRUE` in the final phase to ensure that the number of sequences can be reduced to m).

```
phases = [[3, 3, FALSE], [1, 3, FALSE], [1, 1, FALSE], [1, 1, TRUE]]

function merge(fixation_XY, line_Y, y_thresh=32, g_thresh=0.1, e_thresh=20)
|  n = length(fixation_XY)
|  m = length(line_Y)
|  diff_X = diff(fixation_XY[:, 1])
|  dist_Y = abs(diff(fixation_XY[:, 2]))
|  sequence_boundaries = where(diff_X < 0 or dist_Y > y_thresh)
|  sequence_boundaries = append(sequence_boundaries, n)
|  sequences = []
|  start_of_sequence = 1
|  for end_of_sequence in sequence_boundaries
|  |  sequence = start_of_sequence : end_of_sequence
|  |  sequences = append(sequences, sequence)
|  |  start_of_sequence = end_of_sequence + 1
|  for min_i, min_j, no_constraints in phases
|  |  while length(sequences) > m
|  |  |  best_merger = NONE
|  |  |  best_error = INFINITY
|  |  |  for i in 1 : length(sequences) - 1
|  |  |  |  if length(sequences[i]) < min_i
|  |  |  |  |  next # first sequence too short, skip to next i
|  |  |  |  for j in i+1 : length(sequences)
|  |  |  |  |  if length(sequences[j]) < min_j
```

```
|  |  |  |  |  |  next # second sequence too short, skip to next j
|  |  |  |  |  candidate_sequence = concatenate(sequences[i], sequences[j])
|  |  |  |  |  g, e = linear_model(fixation_XY[candidate_sequence])
|  |  |  |  |  if no_constraints == TRUE or (g < g_thresh and e < e_thresh)
|  |  |  |  |  |  if e < best_error
|  |  |  |  |  |  |  best_merger = [i, j]
|  |  |  |  |  |  |  best_error = e
|  |  |  if best_merger == NONE
|  |  |  |  break # no possible mergers, break while and move to next phase
|  |  |  i, j = best_merger
|  |  |  combined_sequence = concatenate(sequences[i], sequences[j])
|  |  |  sequences = append(sequences, combined_sequence)
|  |  |  delete sequences[j], sequences[i]
|  mean_Y = zeros(length(sequences))
|  for sequence_i in 1 : length(sequences)
|  |  mean_Y[sequence_i] = mean(fixation_XY[sequences[sequence_i], 2])
|  ordered_sequence_indices = argsort(mean_Y)
|  for sequence_i in 1 : length(sequences)
|  |  line_i = where(ordered_sequence_indices == sequence_i)
|  |  fixation_XY[sequences[sequence_i], 2] = line_Y[line_i]
|  return fixation_XY
```

Regress

`regress` takes three additional arguments, `K`, `O`, and `S`, which give the lower and upper bounds of the slope, offset, and standard deviation. `regress` calls on one external function, `minimize`, which minimizes the objective function `fit_lines`. The `fit_lines` function is nested inside the `regress` function so that it inherits its lexical scope.

```
function regress(fixation_XY, line_Y, K=[-0.1,0.1], O=[-50,50], S=[1,20])
|  n = length(fixation_XY)
|  m = length(line_Y)
|
|  function fit_lines(params, return_line_assignments=FALSE)
|  |  density = matrix(n, m)
|  |  k = K[1] + (K[2] - K[1]) * cdf(params[1])
|  |  o = O[1] + (O[2] - O[1]) * cdf(params[2])
|  |  s = S[1] + (S[2] - S[1]) * cdf(params[3])
|  |  predicted_Y_from_slope = fixation_XY[:, 1] * k
|  |  line_Y_plus_offset = line_Y + o
|  |  for line_i in 1 : m
|  |  |  fit_Y = predicted_Y_from_slope + line_Y_plus_offset[line_i]
|  |  |  density[:, line_i] = logpdf(fixation_XY[:, 2], fit_Y, s)
|  |  if return_line_assignments == TRUE
|  |  |  return argmax(density, axis=2)
|  |  return -sum(max(density, axis=2))
```

```
|
|  initial_params = [0, 0, 0]
|  best_params = minimize(fit_lines, initial_params)
|  line_assignments = fit_lines(best_params, TRUE)
|  for fixation_i in 1 : n
|  |  line_i = line_assignments[fixation_i]
|  |  fixation_XY[fixation_i, 2] = line_Y[line_i]
|  return fixation_XY
```

Segment

```
function segment(fixation_XY, line_Y)
|  n = length(fixation_XY)
|  m = length(line_Y)
|  diff_X = diff(fixation_XY[:, 1])
|  saccades_ordered_by_length = argsort(diff_X)
|  line_change_indices = saccades_ordered_by_length[1:m-1]
|  current_line_i = 1
|  for fixation_i in 1 : n
|  |  fixation_XY[fixation_i, 2] = line_Y[current_line_i]
|  |  if fixation_i is in line_change_indices
|  |  |  current_line_i = current_line_i + 1
|  return fixation_XY
```

Split

`split` calls on one external function, `kmeans`, which returns `clusters`, an array of length $n-1$ that gives the cluster index of each saccade, and `centers`, an array of length 2 that gives the mean saccade length of each cluster. Whichever cluster has the smaller (i.e., more negative) mean saccade length is assumed to be the cluster that contains the return sweeps.

```
function split(fixation_XY, line_Y)
|  n = length(fixation_XY)
|  diff_X = diff(fixation_XY[:, 1])
|  clusters, centers = kmeans(diff_X, 2)
|  sweep_marker = argmin(centers)
|  end_line_indices = where(clusters == sweep_marker)
|  end_line_indices = append(end_line_indices, n)
|  start_of_line = 1
|  for end_of_line in end_line_indices
|  |  mean_y = mean(fixation_XY[start_of_line:end_of_line, 2])
|  |  line_i = argmin(abs(line_Y - mean_y))
|  |  fixation_XY[start_of_line:end_of_line, 2] = line_Y[line_i]
|  |  start_of_line = end_of_line + 1
|  return fixation_XY
```

Warp

Instead of `line_Y`, `warp` takes `word_XY` as its second argument: an array representing the *xy* center positions of all words in the order in which they are expected to be read. `warp` calls on one external function, `dynamic_time_warping`, which returns the warping path, a list-of-lists structure that records which words are mapped to each fixation.

```
function warp(fixation_XY, word_XY)
|  n = length(fixation_XY)
|  _, path = dynamic_time_warping(fixation_XY, word_XY)
|  for fixation_i in 1 : n
|  |  words_mapped_to_fixation_i = path[fixation_i]
|  |  candidate_Y = word_XY[words_mapped_to_fixation_i, 2]
|  |  fixation_XY[fixation_i, 2] = mode(candidate_Y)
|  return fixation_XY
```

Acknowledgments

This work was funded by an ERC Starting Grant (no. 679010, STATLEARN) and a FARE 2016 grant (no. R164XBHNYR, CROWDLEARN), both awarded to **DC**. **VNP**, **MK**, and **DC** conceived the original idea behind this work. **JWC** and **VNP** explored the literature and selected the algorithms. **VNP** and **MF** developed the `split` algorithm, with input from **MK** and **DC**. **JWC** developed the `warp` algorithm, implemented all algorithms in a single framework, conducted the simulations, and performed the analyses. **JWC** and **VNP** performed the manual corrections and drafted the paper. **DC** revised the paper and coordinated the project.

References

Abdulin, E. R., & Komogortsev, O. V. (2015). Person verification via eye movement-driven text reading model. In *2015 IEEE 7th International Conference on Biometrics Theory, Applications and Systems*. IEEE. https://doi.org/10.1109/BTAS.2015.7358786

Beymer, D., & Russell, D. M. (2005). WebGazeAnalyzer: A system for capturing and analyzing web reading behavior using eye gaze. In *CHI '05 extended abstracts on Human Factors in Computing Systems* (pp. 1913–1916). Association for Computing Machinery. https://doi.org/10.1145/1056808.1057055

Blythe, H. I., & Joseph, H. S. S. L. (2011). Children's eye movements during reading. In S. Liversedge, I. Gilchrist, & S. Everling (Eds.), *The Oxford handbook of eye movements* (pp. 643–662). Oxford University Press. https://doi.org/10.1093/oxfordhb/9780199539789.013.0036

Blythe, H. I., Liversedge, S. P., Joseph, H. S. S. L., White, S. J., & Rayner, K. (2009). Visual information capture during fixations in reading for children and adults. *Vision Research*, *49*(12), 1583–1591. https://doi.org/10.1016/j.visres.2009.03.015

Brudno, M., Malde, S., Poliakov, A., Do, C. B., Couronne, O., Dubchak, I., & Batzoglou, S. (2003). Glocal alignment: Finding rearrangements during alignment. *Bioinformatics*, *19*(supplement 1), i54–i62. https://doi.org/10.1093/bioinformatics/btg1005

Carl, M. (2013). Dynamic programming for re-mapping noisy fixations in translation tasks. *Journal of Eye Movement Research*, *6*(2), Article 5. https://doi.org/10.16910/jemr.6.2.5

Cohen, A. L. (2013). Software for the automatic correction of recorded eye fixation locations in reading experiments. *Behavior Research Methods*, *45*(3), 679–683. https://doi.org/10.3758/s13428-012-0280-3

Cop, U., Dirix, N., Drieghe, D., & Duyck, W. (2017). Presenting GECO: An eyetracking corpus of monolingual and bilingual sentence reading. *Behavior Research Methods*, *49*(2), 602–615. https://doi.org/10.3758/s13428-016-0734-0

Frazier, L., & Rayner, K. (1982). Making and correcting errors during sentence comprehension: Eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology*, *14*(2), 178–210. https://doi.org/10.1016/0010-0285(82)90008-1

Hyrskykari, A. (2006). Utilizing eye movements: Overcoming inaccuracy while tracking the focus of attention during reading. *Computers in Human Behavior*, *22*(4), 657–671. https://doi.org/10.1016/j.chb.2005.12.013

Jarodzka, H., & Brand-Gruwel, S. (2017). Tracking the reading eye: Towards a model of real-world reading. *Journal of Computer Assisted Learning*, *33*(3), 193–201. https://doi.org/10.1111/jcal.12189

Joseph, H. S. S. L., Liversedge, S. P., Blythe, H. I., White, S. J., & Rayner, K. (2009). Word length and landing position effects during reading in children and adults. *Vision Research*, *49*(16), 2078–2086. https://doi.org/10.1016/j.visres.2009.05.015

Keogh, E. J., & Pazzani, M. J. (2001). Derivative dynamic time warping. In V. Kumar & R. Grossman (Eds.), *Proceedings of the 2001 SIAM International Conference on Data Mining* (pp. 1–11). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611972719.1

Kuperman, V., Matsuki, K., & Van Dyke, J. A. (2018). Contributions of reader- and text-level characteristics to eye-movement patterns during passage reading. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *44*(11), 1687–1713. https://doi.org/10.1037/xlm0000547

Lima Sanches, C., Kise, K., & Augereau, O. (2015). Eye gaze and text line matching for reading analysis. In *Adjunct proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and proceedings of the 2015 ACM International Symposium on Wearable Computers* (pp. 1227–1233). Association for Computing Machinery. https://doi.org/10.1145/2800835.2807936

Luke, S. G., & Christianson, K. (2018). The Provo Corpus: A large eye-tracking corpus with predictability norms. *Behavior Research Methods*, *50*(2), 826–833. https://doi.org/10.3758/s13428-017-0908-4

Martinez-Gomez, P., Chen, C., Hara, T., Kano, Y., & Aizawa, A. (2012). Image registration for text-gaze alignment. In *Proceedings of the 2012 ACM International Conference on Intelligent User Interfaces* (pp. 257–260). Association for Computing Machinery. https://doi.org/10.1145/2166966.2167012

Mishra, A., Carl, M., & Bhattacharyya, P. (2012). A heuristic-based approach for systematic error correction of gaze data for reading. In *Proceedings of the First Workshop on Eye-tracking and Natural Language Processing* (pp. 71–80).

O'Regan, J. K., Lévy-Schoen, A., Pynte, J., & Brugaillère, B. (1984). Convenient fixation location within isolated words of different length and structure. *Journal of Experimental Psychology: Human Perception and Performance*, *10*(2), 250–257. https://doi.org/10.1037/0096-1523.10.2.250

Palmer, C., & Sharif, B. (2016). Towards automating fixation correction for source code. In *Proceedings of the 9th biennial ACM Symposium on Eye Tracking Research & Applications* (pp. 65–68). Association for Computing Machinery. https://doi.org/10.1145/2857491.2857544

Parker, A. J., Slattery, T. J., & Kirkby, J. A. (2019). Return-sweep saccades during reading in adults and children. *Vision Research*, *155*, 35–43. https://doi.org/10.1016/j.visres.2018.12.007

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Pescuma, V. N., Crepaldi, D., & Ktori, M. (in prep.). EyeReadIt: A developmental database of eye movement measures during natural reading in Italian. https://doi.org/10.17605/OSF.IO/HX2SJ

Pickering, M. J., & Traxler, M. J. (1998). Plausibility and recovery from garden paths: An eye-tracking study. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *24*(4), 940–961. https://doi.org/10.1037//0278-7393.24.4.940

Rayner, K. (1986). Eye movements and the perceptual span in beginning and skilled readers. *Journal of Experimental Child Psychology*, *41*(2), 211–236. https://doi.org/10.1016/0022-0965(86)90037-8

Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin*, *124*(3), 372–422. https://doi.org/10.1037/0033-2909.124.3.372

Rayner, K., Binder, K. S., Ashby, J., & Pollatsek, A. (2001). Eye movement control in reading: Word predictability has little influence on initial landing positions in words. *Vision Research*, *41*(7), 943–954. https://doi.org/10.1016/S0042-6989(00)00310-2

Reichle, E. D., Rayner, K., & Pollatsek, A. (2003). The E-Z Reader model of eye-movement control in reading: Comparisons to other models. *Behavioral and Brain Sciences*, *26*(4), 445–476. https://doi.org/10.1017/S0140525X03000104

Sakoe, H., & Chiba, S. (1978). Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *26*(1), 43–49. https://doi.org/10.1109/tassp.1978.1163055

Schotter, E. R., Angele, B., & Rayner, K. (2012). Parafoveal processing in reading. *Attention, Perception, & Psychophysics*, *74*(1), 5–35. https://doi.org/10.3758/s13414-011-0219-2

Schroeder, S. (2019). popEye – An R package to analyse eye movement data from reading experiments. https://github.com/sascha2schroeder/popEye

Sereno, S. C., O'Donnell, P. J., & Rayner, K. (2006). Eye movements and lexical ambiguity resolution: Investigating the subordinate-bias effect. *Journal of Experimental Psychology: Human Perception and Performance*, *32*(2), 335–350. https://doi.org/10.1037/0096-1523.32.2.335

Špakov, O., Istance, H., Hyrskykari, A., Siirtola, H., & Räihä, K.-J. (2019). Improving the performance of eye trackers with limited spatial accuracy and low sampling rates for reading analysis by heuristic fixation-to-word mapping. *Behavior Research Methods*, *51*(6), 2661–2687. https://doi.org/10.3758/s13428-018-1120-x

Tiffin-Richards, S. P., & Schroeder, S. (2015). Children's and adults' parafoveal processes in German: Phonological and orthographic effects. *Journal of Cognitive Psychology*, *27*(5), 531–548. https://doi.org/10.1080/20445911.2014.999076

Tiffin-Richards, S. P., & Schroeder, S. (2020). Context facilitation in text reading: A study of children's eye movements. *Journal of Experimental Psychology: Learning, Memory, and Cognition*. https://doi.org/10.1037/xlm0000834

Tomasi, G., van den Berg, F., & Andersson, C. (2004). Correlation optimized warping and dynamic time warping as preprocessing methods for chromatographic data. *Journal of Chemometrics*, *18*(5), 231–241. https://doi.org/10.1002/cem.859

Tormene, P., Giorgino, T., Quaglini, S., & Stefanelli, M. (2009). Matching incomplete time series with dynamic time warping: An algorithm and an application to post-stroke rehabilitation. *Artificial Intelligence in Medicine*, *45*(1), 11–34. https://doi.org/10.1016/j.artmed.2008.11.007

Uchida, S. (2005). A survey of elastic matching techniques for handwritten character recognition. *IEICE Transactions on Information and Systems*, *E88-D*(8), 1781–1790. https://doi.org/10.1093/ietisy/e88-d.8.1781

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Vitu, F., Kapoula, Z., Lancelin, D., & Lavigne, F. (2004). Eye movements in reading isolated words: Evidence for strong biases towards the center of the screen. *Vision Research*, *44*(3), 321–338. https://doi.org/10.1016/j.visres.2003.06.002

Zhang, Y., & Hornof, A. J. (2011). Mode-of-disparities error correction of eye-tracking data. *Behavior Research Methods*, *43*(3), 834–842. https://doi.org/10.3758/s13428-011-0073-0

Zhang, Y., & Hornof, A. J. (2014). Easy post-hoc spatial recalibration of eye tracking data. In *Proceedings of the Symposium on Eye Tracking Research and Applications* (pp. 95–98). Association for Computing Machinery. https://doi.org/10.1145/2578153.2578166