Parsing Ink Annotations on Heterogeneous Documents

Xin Wang[1] and Michael Shilman[2] and Sashi Raghupathy[1]

[1]Ink Parsing Team, TabletPC, Microsoft Corp, One Microsoft Way, RedMond, WA 98052, USA
[2]Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA

Abstract

Annotation is an integral part of reading, comprehending, commenting, and authoring notes and documents. In this paper we present a system for recognizing annotations in a flexible digital notebook that may contain a variety of content ranging from text, to images, to handwritten notes. To accomplish the recognition task in real-time makes the complicated annotation parsing problem more difficult.

Our approach differs from previous approaches in several ways. First, our approach handles annotations on ink notes, which are significantly more ambiguous than annotations on printed documents and hence more difficult to recognize. Second, our approach is entirely learned from data, so it is easy to adapt to other scenarios. Third, our approach is more thoroughly evaluated than previous systems. On a test set of real user notes, the system has achieved an average recall of 0.9258 on all annotation types. Finally, the implementation of the approach will be commercially available as an API in the upcoming release of Windows® Vista® and Office 12®.

Categories and Subject Descriptors (according to ACM CCS): I.7.m [Computing Methodologies]: Document and Text Processing; I.5.4 [Computing Technology]: Pattern Recognition

1. Introduction

A Holy Grail of personal information management is a digital notebook application that simplifies storage, sharing, retrieval, and manipulation of a user's notes, diagrams, web clippings, and so on. This application should be able to flexibly incorporate a wide variety of data types and deal with them reasonably. One approach, as exemplified by Microsoft OneNote®, is to explicitly represent different data types in a single application, and let users capture and fluidly manipulate text, digital ink, and images in data type-specific ways. The application becomes more powerful when ink is intelligently interpreted and given appropriate behaviors according to the type. For instance, hierarchical lists in digital ink notes should be able to expand and collapse just like hierarchical lists in text-based note-taking tools.

Annotations are an important part of a user's interaction with both paper and digital documents, and can be used in numerous ways within the digital notebook. Users annotate documents for comprehension, authoring, editing, note-taking, author feedback, and so on.

When annotations are recognized, they become a form of structured content that semantically decorates any of the
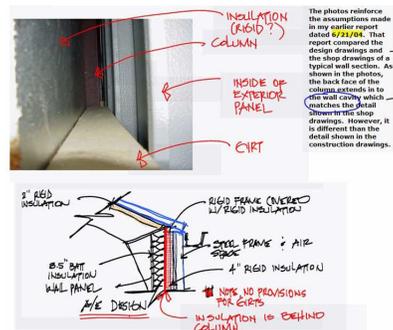

Figure 1: *A OneNote 12® file with a mixture of ink, text and images. The fist section of the file is a regular text region. The second section consists of two images with ink annotations. The last section is an ink-drawing with ink annotations surrounding it.*

other data types in a digital notebook. Recognized annotations can be anchored to document content, so that the annotations can be reflowed as the document layout changes. They can assist in information retrieval, marking places in

the document of particular interest or importance. Editing marks such as deletion or insertion can be invoked as actions on the underlying document. Users annotate documents by habit; recognizing those annotations increases their value in the lifecycle of the digital document.

In this paper, we present a set of techniques for recognizing an assortment of digital ink annotations against a variety of underlying document content, including other digital notes and diagrams. Unlike previous work that heuristically recognizes, anchors, and reflows digital ink annotations against text documents [SW04], our approach works on document with heterogeneous content types. In particular, recognizing digital ink annotations in the context of other digital ink notes is highly ambiguous, and therefore extremely difficult. With the real–time requirement of an API to be integrated into commerical note taking softwares such as One Note, the problem is even more complicated.

Furthermore, our method is based entirely on learning from training data, so if new annotation types or new content types are added, the system can be retrained to incorporate these new types. This flexibility in the system design allows us to recognize more annotation types when moving from our Beta–1 version to Beta–2.

Finally, the technique we describe achieves reasonable accuracy on real user notes. It will be shipping with Windows Vista®† and OneNote® 2007‡.

Recognizing ink annotations that occur in ink notes is significantly more difficult than in printed documents. On a printed document, every ink stroke must belong to an annotation of some type. For example, a strikethrough is identifiable if it is crosses through a printed line of text and aligns with its baseline. However, in an ink note it is not always clear which strokes should be grouped into lines, or, given a hypothesized line and a potential annotation stroke, whether the stroke is a strikethrough or perhaps merely a long crossing stroke on the letter "t". Numerous such problems make it difficult for the computer to accurately discriminate between the handwritten notes and the annotations that modify those notes.

In Section 2, we give an introduction to the terminology. We also introduce several important user scenarios of the annotation system as part of an introduction to the scope of the system. In Section 3, we describe the functional details of our ink annotation parsing system: its tasks, its architecture, and its integration with the rest of our ink parsing system. We also cover the algorithmic aspects of the system: its classification, segmentation, and annotation anchoring. In Section 5, we present the evaluation results of our system. In Section **??**, we describe future work.

---

2. Definitions and User Scenarios

2.1. Annotation

An ink annotation on such a document consists of a group of semantically and spatially related ink strokes that annotate the main content of the document. They provide supplementary information to the main body and sometimes establish relationships between different parts of the document. In this paper, we will focus on annotations formed by drawing strokes, which do not group with the rest of the text the user has written.

As pointed out by [Mar97], there are many different types of annotation. Each serves a different type of marking or editing activity. There are two major classes of annotations:

- **Non-Actionable annotations:** annotations that just explain, summarize, emphasize or comment on the main content, see Figure 2;
- **Actionable annotations:** annotations that denote editorial actions such as insertion, deletion, transposition, and movement.
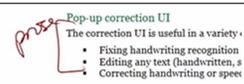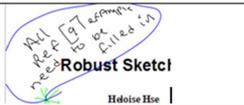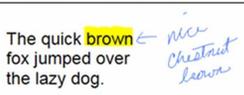

Figure 2: *Non-actionable annotations. Instead of specifying an special editorial actions on the main content, these annotations explain, summarize, emphasize, and comment on them.*

No matter whether it specifies an action or not, an annotation involves two types of information, the **geometric** information and the **semantic** information. In this paper, we use geometric information to refer to what kind of ink-strokes the annotation has, how the strokes form a geometric shape, and how the shape relates (both temporally and spatially) to other ink-strokes in the file. For all the annotation types supported by our system, we allow multiple-to-multiple mapping between the shapes of an annotation and their types. Without restricting the system to handle only the situation where a set of shapes are reserved for one annotation type only, this introduces additional difficulty into the parsing task.

We use semantic information to refer to the meaning or the function of the annotation, and how it relates to other semantic objects in the document—words, lines, and blocks of text, or images.

2.2. Supported Annotation Types

As shown in Figure 3, our system supports four categories and eight types of annotation according to both the semantic and the geometric information they carry.
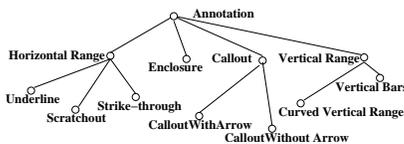

Figure 3: *Class hierarchy supported by our ink annotation system.*

The four categories we support are: horizontal ranges, vertical ranges, enclosures, and callouts. For horizontal ranges, we support three subtypes, underlines, strike-throughs, and scratch-outs of different shapes. For vertical ranges, to im-

Figure 4: *Samples for annotation types that are currently supported by the annotation parsing system. For each type, only one example of shape is shown, even though in our system they are not restricted to take only one shape.*

prove recognition accuracy, we divide the category into two subtypes, vertical range in general (brace, bracket, parantheses and etc), and vertical bar in particular (both single and double vertical bars).

For enclosure, we recognize blobs of different shapes: rectangle, ellipse, and other regular or irregular shapes. Our system can even recognize partial enclosures or enclosures that overlap more than once.

For callouts, we support both straight line callouts with or without arrowheads, curved callouts with or without arrowheads, and elbow callouts with or without arrowheads.

2.3. Anchoring

No matter what geometric shape it takes, an annotation always establishes a semantic relationship among parts of a document. The parts can be regions or *spans* in the document, such as part of a line, a paragraph, an ink or text region, or an image. The annotation can also denote a specific position in the document such as before or after a word, on top of an image and so on. We call these relationships anchors, and in addition to identifying the type of annotation for a set of strokes, the annotation parser must also identify its anchors.

3. Parsing System

3.1. System Overview

Our ink parsing system consists of a stack of engines as shown in 5. Each engine works on a specific semantic problem and enriches or improves upon the partial parsing results that are passed to it. For example, the writing-drawing classification engine classifies all the incoming ink strokes into writing or drawing [BSH04], and the line finding engine groups ink strokes into lines of writing [YSR*05]. The annotation engine is a new engine added to the end of the stack. It identifies groups of ink strokes that are annotations, their types, and their corresponding anchors.

Anybody who has tried to interpret full pages of ink notes from real user data knows that ink is locally ambiguous, and can only be accurately interpreted in a global context. Therefore it is not obvious how our feed-forward architecture can work on real notes. In some sense, each engine is responsible for its own task, plus some subset of the tasks before it in the stack. For example, the annotation engine will often second guess earlier writing-drawing decisions, examining writing strokes at the end of connectors to try to find arrowheads that have been misclassified. This increases the responsibility and reduces the modularity of each engine, but allows us to optimize each stage for accuracy and performance without resorting to a global optimization strategy which will be difficult to complete in real time on today's computers.

Our annotation parsing approach is an evolution of the annotation parser presented in [SW04] and the symbol grouping and classification approach of [SVC04]. [SW04] identified annotations and their anchors using a complex set of heuristics. [SVC04] simultaneously optimized over a set of segmentation and recognition hypotheses and was entirely learned from data. We first present an adaptation of [SVC04] to the problem of annotation parsing and anchoring. We then heuristically and greedily refine this adaptation to operate in close to real-time.

3.2. Optimal Annotation Parsing

The job of the annotation parser is to segment, recognize, and anchor ink strokes against a background document. We

can perform all of these functions simultaneously using a variant of the technique described in [SVC04].

Assume a trained recognizer R, which, given a candidate set of strokes, anchors, and background, can reasonably hypothesize the candidate as an annotation of a specific type, or as garbage. Given such a recognizer, one merely needs to enumerate over a reasonable set of candidates. One method is to connect all of the strokes into a neighborhood graph. Two strokes are connected in the graph if the Euclidian distance between their convex hulls is less than a threshold, as shown in Figure Y. This threshold can be empirically determined based on the maximum distance between any two strokes that fall into the same labeled symbol in training data. Assuming some maximum number of strokes per symbol, K, [SVC04] presents an efficient way to enumerate connected subsets of this graph, which form symbol candidates.

Given the recognizer R, and a candidate enumeration method, it is possible to solve for an optimal grouping, recognition over all the strokes through dynamic programming on the recurrence equation in [SVC04].

Unfortunately, in consumer user interfaces we must often sacrifice optimality and simplicity for performance. Our entire stack of engines, including writing-drawing classification, line grouping, annotations parsing, and so on, must complete in approximately 1*ms* per stroke. If we budget 10% of this time for annotations parsing, this means our annotation engine must process a 500 stroke page, including segmentation and recognition, in 100*ms*! Therefore, we employ a greedy optimization and a set of heuristics to approximate this optimization. In the next section, we describe the features and training procedure for R, and the heuristic accelerations of this optimization.
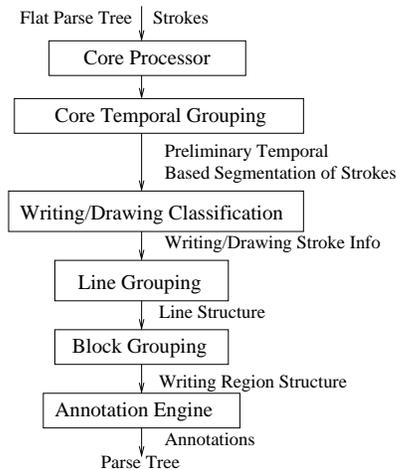
Figure 5: *The engine stack of ink parser. Partial parsing results, represented as parsing trees, are passed from one engine to another.*

3.3. Implementation

As one of the last engines at the engine stack in Fig. 5, in addition to the original ink, text and image information, it can also access the rich temporal and spatial information the other engines generated and their analysis results. For example, the annotation parser can use previous parsing results on ink type property of a stroke (writing/drawing). It can also use the previously parsed word, line, paragraph and block layout structure of the underlying document. As shown in
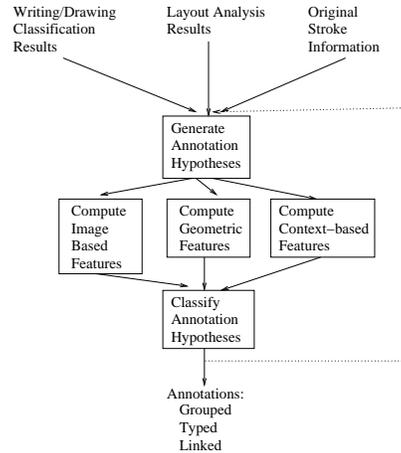
Figure 6: *Architecture of the annotation parser.*

Fig. 6, the annotation parser iterates through the following three steps: hypothesis generation, feature computation and hypothesis evaluation.

3.3.1. Generate Hypothesis

The first step is to generate hypothesis. Ideally, we want to generate a hypothesis for each possible stroke grouping, annotation type, and anchor set, but this is not feasible for a real-time system. Aggressive heuristic pruning has to be adopted to parse within the system's time limits. In practice, we found that spatial and temporal heuristics are not sufficient to achieve acceptable recognition results. Instead, it is necessary to use heuristics based on knowledge of previous parsing results.

For stroke grouping, we can prune the set of all possible annotation stroke group candidates greatly based on previous writing/drawing classification results. [§]

If we know the type of the underlying and surrounding regions of a stroke group candidate, we can limit its set of feasible annotation types to a subset of all annotation types supported by the system. For example, if we know a line segment goes from an image region to a text region, it is more

---

[§] Since the writing/drawing classification engine makes mistakes, we can not limit our choices to drawing strokes only.

likely to be a callout without arrow or a vertical range than a strike-through.

Similarly if we know the type of an annotation, we can also reduce the set of possible anchors. For a vertical range, its anchor can only be on its left or right side, and for an underline, its anchor can only be above it.

With carefully designed heurisitcs, we are able to significantly reduce the number of hypotheses generated.

3.4. Feature Computation

For each hypothesis we enumerate through, we compute a combined set of shape and context features. We use two types of shape features—the cheap image-based Viola-Jones filters and the more expensive features based on the geometric properties of its polyline and convex hull. For the geometric features, we use both features that are general enough to work across a variety of shapes and annotation types and features designed to discriminate two or more specific annotation types. More details can be found in Section 4.

3.5. Feature Selection and Hypothesis Evaluation

The annotation parser uses an *AdaBoost.M*1 [FS97] based classifier system to evaluate each hypothesis. If the hypothesis is accepted, it can be used to generate more annotation hypotheses, or to compute features for the classification other annotation hypotheses. By the end, the annotation parser produces annotations that are grouped, typed and anchored to its context.

4. Annotation Features

For each hypothesis, the annotation parser computes both shape features, and contextual features. We use two types of shape features. The first group consists of inexpensive image-based shape features as introduced by Viola and Jones in [VJ01]. The second group of features are similar to the carefully designed geometric feauatures by Fonseca et al in [FPJ02]. The third group of features are the context-based features.

4.1. Geometric Features

All these geometric features are shape-related. Shape is an important clue to what the type of annotation could be. The following are examples of the geometric features used in the annotation engine:

1. **Aspect Ratio:** the aspect ratio of the minimal enclosed rectangle is used as a feature to estimate the "likelihood" of a shape being a line segement.
2. **Total Curvature:** the sum of curvature changes of the stroke(s) as it (they) forms the geometric shape.
3. **Total Turning Angle:** the sum of angle changes of the vertices in relinked polyline (in Radian).

4. **Curvature Profiles:** we divide the baseline (the major axis) of the geometric shape formed by the stroke into two or three buckets and compute the change of curvatures in each bucket.
5. **Horizontal Density:** The ratio between the *absolute horizontal movement* and the width of the minimal enclosed rectangle.
6. **Start-End Distance Ratio:** The ratio between the distance between the start and the end vertices and the width of the minimal enclosed rectangle–to measure the "closedness" of the shape.
7. **Shape Open Sided:** an heuristic binary feature, true when the polyline is open to a side (like for a paranthesis, a brace, or bracket)
8. **Open To Left Side:** a binary feature, heuristic, true when the polyline is open to the left side
9. **Side-Center Distance Ratio:** The distance between the mid-point of the open side and the center of the baseline, normalized by the width of the baseline.
10. **Maximal Inscribed Triangle Area Ratio:** Area of the maximal inscribed triangle of the convex hull of a stroke, divided by the area of its convex hull.

4.2. Context Features

As in [SW04], the annotation parser not only evaluates each hypothesis according to its geometric shape, but also according to its spatial context. However, unlike in [SW04], the context also contains ink strokes parsed from engines earlier in the stack. The ink context contains writing grouped into words, lines, and paragraphs that earlier engine has parsed with high confidence. It is the annotation parser's job to determine whether ambiguous strokes from the previous stages are actually annotations or are simply part of the notes.

All of the previous parsing results can be used to reduce the hypothesis space. For example, a straight line segment that is nowhere near a writing region is very unlikely to be a horizontal range. A straight line segment that is to the right and to the left of a writing region, and is perpenticular to its major axis, is very likely to a vertical bar than a horizontal range. In the annotation parser, these important contextual clues are captured through carefully designed contextual features, and fed into the classifier system, let it to determine the relative importance of each feature, and arbitrate between each hypothesis.

There are four different types of contextual feature. For each of the four categories of annotations we support, we designed a set of contextual features that are specific to the category. For example, if a stroke or a group of strokes form an enclosure, one important information is that how much "context" it contains. Since we have the structure of the underlying document, so we can search through the partial parse tree, and determine how many words, lines, or paragraphs in the tree fall into the polygon shape formed by the strokes.

As an illustration, Section 4.2.1 and Section 4.2.2 list the

contextual features designed for horizontal ranges and vertical ranges respectively.

4.2.1. Context Features for Horizontal Ranges

For horizontal ranges, we use two different groups of contextual features, one group with respect to the line above the horizontal range, and one group with respect to the line lying under the horizontal ranges. For each group, we compute the following features:

1. **Existence of context line:** true if there is an anchor line—an underlying line or an above line, respectively.
2. **Angle Difference:** the angle difference between the baseline of theEnclosures anchor line, and the baseline of the annotation, rounded to $\left(-\frac{\pi}{2}, \frac{\pi}{2}\right]$.
3. **Anchor Line Center to Baseline Distance Ratio:** The distance between the center of the anchor line to the baseline of the annotation, normalized by the height of the anchor line.
4. **Baseline Center to Anchor Line Distance Ratio:** The distance between the center of the annotation's baseline to the baseline of the anchor line, normalized by the height of the anchor line.
5. **Anchor Line to Baseline Width Projection Ratio:** Project the baseline of the anchor line to the baseline of the annotation, and compute the ratio of the length between the projected line segment, and the length of the baseline it is projected to.
6. **BaseLine to Anchor Line Width Projection Ratio:** Same as above except the baseline of the annotation is projected to the baseline of the anchor line.

4.2.2. Context Features for Vertical Ranges

For common vertical ranges such as parantheses, braces and brackets, the shape itself is often a sufficient clue for determining its type. But for vertical bars as in Figure 4, it is very difficult to differentiate them from vertical dividers (as the vertical green line in Figure 7, without using context information such as which words, lines or paragraphs they refer to.

Frequently, a vertical range has lines of context on both sides. For vertical ranges such as braces, brackets, and parantheses, most of times, it is easy to determine which side is the open side, and which side is the back side, and thus which set of lines to anchor to. But for vertical bars, it is very difficult to determine which set to anchor to, without looking at context features that are computed with respect to both sets of lines.

1. **Number of Overlapped Lines:** Number of lines in the set that is vertically overlapped with the baseline of the vertical range
2. **Angle Difference with Anchor Block:** The angle difference between the baseline of vertical range and the vertical axis of the neighboring block
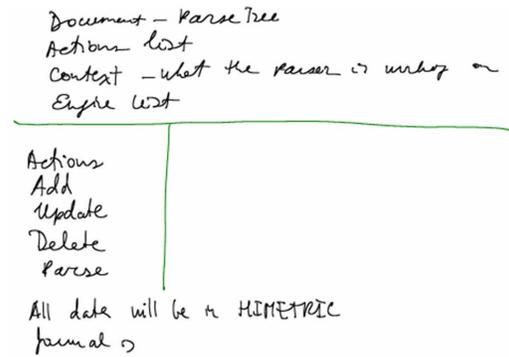

Figure 7: *With only "shape" information and no context information, it is very difficult to differentiate a vertical divider from a vertical bar.*

3. **Average Line Distance Ratio:** The average of distance from the start or end side of each vertically overlapped line in the set to the baseline of the vertical range.
4. **Sum of Vertical Overlap:** The sum of the vertical overlap of each neighboring line, normalized by the length of the baseline of the vertical range.

4.3. Context Feature Computation and Errors of Previous Engine

As shown in Section 4.2, the computation of context features utilizes parsing results of previous engines in the engine stack. But what if these engines make errors? Fed with the wrong values of the features, can the classifier still make the correct prediction? If we train our annotation parser with only the correctly labeled files, it is very likely for the classification system to produce poor results, since they have never seen these erratic configurations of feature values before.

The trick here is to train the annotation parser with partial parsing results from the previous engines instead of the correctly labeled files only. In fact, if we can predict the exact distribution of the annotation scenarios that the annotation parsing system will encounter when it is released to real world users, and if we have an unlimited amount of training data, it is better that we train with partial parsing results only. But since we do not know what the actual distribution will be, we train also with the labeled files, hopefully introducing a bias toward the more correct configurations of feature values.

5. Results

The annotation parsing system described here will be exposed through the Tablet PC Ink Analysis SDK for the development of ink applications for Tablet PC. It will be available with Windows Vista®. And in addition, it is also part

of the entire ink parsing system that is used by the next version of OneNote® also to analyze ink and mixed ink and text documents.

5.1. Evaluation

To evaluate the system, we collected a large set of OneNote files from Microsoft employees who use OneNote as part of their day-to-day work. Many of these files contain annotations as described in this paper. To increase the size of our data set, we also had users create semi-natural annotations on documents. By semi-natural we mean that we asked them to perform natural tasks ("correct spelling errors in the third paragraph", "indicate that the author should move Figure 3 to the top of the page") without telling them exactly which annotations to use. Then for all these files, we labeled the annotations and their anchors to generate a ground truth data set of 1294 files. These files containing 6974 examples of annotations. Out of these examples, 1413 examples are set aside for cross-validation.

After training the engine, we tested its accuracy on another test set of 138 files. Parsing result of an actual file is shown in Fig 8.
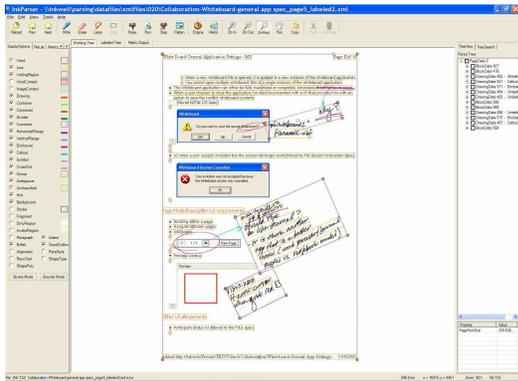

Figure 8: *Parsed Real World Example.*

To simplify the presentation of the results, we merge the results on vertical range and vertical bar together, and the results on callout with arrow and callout without arrow into one. The distribution of examples in this set is given as: $(22.08\%, 34.59\%, 4.57\%, 3.82\%, 5.15\%, 13.4\%, 16.39\%)$ for non-annotation drawings, underlines, strike-throughs, scratch-outs, enclosures, vertical ranges, and callouts respectively.

Table 2 shows the confusion matrix on the test set. The rows represent the labeled annotations, and the columns represent the parsed annotation results. For example, the cell $(1,1)$ shows the percentage of examples of non-annotation drawings correctly classified as non-annotation drawings.

Table 1: *Semantic Recall Results.*

File Type	Priority 1	Priority 2
Lightly Annotated	96.34	92.00
Highly Annotated	91.67	84.75
Mixed Ink and Text	91.30	76.87

The cell at $(2,3)$ shows the percentage of examples of underline misclassified as strike-through. The black-fonted number in each row is the recall number for that type of annotation. For example, the recall of underline is 98.02%.

On the average, the annotation parser has achieved an average recall of 0.9258 on all annotation types. Unlike many research findings in this area, which report accuracy numbers on a predefined set of pre-segmented symbols, these numbers are based on real user notes. Real users do not obey any fixed conventions when they take notes, and can be arbitrarily messy. Furthermore, the numbers reported here are a function not just of the annotation recognizer, but of the entire stack of engines that come before it. Given today's state of the art, a learning-based system that performs with 92.5% accuracy across a wide set of user notes is remarkable.

The errors shown here are not surprising. The largest number of misclassifications is between strike-through and scratch-out. Since both annotations indicate deletion, this error would actually not affect any user experience. The second largest confusion is underlines misrecognized as strikethroughs and vice versa. Such confusion is natural for a human reading an annotated paper, and is disambiguated using the underlying semantics of the document, or using higher-level context then we employ.

The worst-looking number in the confusion matrix is the 54% of drawing strokes that are misinterpreted as annotations. This number is poor but it is also misleading. Because only 4drawings, the actual number of errors is minor relative to the overall number of annotations processed.

6. Future Work

While we believe this system significantly advances the state of the art in processing handwritten annotations, it also opens new problems. On the recognition side, we would like to recognize increasingly more sophisticated annotation structures, including linkages between containers, callouts, ranges, and so on. By performing these linkages as part of the optimization strategy, we should be able to improve the system accuracy. We also believe that in the long-term, our feed-forward, greedy, multiple engine recognition strategy limits accuracy, but see no obvious ways to get around this without significantly reducing system performance. Another set of issues that we do not address in this paper is appropriate user interfaces for exposing and mediating the recognition results. In this work we present our best effort at provid-

Table 2: *Recognition Results on the Test Set.*

Labeled	Drawing	Underline	Strike-through	Scratch-out	Enclosure	Vertical Range	Callout
Underline	0.0099	**0.9802**	0.0035	0.0023	0.0006	0	0.0035
Strike-through	0.0176	0.0441	**0.8062**	0.1101	0.0132	0	0.0088
Scratch-out	0.0211	0.0053	0	**0.9474**	0.0053	0	0.0211
Enclosure	0.0078	0	0	0.0117	**0.9688**	0	0.0117
Vertical Range	0.0180	0	0.0015	0	0	**0.9099**	0.0706
Callout	0.0172	0.0147	0.0037	0.0025	0	0.0196	**0.9423**
Drawing	**0.4572**	0.1202	0.2996	0.0592	0.0082	0.0118	0.0437

ing a real-time recognition, but do not address the system's usability in the presence of errors.

.

References

[AD05] ALVARADO C., DAVIS R.: Dynamically constructed bayes nets for multi-domain sketch understanding. In *Proceedings of IJCAI-05* (San Francisco, California, August 1 2005), pp. 1407–1412.

[AVK93] APTE A., VO V., KIMURA T. D.: Recognizing multistroke geometric shapes: An experimental evaluation. In *ACM Symposium on User Interface Software and Technology* (1993), pp. 121–128.

[BMP02] BELONGIE S., MALIK J., PUZICHA J.: Shape matching and object recognition using shape contexts. *IEEE Trans. Pattern Anal. Mach. Intell. 24*, 4 (2002), 509–522.

[BSH04] BISHOP C. M., SVENSEN M., HINTON G. E.: Distinguishing text from graphics in on-line handwritten ink. *iwfhr* (2004), 142–147.

[CSKK02] CALHOUN C., STAHOVICH T. F., KURTOGLU T., KARA L. B.: Recognizing multi-stroke symbols. In *AAAI Spring Symposium, Sketch Understanding* (2002), pp. 15–23.

[FPJ02] FONSECA M. J., PIMENTEL C., , JORGE J. A.: Cali: An online scribble recognizer for calligraphic interfaces. In *AAAI Spring Symposium, Sketch Understanding* (2002), pp. 51–58.

[FS97] FREUND Y., SCHAPIRE R. E.: A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci. 55*, 1 (1997), 119–139.

[HD03] HAMMOND T., DAVIS R.: LADDER: A language to describe drawing, display, and editing in sketch recognition. *Proceedings of the 2003 Internaltional Joint Conference on Artificial Intelligence (IJCAI)* (2003), 461–467.

[Kar04] KARA L. B.: *Automatic Parsing And Recognition Of Hand-Drawn Sketches For Pen-Based Computer Interfaces*. PhD thesis, Department of Mechanical Engineering, Carnegie Mellon University, Pittsburg, PA, 2004.

[Mar97] MARSHALL C.: Annotation: from paper books to the digital library. In *Proceedings of the ACM Digital Libraries Conference* (1997).

[ÖÖT*01] ÖZER Ö. F., ÖZÜN O., TÜZEL C. Ö., ATALAY V., ÇETIN A. E.: Vision-based single-stroke character recognition for wearable computing. *IEEE Intelligent Systems 16*, 3 (2001), 33–37.

[PdFJ02] PIMENTEL C. F., DA FONSECA M. J., JORGE J. A.: Experimental evaluation of a trainable scribble recognizer for calligraphic interfaces. In *Lecture Notes in Computer Science: Graphics Recognition, Algortihms and Applications : 4th International Workshop, GREC 2001,* (2002), vol. 2390, pp. 81–91.

[SV04] SHILMAN M., VIOLA P.: Spatial recognition and grouping of text and graphics. In *1st Eurographics Workshop on Sketch-Based Interfaces and Modeling* (2004).

[SVC04] SHILMAN M., VIOLA P., CHELLAPILLA K.: Recognition and grouping of handwritten text in diagrams and equations. In *Ninth International Workshop on Frontiers in Handwriting Recognition (IWFHR'04)* (2004), pp. 569–574.

[SW04] SHILMAN M., WEI Z.: Recognizing freeform digital ink annotations. In *Document Analysis Systems VI* (2004), pp. 322–331.

[SWR*03] SHILMAN M., WEI Z., RAGHUPATHY S., SIMARD P., JONES D.: Discerning structure from freeform handwritten notes. In *ICDAR* (2003), pp. 60–65.

[VJ01] VIOLA P. A., JONES M. J.: Robust real-time face detection. In *ICCV* (2001), p. 747.

[Wen03] WENYIN L.: On-line graphics recognition: State-of-the-art. In *GREC* (2003), pp. 291–304.

[YSR*05] YE M., SUTANTO H., RAGHUPATHY S., LI C., SHILMAN M.: Grouping text lines in freeform handwritten notes. In *ICDAR* (2005), pp. 367–373.

[YV04] YE M., VIOLA P.: Learning to parse hierarchical lists and outlines using conditional random fields. *iwfhr* (2004), 154–159.