Layout Evaluation Library

Christian Clausner

University of Salford

Document History

Date	Action	Author
04.02.2010	created	Christian Clausner
17.06.2010	Reading Order Evaluation Alg.	Christian Clausner
24.06.2010	Allowable Merge/Split Detect.	Christian Clausner
25.06.2010	Evaluation Raw Data	Christian Clausner
01.07.2010	Evaluation Error Metrics	Christian Clausner
05.07.2010	F-Measure	Christian Clausner
26.07.2010	New merge and split formulas	Christian Clausner
11.08.2010	New formulas for overall success rates, ...	Christian Clausner
13.06.2011	Extended allowable check	Christian Clausner
23.08.2011	Border evaluation	Christian Clausner
2018	Nested regions	Christian Clausner

Contents

Open Issues, Proposals, ...

Problems for the Usage of the Number of Foreground Pixels Instead the Region Area

The idea of using the region area as weight for segmentation errors was that large regions are more important than small regions. Using the region area itself (defined by the polygon) can be problematic, if the polygons cover a lot of white space around the actual region object (e.g. text). That's why the 'Use Pixel Area' option has been introduced, where only the area of foreground pixels is used (black pixels within the polygon). However, this can cause problems for regions with above-average black pixels (e.g. images with dark background). These regions will be weighted much more than a normal text region for instance.

A solution would be (as proposed in Dave's thesis) to use the foreground pixel area only for text regions.

Evaluation Raw Data

The evaluation raw data is a collection of basic comparison data between ground-truth and segmentation result. The final evaluation metrics are based on the raw data.

Following data is collected / calculated:

- region overlaps (involved regions, area, number of foreground pixels)
- region area
- number of foreground pixels
- merges (allowable, non-allowable) – involved regions, area
- splits (allowable, non-allowable) – involved regions, area
- misses – regions
- partial misses – involved regions, area
- misclassification – regions
- false detection – regions

Note: The area is either the actual region area defined by the polygon or the number of black pixels within this area. There is a flag in the evaluation profile to specify which of the two options to use ('Use Pixel Area').

Data Structures for Raw Data

CLayoutEvaluation

- Ground-truth and segmentation result file name
- Document image file names
- map of CEvaluationResults per region type (block, text line, word, glyph)

CEvaluationResults

- Overlap maps (which segmentation result region overlaps which ground-truth region and vice-versa)
- CReadingOrderEvaluationResult (penalized reading relations between segmentation result regions)
- map of CRegionEvaluationResult per region (all errors for a region: merges, splits, ...)

CRegionEvaluationResult

- map of CRegionEvaluationError per error type (MERGE, SPLIT, ...)

CReadingOrderEvaluationResult

- collection of CReadingOrderError (accessible by involved region)

CReadingOrderError

- involved segmentation result regions (pair)
- relation between the regions
- penalty (of comparison to ground-truth reading order relation)

CRegionEvaluationError

- Base class for region errors
- area
- number of foreground pixels
- collection of rectangles representing the error area
- weighted area (error value)
- weighted count (error value)

CEvaluationErrorMerge

- map with the merging segmentation result regions and the ground-truth regions merged with the ground-truth region of the error (includes overlap information)
- map with the merged ground-truth regions and a flag if the merge was allowable (see chapter on allowable merge and split)

CEvaluationErrorSplit

- segmentation result regions the ground-truth region is split into (includes overlap information)
- flag if the split is allowable (see chapter on allowable merge and split)

CEvaluationErrorMisclass

- misclassified segmentation result regions that overlap the ground-truth region (includes overlap information)

Collecting the Raw Data

First all ground-truth regions are processed and the overlapping segmentation result regions are determined. Form that step we get the overlap lookup tables (which ground-truth region overlaps with which segmentation result region and vice versa).
Afterwards we use the overlap information to find the layout errors (merge, split, ...).

```
ProcessGroundTruthRegions(regionType)
{
  Make sure all region polygons are isothetic and loop-free;

  Calculate bounding box map (BBMap) for the segmentation result regions;

  for (each ground-truth region G)
  {
    Calculate and save the interval representation for G;
    overlapCandidates = BBMap.GetOverlappingRegions(G);
    regionList = {};

    for (each region S in overlapCandidates)
    {
```

```
      Calculate and save the interval representation for S;
      overlap = RegionOverlap(G.intervalRep, S.intervalRep);

      if (overlap.AreRegionsOverlapping())
        Save the overlap;
        regionList = regionList + S;
    }

    multiOverlap = RegionOverlap(G, regionList);
    Save the multiOverlap;
  }
}

FindErrors(regionType)
{
  for (each ground-truth region G)
  {
    Get the overlapping segm. result regions listS from the raw data;
    CheckMerge(G, listS);
    CheckSplit(G, listS);
    CheckMiss(G, listS);
    CheckPartMiss(G, listS);
    CheckMisclass(G, listS);
  }

  for (each segmentation result region S)
  {
    Get the overlapping ground-truth regions listG from the raw data;
    CheckFalseDetection(S, listG);
  }

  if (regionType == BLOCKS)
    EvaluateReadingOrder();
}
```

Finding Merges

A ground truth region is connected to another ground truth region via a segmentation result region. The first ground truth region and all overlapping segmentation result regions are given. So we have to check for each segmentation result region if it overlaps another ground truth region other than the given one.

```
CheckMerge(ground-truth region G, overlapping segm. result regions listS)
{
  for (each segmentation result region S in listS)
  {
    Get the overlap between S and G1 (overlap1);
    listG = S.GetOverlappingGroundTruthRegions;
```

```
    for (each ground-truth region G2 in listG)
    {
      if (G != G2)
      {
        Get the overlap between S and G2 (overlap2);
        Save the overlap information of the smaller overlap;
        CheckIfMergeAllowable(G, G2, readingOrder);
      }
    }
  }

  CheckForFalseAlarm();
  Save the evaluation error;
}
```

Finding Splits

A split is given if a ground-truth region overlaps more than one segmentation result region.

```
CheckSplit(ground-truth region G, overlapping segm. result regions listS)
{
  Save number of split regions (lists.GetSize);

  allowable = true;
  for (each S in listS)
  {
    Get the overlap between G and S;
    Save the overlap information;

    if (allowable)
    {
      for (each S2 in lists)
      {
        if (S != S2)
         if ( NOT CheckIfMergeAllowable(S, S2, G.readingDirection,
                                        G.readingOrientation);
           allowable = false;
    }
  }

  CheckForFalseAlarm();
  Save the evaluation error;
}
```

Finding Misses

A miss is given if a ground-truth region is not overlapped by any segmentation result region.

```
CheckMiss(ground-truth region G, overlapping segm. result regions listS)
{
  if (listS.IsEmpty)
  {
    Save miss evaluation error;
  }
}
```

Finding Partial Misses

To find partial misses we investigate the multi overlap interval representation of the regarded ground-truth region and all overlapping segmentation result regions. Partial misses are the interval parts that only belong to the ground-truth region.

```
CheckPartMiss(ground-tr. region G, overlapping segm. res. regions listS)
{
  overlap = GetRegionOverlap(G, listS);

  rects = overlap.GetUniqueRects(G); //Intervals that belong to only one region

  if (NOT rects.IsEmpty)
  {
    CheckForFalseAlarm();
    Save the information;
  }
}
```

Finding Misclassifications

To find a misclassification we have to check the region type of for each segmentation result region that overlaps the regarded ground-truth region and compare it to the ground-truth region type.

```
CheckMisclass(ground-tr. region G, overlapping segm. res. regions listS)
{
  for (each segmentation result region S in listS)
  {
    if (G.type != S.type)
      Save misclass information;
  }
  CheckForFalseAlarm();
  Save the evaluation error;
}
```

Finding False Detections

False detections are segmentation result regions that are not overlapped by any ground-truth region.

```
CheckFalseDetection(seg. res. reg. s, overlapping gr.-tr. regions listG)
{
  if (listG.IsEmpty)
  {
    Save false detection evaluation error;
  }
}
```

Evaluation Statistics and Error Metrics

Based on the raw data, the actual error values and success rates are calculated (see class CEvaluationMetrics). The metrics are calculated for a specified region type level (block, text line, word or glyph) and they are stored in the CEvaluationResults object for this region type level.
Following figures are produced:

- Statistics
    - Image Area
    - Number of black pixels within the image
    - Overall number of regions (ground-truth and segmentation result)
    - Number of regions per type (text, image, ...) (ground-truth and segmentation result)
    - Overall region area (ground-truth and segmentation result)
    - Overall number of black pixels in regions (ground-truth and segmentation result)
- Metrics
    - Overall weighted area error per error type (merge, split, ...)
    - Overall weighted area error per region type (text, image, ...)
    - Overall weighted area error
    - Weighted area success rate per error type
    - Overall weighted count error per error type (merge, split, ...)
    - Overall weighted count error per region type (text, image, ...)
    - Overall weighted count error
    - Weighted count success rate per error type
    - Reading order error
    - Reading order success rate
    - Overall weighted area success rate (arithmetic mean, harmonic mean)
    - Overall weighted count success rate (arithmetic mean, harmonic mean)
    - Recall per type
    - Recall (strict / non-strict)
    - Precision per type
    - Precision (strict / non-strict)
    - F-Measure (strict / non-strict)
    - Region count deviation (absolute / relative)

The same figures are also calculated for each region type (text, image, ...) separately. Only merges are a small exception here. If we look for the merge errors of graphics, we also take into account merges of graphic regions with other region types (e.g. separator).

The following paragraphs describe the different values in detail.

Image Area

Image Width * Image Height

Number of Black Pixel within the Image

Number of black pixels within the black-and-white image.

Overall Number of Regions

Number of regions of the chosen type level (block, text line, word or glyph) within the document layout.

Number of Regions per Type

Number of regions of for each region type (text, image, ...) within the document layout. This value is only available in block level and not in text line, word or glyph level.

Overall Region Area

The combined area of all regions regarded for the 'Overall Number of Regions' value. Possible region overlaps are not left out. So if there are overlaps, some image parts are counted twice.

Overall Number of Black Pixels in Regions

The combined count of black pixels of all regions regarded for the 'Overall Number of Regions' value. Possible region overlaps are not left out. So if there are overlaps, some image parts are counted twice.

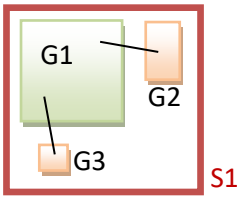
Weighted Errors

For the weighted area and count the weights defined in the evaluation profile are being used (see the chapter 'Evaluation Profile' for details).
There are two types of weighted errors: the 'Weighted Area' and the 'Weighted Count'. The weighted area is based on the assumption that bigger regions are more important than smaller ones. The error value is the region area (or the number of black pixels) multiplied with the weight. The weighted count only takes into account the error quantity. A misclassified region for instance is counted as one. A ground-truth region split into three regions is counted as 3. The count is then also multiplied with the weight.

Remarks:

Merge: For merge errors all possible pairs of the involved ground-truth regions are evaluated. For each pair of regions the smaller of the two overlaps with the merging segmentation result region is used. In the following example the regions G1, G2 and G3 are merged by region S1.

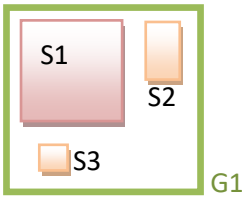
From the view of ground-truth region G1 the overlaps G2S1 and G3S1 are used for the weighted area error.



Split: The weighted area error for split has been extended by the count and relative split area. The count is the same as for the weighted count error, i.e. the number of regions the regarded ground-truth region has been split into. The relative split area is the area of the largest single split area (overlap) divided by the total split area (overlap). The final formula for the weighted area error is then:

error = weight * (1 – relative split area) * split count * area of the split region

In the following example ground-truth region G1 has been split into three regions (S1, S2 and S3). S3 has the largest overlap with G1 and is therefore used for the relative split area.



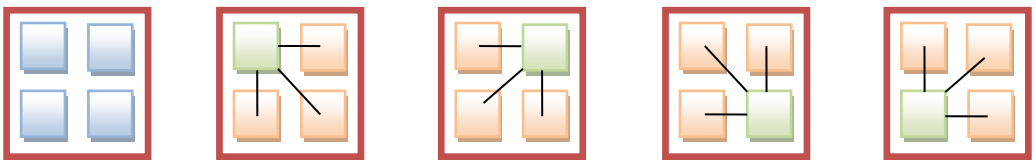
The Algorithm:

```
CalculateWeightedErrors()
{
  for (each region error in the raw data)
  {
    if (!error.IsFalseAlarm())
    {
      if (error.type == MISS or PART_MISS or FALSE_DETECTION)
      {
        region = error.EffectedRegion;
        weight = profile.GetErrorTypeWeight(error.type, region.type,
                                            [region.subtype]);
        weight = weight * profile.GetRegionTypeWeight(region.type,
                                                      [region.subtype]);
        weightedArea = weight * region.area;
        weightedCount = weight * 1;
      }
      else if (error.type == SPLIT)
      {
        region1 = error.EffectedRegion; //(ground-truth region)

        segOverlaps = error.GetSplittingRegionOverlaps();
```

```
  maxSplitArea = 0;
  totalSplitArea = 0;
  for (each overlapped seg. result region2 in overlap)
  {
    currentArea = overlap.GetOverlapArea(region2);
    if (currentArea > maxSplitArea)
      maxSplitArea = currentArea;
    totalSplitArea = totalSplitArea + currentArea;
  }
  relativeSplitArea = maxSplitArea / totalSplitArea;

  allowable = error.IsAllowable();
  weight = profile.GetErrorTypeWeight(error.type, region.type,
                                      [region.subtype], allowable);
  weight = weight * profile.GetRegionTypeWeight(region.type,
                                                [region.subtype]);
  weightedArea = weight * (1-relativeSplitArea)
                        * (ln(error->GetSplitCount())+0.31)
                        * region1.area;

  weightedCount = weight * error->GetSplitCount();
}
else if (error.type == MERGE)
{
  region1 = error.EffectedRegion; //(ground-truth region)

  segOverlaps = error.GetMergingRegionOverlaps(); //(overlaps of segm.
                                                   result regions with
                                                   ground-truth regions)
  weightedArea = 0;
  weightedCount = 0;
  for (each overlap in segOverlaps)
  {
    overlapWeight = 1 / (overlap.size() - 1); (*)

    for (each overlapped ground-truth region2 in overlap)
    {
      if (region1 != region2)
      {
        allowable = error.IsAllowable();
        weight = profile.GetErrorTypeWeight(error.type,
                                            region1.type,
                                            [region1.subtype],
                                            region2.type,
                                            [region2.subtype]);
        weight = weight * profile.GetRegionTypeWeight(region1.type,
                                                      [region1.subtype]);
        weight = weight * overlapWeight;
```

```
                weightedArea = weightedArea + weight *
                               overlap.GetOverlapArea(region2);
                weightedCount = weightedCount + weight * 1;
              }
            }
          }
        }
        else if (error.type == MISCLASS)
        {
          //Equivalent to the above
        }

        Increase the overall errors per error type and per region type;
      }
    }

    Calculate the success rates per error type using specialized functions;
}
```

(*) The additional **'overlap weight'** used for the merge errors prevents counting merge errors to be counted multiple times. Evaluation errors are organized by ground-truth region. As a merge involves two or more ground-truth regions, there are equally many errors. A merge of two regions is unproblematic but if there are more regions, the error piles up. For some examples the resulting error area was bigger than the image area! See following simple example:



Four ground-truth regions are merged by one segmentation result region. Each ground-truth region is merged with three other ground-truth regions (form its point of view). Overall that results in 12 single merges. We cannot treat the merge as one, because there may be different weights for the merge of region 1 with region 2 and region 3 with region 2 and so on. Instead we introduce an additional weight based on the number of involved regions. In the end we get a value that represents the merge as one entity.

Functions for the Success Rate

Depending on the error type, different functions are used to calculate the success rate. The basic function type used is an inverse (f(x) = a/x). The idea behind was to have good contrast for small to medium error values and still be able to handle high error values. As there is no upper limit for many error types (e.g. a ground-truth region can be split into two regions or 1000 regions), a linear relation between error value and success rate is not suitable. An inverse function gets zero in the infinity; that means the higher the x value (error value) the nearer to zero is the resulting value (success rate).

We also tried a logarithmic function to achieve a similar result, but logarithmic functions fall to much at the beginning and too little for higher x values.
The general formula used for the success rate function is

success_rate = 1 / (a * error_value + 1)

The factor 'a' scales the function in x direction. Values smaller than 1 stretch the function graph and values greater than 1 compress the graph. For more convenience there is a wrapper class (CInverseSuccessRate) for the success rate function that can be initialized by a parameter called '50 percent X'. This parameter is the x value (error value) where the resulting success rate should be 50 percent. The factor 'a' is then calculated that the function does exactly that. The formula is simply:

```
if (fiftyPercentX < 1)
      fiftyPercentX = 1;
a = 1 / fiftyPercentX;
```

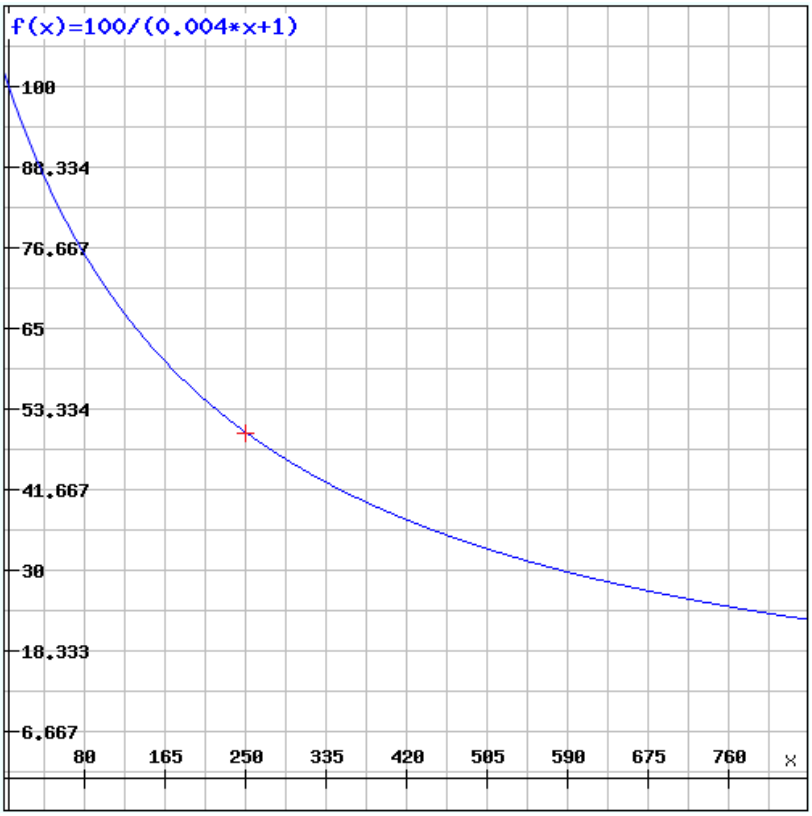
The value of the '50 percent X' depends on the error type. See following table for the different values:

Error Category	**Error Type**	**Fifty Percent X**	**Comments**
Weighted Area	false detection	image area / 10	10 % of the image area
	merge	overall region area / 4	
	split, miss, partial miss, misclassification	overall region area / 2	50 % of the combined area covered by all ground-truth regions
Weighted Count	false detection	# ground-truth regions	Experimental value
	split	# ground-truth regions	If half of the ground-truth regions are split in two we have an error value equal to the number of ground-truth regions, because each split is counted as 2.
	merge	# ground-truth regions / 2	If there are only merges with two regions involved, a number of merges equal to a fourth of the number of ground-truth regions would involve 50 % of all regions.
	miss, partial miss, misclassification	# ground-truth regions / 2	50 % of the number of ground-truth regions

The '50 Percent X' values are throughout based on ground-truth and image parameters. They differ from document image to document image but are independent from the segmentation result.

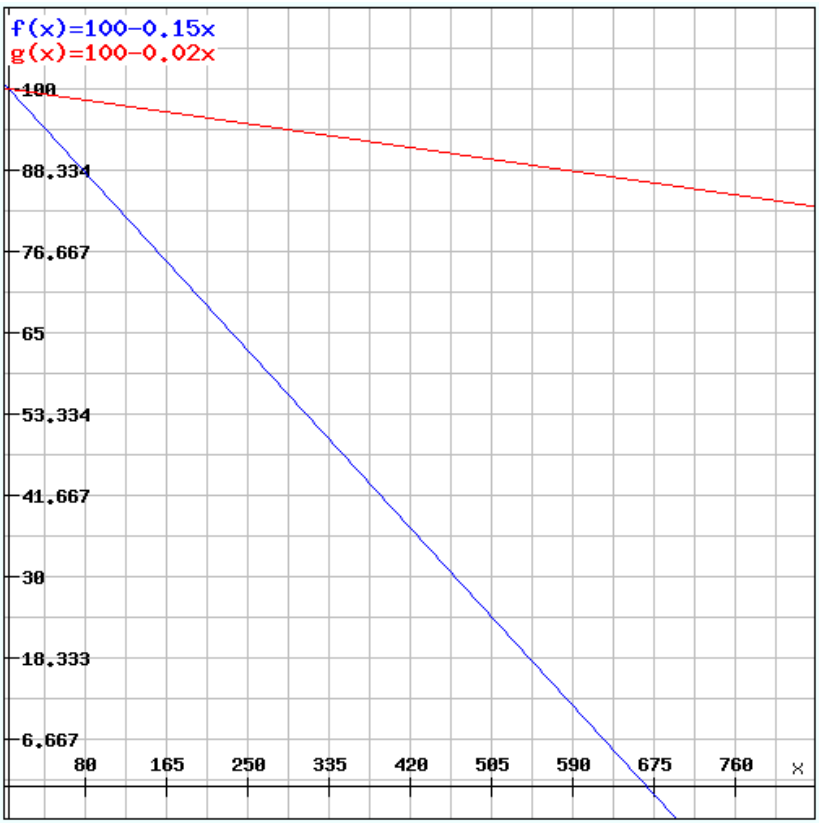
Note: For region type specific success rates, the overall region area and the number of ground-truth regions is constrained to the region type. That means if we calculate for instance the success rate for the weighted count for splits, the number of ground-truth regions means the number of text regions.

The following image shows the graph of a success rate function with 250 as ‘Fifty Percent X’ value. The y value is 50 % for an x value of 250. The contrast of two errors is higher for lower error values and gets smaller and smaller with increasing error values.



(http://rechneronline.de/funktionsgraphen/)

A linear function would either have a good contrast for small error values but would cut of the error after the zero point or it would have a bad contrast (see screenshot). No matter how, when using linear functions the contrast always gets zero for large error values.



Reading Order Error

See chapter ‘Reading Order Evaluation’.

Overall Success Rates

The overall success rates combine all error type success rates to one number. There are two different types of overall success rates: the arithmetic mean and the harmonic mean. And for each type there are again two success rates: One including the weighted area success rates and the reading order and the other one with the weighted count success rates and the reading order.

The general formula for the weighted arithmetic mean is:

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i \cdot x_i}{\sum_{i=1}^{n} w_i}.$$

Where n is the number of values, x_i are the values (in our case the success rates for the error types) and w_i are the weights. For the reading order the weight is directly the one defined in the evaluation profile. The other weights are defined by:

$w_i = (5 * (1-x_i) + 1) / 6$

This highlights low error type success rates and diminishes high success rates, without erasing them completely. The influence of a partial success rate to the overall rate is somewhere between 1/6 and 1.

The reason behind this extended formula is to filter out trivial segmentation results like an empty page or a text region for the whole page. If we wouldn't use the weights, these cases would be rated too well.
For example: A segmenter delivered an empty page. That means we have miss errors for all regions and nothing else. Merge, split, partial miss, misclassification and partial miss are all 100%. If we calculate the average of those partial success rates without weighting them, we get an overall success rate of about 83%. If we use the weights defined above, each partial success rate that is 100% only has an influence of 17% to the overall rate and the miss has a much bigger influence. Thus we would get an overall success rate of about 45%.
To resolve the above problem we could also have used $w_i = (1-x_i)$ as formula for the weights. But then we would lose the good success rates completely, what would cause problems for other examples. Let's say a segmenter delivers a result with merges for some regions (merge success rate 70%) and another segmenter has the same amount of merges plus some misclassifications (misclassification success rate 70%). If we use the formula that erases the good partial success rates we get an overall success rate of 70% for both result, though the result with merge and misclassification is obviously worse.
The extended weight formula guarantees a minimum of 1/6 per weight. Hence we don't lose the good partial success rates completely.

The general formula for the weighted harmonic mean is:

$$\frac{\sum_{i=1}^{n} w_i}{\sum_{i=1}^{n} \frac{w_i}{x_i}}.$$

Note: If one of the error type success rates is zero, the harmonic mean is also zero.

The harmonic mean is always smaller than or equal to the arithmetic mean.

Precision and Recall

Precision and Recall are generally defined as follows (Wikipedia):

Precision is the number of relevant documents retrieved by a search divided by the total number of relevant documents.

Recall is the number of relevant documents retrieved by a search divided by the total number of documents retrieved by that search.

In terms of document image evaluation this can be interpreted as follows (example for text regions):

Recall is the number of pixels within ground-truth text regions that are also within a text-region in the segmentation result divided by the overall number of pixels in ground-truth text regions.

Precision is the number of pixels within ground-truth text regions that are also within a text-region in the segmentation result divided by the overall number of pixels in segmentation result text regions.

For the overall recall and precision we differentiate between strict and non-strict. For the strict recall and precision the region type must be matched correctly. That means a ground-truth text region overlapped by a segmentation result image region does not count as recall. For non-strict recall and precision the region type doesn't matter. You could also say that strict means 'with classification' and non-strict means 'without classification'.

To evaluate precision and recall we can use the collected raw data. In short we investigate the overlap of each ground-truth region with segmentation result regions and count the recalled pixels. Possible overlapping regions within the segmentation result are not taken into account!
Note: Depending on the profile setting 'area' means the actual region area defined by its polygon or the number of black pixels within this area.

The algorithm:

```
CalculateRecallAndPrecision()
{
  for (each ground-truth region G)
  {
    groundTruthAreaPerType[G.type] += G.area;
    overlaps = GetOverlappingSegmentationResultRegions(G);

    if (overlaps.size > 0)
    {
      recallAreaStrict[G.type] += GetStrictRecallArea(overlaps, G);
      overallRecallAreaNonStrict += GetNonStrictRecallArea(overlaps, G);
    }
  }

  for (each segmentation result region S)
  {
    segResultAreaPerType[S.type] += S.area;
  }

  for (each regionType in recallAreaStrict)
  {
    overallRecallAreaStrict += recallAreaStrict[regionType];

    recallPerType[regionType] = recallAreaStrict[regionType] /
                                groundTruthAreaPerType[regionType];
    precisionPerType[regionType] = recallAreaStrict[regionType] /
                                   segResultAreaPerType[regionType];
  }

  recallStrict = overallRecallAreaStrict / overallGroundTruthRegionArea;
```

```
  recallNonStrict = overallRecallAreaNonStrict /
                    overallGroundTruthRegionArea;
  precisionStrict = overallRecallAreaStrict / overallSegResultRegionArea;
  precisionNonStrict = overallRecallAreaNonStrict /
                       overallSegResultRegionArea;
}
```

```
GetStrictRecallArea(overlaps, ground-truth region G)
{
  for (each interval)
  {
    if (G is assigned to the interval)
    {
      if (another region with the same type as G.type is assigned)
        count += interval.count;
    }
  }
  return count;
}

GetNonStrictRecallArea(overlaps, ground-truth region G)
{
  for (each interval)
  {
    if (G is assigned to the interval)
    {
      if (another region is assigned)
        count += interval.count;
    }
  }
  return count;
}
```

F-Measure

The F-Measure is defined by: 2 * precision * recall / (precision + recall).

Region Count Deviation

The region count deviation is simply the difference between the number of ground-truth regions and the number of segmentation result regions. These figures can be useful to detect trivial segmentation results like an empty page or one text region for the whole page.

absolute region count deviation = |#ground-truth regions - #segmentation result regions|

relative region count deviation = absolute deviation / #ground-truth regions

Note: If there are no ground-truth regions (i.e. blank page), the relative value is the same as the absolute value.

Evaluation Profile

The evaluation profile is used to specify weights and other parameters to customize the layout evaluation. In some scenarios text regions may be important, in others it may be image regions. The weights can be adjusted from a general level (e.g. merge) down to the most detailed level (e.g. merge of text paragraph with text headline).
A weight can have values from 0.0 to 10.0, whereas 1.0 is the default. A value of 0.0 means that the region or error type is not regarded at all for the evaluation results. A value higher than 1.0 means that the region or error type is emphasized in comparison to other region or error types.
The profile is stored together with the evaluation results. So the weights that were used for the evaluation can always be examined.
The profile will usually be defined in the Graphical Layout Evaluation Tool and can then be referenced in the command line tool.

The evaluation profile contains following weights and parameters:

- error type weights stored hierarchical per error type (merge, split, ...), region type (text, image, ...) and for text regions also subtype (paragraph, headline, ...). The weights for merge and split have an extra value for 'allowable' (see chapter on 'Allowable Merges and Splits').
- region type weights stored per region type (text, image, ...)
- reading order weight (influence of the reading order to the overall success rate)
- general parameters
    - 'Use Pixel Area' – if TRUE, the number of black pixels is used for the error calculations instead of the polygon area
    - 'Reading Orientation Threshold' – Threshold of how much the reading orientation of two regions can differ to be allowable (see chapter on 'Allowable Merges')
    - 'Default Reading Direction' – Used for 'Allowable Merge' detection. See next parameter
    - 'Reading Direction Usage' – Can be either 'Ground-truth' – always use reading direction as specified in the ground-truth; 'Default if not set in Ground-Truth' – uses the default value if the reading direction isn't defined for the regarded region; 'Default' – always uses the default value, regardless which value is defined for the regarded region
    - 'Default Reading Orientation' - Used for 'Allowable Merge' detection. See explanation above
    - 'Reading Orientation Usage' - Used for 'Allowable Merge' detection. See explanation above

Region Overlaps

To evaluate the layout of a segmentation result the ground truth regions have to be matched against the segmentation result regions. For several error types information is required on how many segmentation result areas overlap a ground truth region and what is the overlap area. Therefore as a first step the overlaps for each ground truth region are calculated (before the actual evaluation starts).

Checking every ground truth region against each segmentation result region would be very costly. That's why a global bounding box search map (CBoundingBoxMap) for the segmentation result regions is created. This map can then be used to determine overlap candidates for a specific ground truth region. Overlap candidates are segmentation result regions whose bounding box touches the bounding box of the ground truth region.
The bounding box map is similar to the interval representation of polygons. Instead of inserting lines to find region intervals, bounding boxes are put into the data structure. The result is a sorted list of y-intervals (rows), each with an own list of x-intervals (columns). An x-interval contains information on which layout regions belongs to it.
Given a ground truth region it is now easy and fast to find the overlap candidates.

Once the overlap candidates for a ground truth region are found, the actual overlaps are calculated. Again a data structure similar to the interval representation of polygons is used (CRegionOverlap). One or more interval representations are combined by inserting every interval part into a shared interval data structure with y-intervals (rows) and x-intervals (columns). The overlap information (overlap area, overlap rectangles) is collected on the fly. If the detailed overlap reveals that actually there is no overlap at all, the candidate is discarded.

The overlap relationships are stored in two maps. One containing which segmentation result regions overlap a specified ground truth region and the other map vice versa which ground truth regions overlap a specified segmentation result region:

map <ground truth region, **set<**segmentation result region**> >**
map <segmentation result region, **set<**ground truth region**> >**

Nested Regions

Regions within regions (nested regions) need to be evaluated differently because using the region correspondence approach, overlaps would be detected and wrongly penalised.

For the evaluation, top-level regions (parent regions) and nested regions (child regions) are treated as being in different layers. For each ground truth region, two error values are calculated: One in the same layer (top-level to top-level) and one across layer (top-level to nested or nested to top-level). The lower of the two is then used as final value for the region. This helps to avoid over-penalisation when the region structure of the page analysis result is different than the ground truth, but most of the content was recognised. For example, in the following figure, if the table region would have been missed, but the table cells were recognised (as

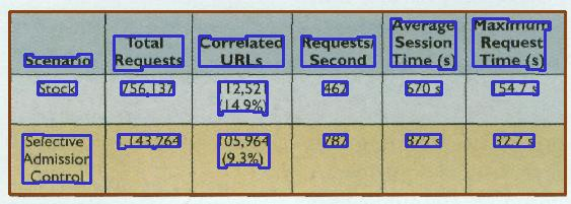
top-level regions), then the across-layer error would be lower. A penalty is already applied from the same-layer evaluation of the ground truth top-level table region.

Errors where nested regions are involved are globally weighted at 50% in combination with the normal weights defined by the evaluation profile.

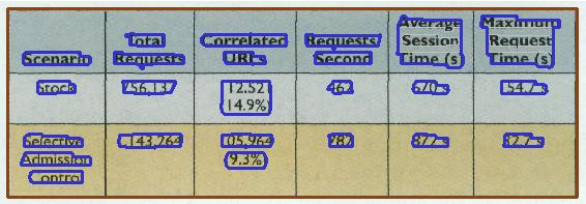
The evaluation of nested regions has to be enabled explicitly in the evaluation profile (general settings).

Scenario	Total Requests	Correlated URLs	Requests/ Second	Average Session Time (s)	Maximum Request Time (s)
Stock	756,137	112,521 (14.9%)	462	670 s	154.7 s
Selective Admission Control	1,143,264	105,964 (9.3%)	782	872 s	32.7 s

Ground Truth



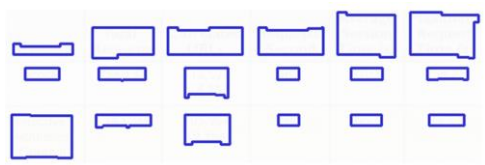
Page analysis result



Top-level ground truth regions

Top-level result regions

Nested ground truth regions



Nested result regions

Reading Order Evaluation

To evaluate the reading order of a segmentation result, an algorithm is used that investigates the relation of all possible region pairs in the segmentation result. A relation is penalized if there is a difference between the relation specified in the segmentation result reading order and the relation specified by the ground-truth reading order. The final reading order error is calculated by adding up all penalties.

Algorithm:

```
penalty = 0;

for each segmentation result text region S1
  for each segmentation result text region S2
    if (S1 != S2)
    {
      segRel = CalculateRelation(S1, S2, SegResult Reading Order);

      Find all ground-truth regions G1 that overlap S1 and all
      regions G2 that overlap S2.

      for (each combination G1 and G2)
      {
        gtRel = CalculateRelation(G1, G2, Ground-Tr. Reading Order);

        overlap1 = overlap area (S1, G1);
        overlap2 = overlap area (S2, G2);

        ws1 = overlap1 / S1.area;
        ws2 = overlap2 / S2.area;
        wg1 = overlap1 / G1.area;
        wg2 = overlap2 / G2.area;

        weight = (ws1 + ws2) / 2 * (wg1 + wg2) / 2;

        penalty = penalty + weight * CalculatePenalty(segRel, gtRel)
      }
    }
```

Calculating the Relation between two Regions Using the Reading Order

The relation is calculated by following the paths from the common parent group of both regions to their position in the reading order tree. Starting with all possible relations, impossible relations are sorted out on the way. The result is a set of possible relations. In the best case, such a set contains only one relation.

Possible base relations are (region R1 – R2):

```
->    R1 precedes R2 directly
<-    R1 succeeds R2 directly
--    unordered relation
->->  R1 is somewhere before R2
<-<-  R1 is somewhere after R2
-x-   R1 has no direct relation to R2
n.d.  the relation between R1 and R2 is not defined
-?-   unknown relation
```

```
CalculateRelation(Region R1, Region R2, ReadingOrder):
{
  if (R1 or R2 is not in the reading order)
    return [n.d.]

  Follow the reading order tree from the regions R1 and R2 to the
  root group and find the first group the two paths have in common
  (this is the common parent group).

  relations = [->, <-, --, ->->, <-<-, -x-];

  Starting from that group, follow the paths (path1, path2) back to
  the regions level by level and sort out impossible relations on
  the way by applying following rules:

  group1 = group2 = common parent group;

  indirect = false;  //'indirect' flag

  while(group1 != NULL or group2 != NULL)
  {
    if (group1 != NULL)
    {
       if (group1 is ordered)
       {
          if (group1 == group2) //both paths in the same group
          {
             if (path1 is before path2)
                relations = relations - [<-, <-<-];
                if (there is an object between path1 and path2)
                   relations = relations - [->, ->->];
             else if (path1 is after path2)
                relations = relations - [->, ->->];
                if (there is an object between path1 and path2)
                   relations = relations - [<-, <-<-];
          }
```

```
         else if (group1 != group2) //not in the same group
         {
            if (there is an object between path1 and path2)
               relations = relations – [<-, <-<-, ->, ->->];
         }
      }
      else //group1 is unordered
      {
         if (group1 == group2)
            if (path1 and path2 have the same position)
               if (group1.size > 1)
                  indirect = true;
            else (path1 and path2 have NOT the same position)
               if (group1.size > 2)
                  indirect = true;
         else
            if (group1.size > 1)
               indirect = true;
      }
   }

   if (group2 != NULL and group1 != group2)
   {
      if (group2 is ordered)
      {
         if (there is an object between path1 and path2)
            relations = relations – [<-, <-<-, ->, ->->];
      }
      else //group2 is unordered
         if (group2.size > 1)
            indirect = true;
   }

   if (there is a next group after group1 in path1)
      group1 = next group in path1;
   else
      group1 = NULL;

   if (there is a next group after group2 in path2)
      group2 = next group in path2;
   else
      group2 = NULL;
} //end while
```

Final sort out:

```
if (relations doesn't contain -> and ->-> or <- and <-<-)
   relations = relations – [--];
```

```
  if (not indirect)
  {
    relations = relations - [->->, <-<-]

    if (relations contains -> or <-)
      relations = relations - [--];
  }

  if (relations contains --)
    relations = relations - [->, <-, ->->, <-<-, -x-];
}
```

Calculating the Penalty for two Different Relations

A relation is here a set of possible base relations (see above). To calculate the penalty between two relations sets, each base relation of one set is compared to each base relation of the other set. The minimum penalty found is the resulting penalty for the relation sets.

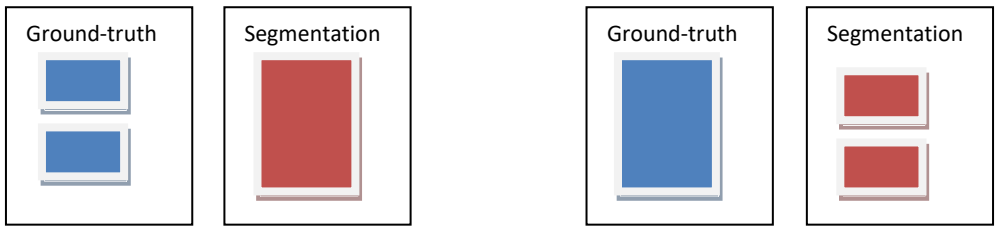
```
CalculatePenalty(segRel, gtRel)
{
  minimum = infinite;
  for (each base relation rel1 of segRel)
    for (each base relation rel2 of gtRel)
      penalty = BasePenalties[rel1, rel2];
      if (penalty < minimum)
        minimum = penalty;

  return minimum;
}
```

BasePenalties Matrix (may have changed):

```
                          //Ground-Truth
      //Seg.Result
                  /*          ->     <-    --      ?     -x-   n.d.   ->->    <-<-  */
      /*     */      {     0,     0,     0,     0,     0,     0,     0,     0,     0},

      /*  -> */      {     0,     0,    30,    10,     0,    20,     0,     0,    10},

      /*  <- */      {     0,    30,     0,    10,     0,    20,     0,    10,     0},

      /*  -- */      {     0,    20,    20,     0,     0,    10,     0,    10,    10},

      /*  ?  */      {     0,     0,     0,     0,     0,     0,     0,     0,     0},

      /* -x- */      {     0,    20,    20,    10,     0,     0,     0,    10,    10},

      /* n.d.*/      {     0,    20,    20,    10,     0,     0,     0,    10,    10},

      /* ->->*/      {     0,     0,    20,     5,     0,     5,     0,     0,    10},

      /* <-<-*/      {     0,    20,     0,     5,     0,     5,     0,    10,     0}
   };
```

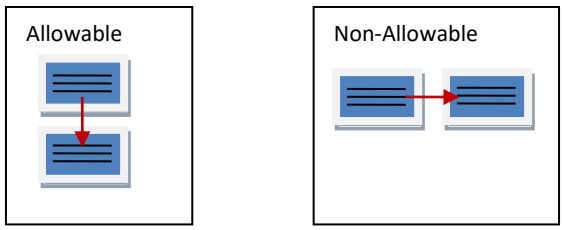
Detection of Allowable Merges or Splits

We speak of a *merge* if a region of the segmentation result overlaps two or more regions of the ground-truth. And we speak of a *split*, if a ground-truth region overlaps two or more segmentation result regions. See the following figure for examples:



Examples of a merge (left) and a split (right)

A merge is *allowable*, if the merged regions have a directed reading order relation and the relative positions to each other allow a merge regarding reading direction and orientation (the exact algorithm follows). For instance, for two paragraphs that are aligned vertically a merge is allowable (if they have a direct reading order relation and the reading direction is left-to-right). For two horizontally aligned paragraphs a merge is not allowable, even if they have a direct reading order relation (reading direction left-to-right). That is because an OCR method would probably mix up the lines of the left paragraph with the lines of the right paragraph and the resulting text would be completely wrong. See following example:



A split is allowable, if the regions involved regions of the segmentation result are in a relative position to each other, which allows merging them (regarding reading direction and orientation).

The general guideline is that the text an OCR engine would detect must be the same for the involved ground-truth regions and the segmentation result regions.

Algorithm for Allowable Merge

Ground-Truth regions G1 and G2.

```
CheckIfMergeAllowable(G1, G2, Ground-Truth ReadingOrder)
{
  if (G1 OR G2 not a text region)
    return false;

  allowable = true;

  hasRelationPredecessor = false;
  hasRelationSuccessor = false;
```

```
if (ReadingOrder contains references to G1 and G2)
{
  relation = ReadingOrder.CalculateRealtion(G1, G2);
  hasRelationPredecessor = relation.Contains(->);
  hasRelationSuccessor = relation.Contains(<-);

  if (!hasRelationPredecessor && !hasRelationSuccessor)
    allowable = false;
}
else
  allowable = false:



if (allowable) //Allowable by reading order
{
  if (G1.readingDirection != G2.readingDirection
   OR |G1.readingOrientation - G2.readingOrientation| > threshold)
  {
    return false; //not allowable
  }

  if (hasRelationSuccessor) //<-
    swap G1 and G2;

  orientation = G1.readingOrientation;

  if (orientation != 0)
  {
    copy1 = G1.Copy();
    copy2 = G1.Copy();

    rotate both polygons by [orientation] degrees around the origin;

    BoundingBox1 = copy1.BoundingBox;
    BoundingBox2 = copy2.BoundingBox;
  }
  else
  {
    BoundingBox1 = G1.BoundingBox;
    BoundingBox2 = G2.BoundingBox;
  }

  direction = G1.readingDirection;
  if (direction is left-to-right OR right-to-left)
  {
    if (BoundingBox2.top < BoundingBox1.bottom - allowance)
      allowable = false;
  }
  else if (direction is top-to-bottom OR bottom-to-top)
  {
    if (BoundingBox2.left < BoundingBox1.right - allowance)
      allowable = false;
  }
}
```

```
  return allowable;
}
```

Extended Check for Allowable Merges

If more than two regions are involved in a merge, transitive allowable relations have to be evaluated. That means if merges between regions A and B as well as between regions B and C are allowable, then the merge between regions A and C is also allowable. Therefore in a second pass all such chains of allowable merges are identified and adjusted. This is realised by following the reading order starting from a ground truth region involved in a merge (in both directions).

Algorithm for Allowable Split

Let's say a ground-truth region is split into n segmentation result regions S_i. The split is allowable if a merge of each region pair (S_i, S_j), $i \neq j$ is allowable.

Segmentation result regions S1 and S2. Reading orientation and direction are determined from the split ground-truth region.

```
CheckIfMergeAllowable(S1, S2, orientation, readingDirection)
{
  if (S1 OR S2 not a text region)
    return false;

  allowable = true;

  if (orientation != 0)
  {
    copy1 = S1.Copy();
    copy2 = S1.Copy();

    rotate both polygons by [orientation] degrees around the origin;

    BoundingBox1 = copy1.BoundingBox;
    BoundingBox2 = copy2.BoundingBox;
  }
  else
  {
    BoundingBox1 = S1.BoundingBox;
    BoundingBox2 = S2.BoundingBox;
  }


  if (readingDirection is left-to-right OR right-to-left)
  {
    if (BoundingBox1 overlaps BoundingBox2 vertically (10px allowance))
      allowable = false;
  }
  else if (readingDirection is top-to-bottom OR bottom-to-top)
  {
    if (BoundingBox1 overlaps BoundingBox2 horizontally (10px allowance))
      allowable = false;
```

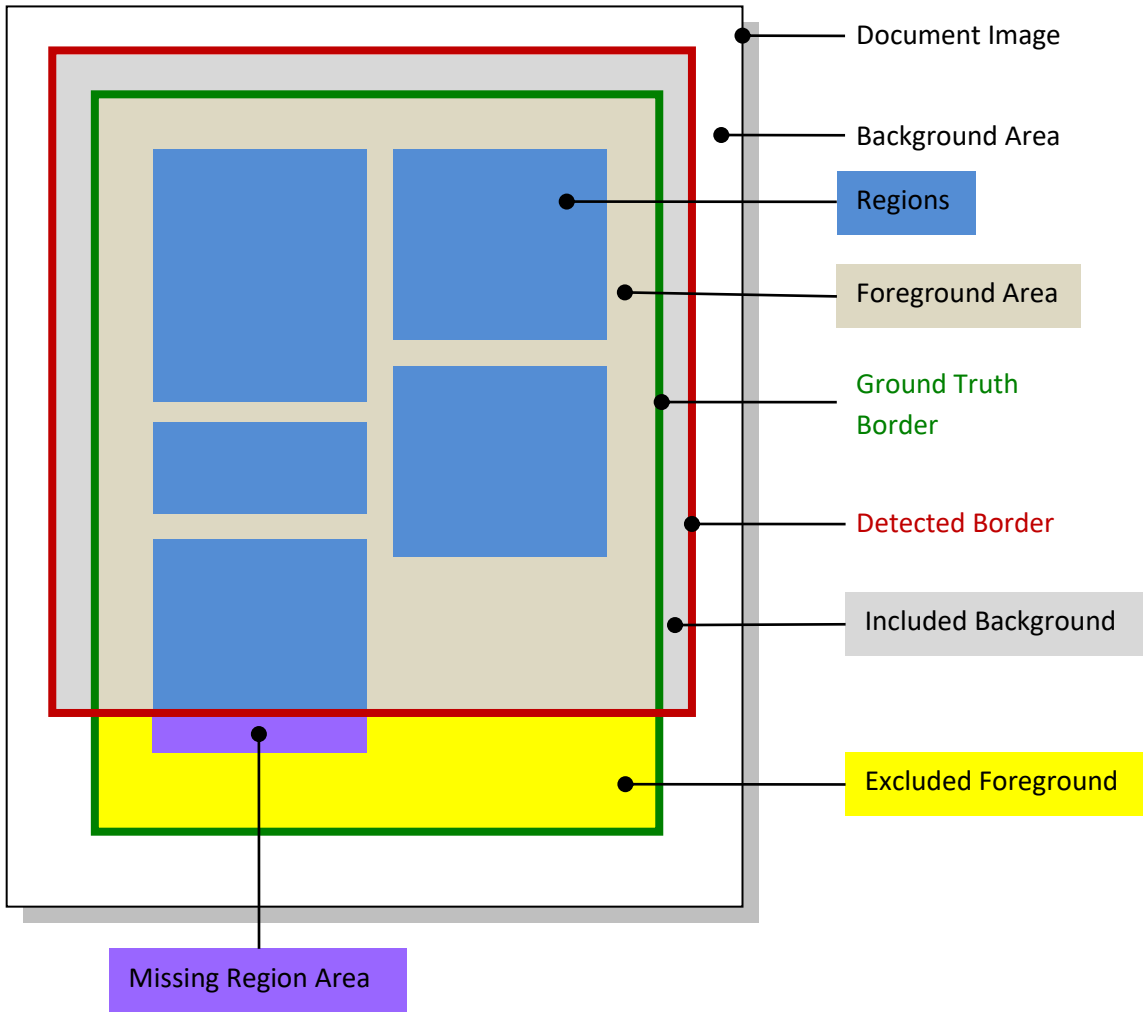
```
  }
  return allowable;
}
```

Border Evaluation

For border evaluation three different types of errors can be identified:

- The border falsely includes parts of the background area (INCL)
- The border falsely excludes parts of the foreground area (EXCL)
- The border excludes parts of regions (missing region area) (MISS)

Example:



If GT is the area defined by the ground truth border, GTR_i the area defined by a ground truth region and SEG the area defined by the automatically detected border, the error areas are calculated as follows:

$$INCL = SEG \setminus (GT \cap SEG)$$

$MISS_i = GTR_i - (GTR_i \cap SEG)$ (one region)

$MISS = \bigcup(MISS_i)$ (all regions)

$EXCL = GT \setminus (GT \cap SEG) \setminus MISS$

Based on the error areas, weighted errors are calculated:

$ERR_{INCL} = w_{INCL} * INCL$

$ERR_{EXCL} = w_{EXCL} * EXCL$

$ERR_{MISS} = \sum(w_{MISS,RTi} * MISS_i)$ (where $w_{MISS,RTi}$ is the weight corresponding to the type of region i)

Then, three success rates are calculated:

$S_{INCL} = f_{INCL}(ERR_{INCL})$

$S_{EXCL} = f_{EXCL}(ERR_{EXCL})$

$S_{MISS} = f_{MISS}(ERR_{MISS})$

The success rate functions are the nonlinear functions discussed in 'Evaluation Statistics and Error Metrics'. The 50% points are set according to following table:

Type	**50% Point**
INCL	(IMG - GT) / 2
EXCL	GT / 2
MISS	$\sum(GTR_i) / 2$

Where IMG is the area of the whole document image.

For the overall success, another function f_{IMG} with 50% point at (IMG/2) is used. The formula is:

$S_{OVERALL} = 3 / (1/f_{IMG}(ERR_{INCL}) + 1/f_{IMG}(ERR_{EXCL}) + 1/f_{IMG}(ERR_{MISS}))$ (harmonic mean)