A Generalization of Otsu's Method and Minimum Error Thresholding Supplement

Jonathan T. Barron

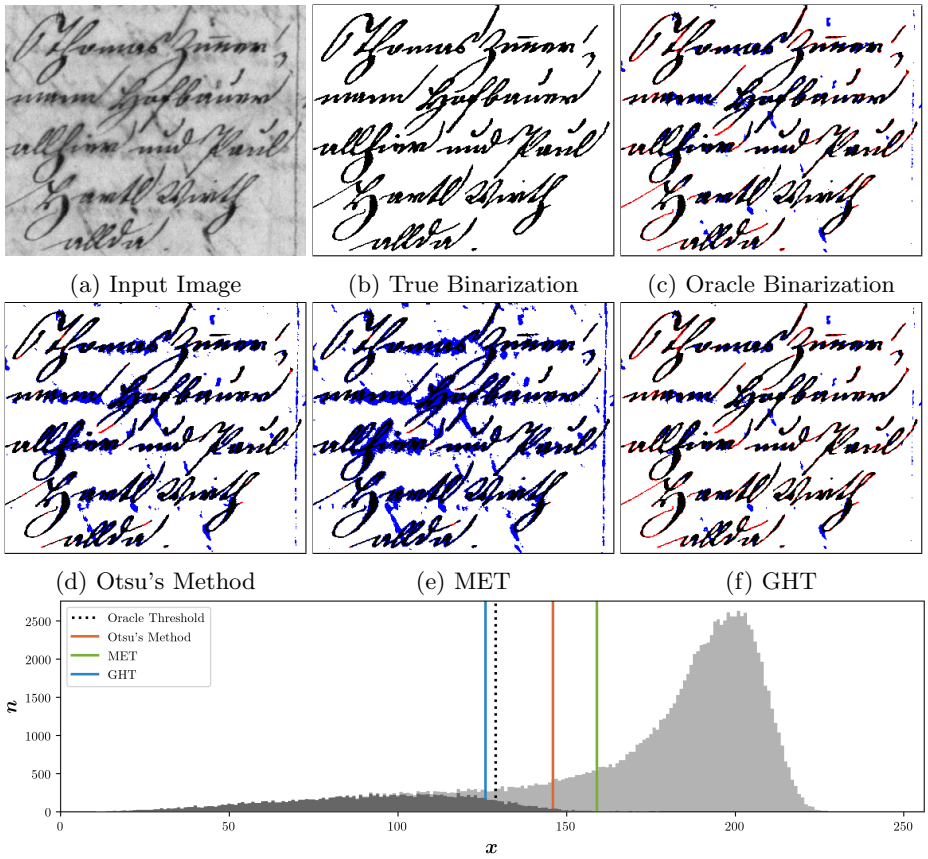
Google Research
barron@google.com

In Figures 1-3 we visualize the output of GHT compared with the algorithms that it generalizes on (test set) images taken from the 2016 Handwritten Document Image Binarization Contest (H-DIBCO) challenge [2].

In Algorithm 1 we present a reference implementation of GHT, as well as reference implementations of the other algorithms that were demonstrated to be special cases of GHT. In Algorithm 2 we present an additional equivalent implementation of GHT that uses an explicit for-loop over splits of the histogram instead of the cumulative sum approach used by the paper. This additional implementation is intended to allow for easier comparisons with similar implementations of MET or Otsu's method, and to allow existing implementations of MET or Otsu's method to be easily generalized into implementations of GHT. We also present a third reference implementation: Algorithm 3 implements GHT in terms of the actual underlying expected complete log-likelihood maximization that reduces to Algorithm 1. This implementation is highly inefficient, but can be used to verify the correctness of GHT in terms of its Bayesian motivation, and may be useful in deriving further probabilistic extensions.

References

1. Dillon, J.V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., Saurous, R.A.: Tensorflow distributions. arXiv preprint arXiv:1711.10604 (2017)
2. Pratikakis, I., Zagoris, K., Barlas, G., Gatos, B.: ICFHR 2016 handwritten document image binarization contest (H-DIBCO 2016). ICFHR (2016)

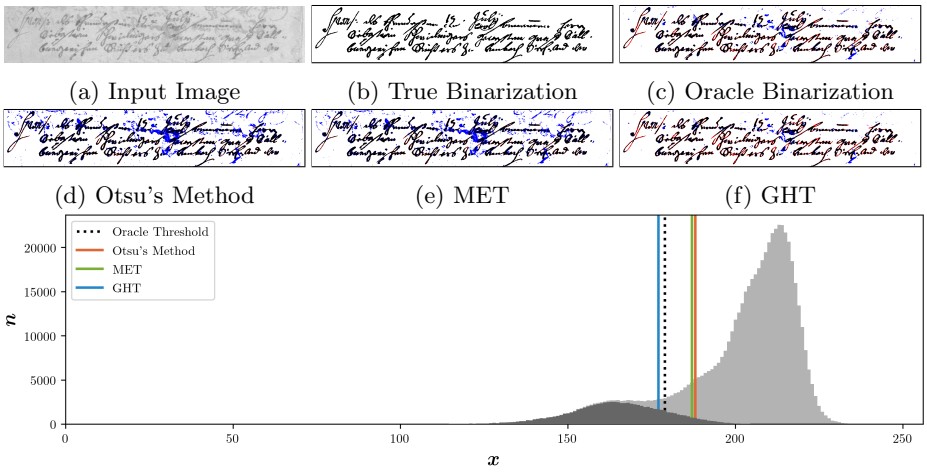


(a) Input Image (b) True Binarization (c) Oracle Binarization

(d) Otsu's Method (e) MET (f) GHT

(g) Histogram and thresholds

Fig. 1: A visualization of our results on an image from the 2016 Handwritten Document Image Binarization Contest (H-DIBCO) challenge [2]. Given (a) the input image we show (b) the ground-truth binarization provided by the dataset, as well as (c) the binarization according to the lowest error (oracle) global threshold, which represents an upper bound on the performance of any global thresholding-based binarization algorithm. The results of Otsu's method, MET, and our GHT are shown in (d-f), where black pixels indicate correct binarization, red pixels indicate false negatives, and blue pixels indicate false positives. In (g) we also show a stacked histogram of input pixel intensities (where masked pixels are rendered in a darker color) alongside the output threshold of each algorithm and the oracle global threshold.

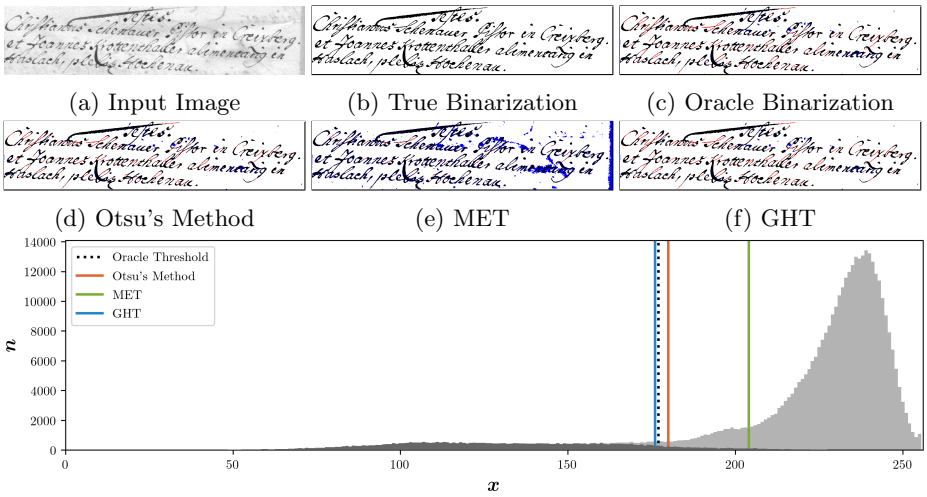


(a) Input Image (b) True Binarization (c) Oracle Binarization

(d) Otsu's Method (e) MET (f) GHT

(g) Histogram and thresholds

Fig. 2: A visualization of additional results, shown in the same format as Figure 1.



(a) Input Image (b) True Binarization (c) Oracle Binarization

(d) Otsu's Method (e) MET (f) GHT

(g) Histogram and thresholds

Fig. 3: A visualization of additional results, shown in the same format as Figure 1.

```
import numpy as np

csum = lambda z: np.cumsum(z)[:-1]
dsum = lambda z: np.cumsum(z[::-1])[-2::-1]
argmax = lambda x, f: np.mean(x[:-1][f == np.max(f)])  # Use the mean for ties.
clip = lambda z: np.maximum(1e-30, z)

def preliminaries(n, x):
  """Some math that is shared across each algorithm."""
  assert np.all(n >= 0)
  x = np.arange(len(n), dtype=n.dtype) if x is None else x
  assert np.all(x[1:] >= x[:-1])
  w0 = clip(csum(n))
  w1 = clip(dsum(n))
  p0 = w0 / (w0 + w1)
  p1 = w1 / (w0 + w1)
  mu0 = csum(n * x) / w0
  mu1 = dsum(n * x) / w1
  d0 = csum(n * x**2) - w0 * mu0**2
  d1 = dsum(n * x**2) - w1 * mu1**2
  return x, w0, w1, p0, p1, mu0, mu1, d0, d1

def Otsu(n, x=None):
  """Otsu's method."""
  x, w0, w1, _, _, mu0, mu1, _, _ = preliminaries(n, x)
  o = w0 * w1 * (mu0 - mu1)**2
  return argmax(x, o), o

def Otsu_equivalent(n, x=None):
  """Equivalent to Otsu's method."""
  x, _, _, _, _, _, _, d0, d1 = preliminaries(n, x)
  o = np.sum(n) * np.sum(n * x**2) - np.sum(n * x)**2 - np.sum(n) * (d0 + d1)
  return argmax(x, o), o

def MET(n, x=None):
  """Minimum Error Thresholding."""
  x, w0, w1, _, _, _, _, d0, d1 = preliminaries(n, x)
  ell = (1 + w0 * np.log(clip(d0 / w0)) + w1 * np.log(clip(d1 / w1))
      - 2 * (w0 * np.log(clip(w0))      + w1 * np.log(clip(w1))))
  return argmax(x, -ell), ell  # argmin()

def wprctile(n, x=None, omega=0.5):
  """Weighted percentile, with weighted median as default."""
  assert omega >= 0 and omega <= 1
  x, _, _, p0, p1, _, _, _, _ = preliminaries(n, x)
  h = -omega * np.log(clip(p0)) - (1. - omega) * np.log(clip(p1))
  return argmax(x, -h), h  # argmin()

def GHT(n, x=None, nu=0, tau=0, kappa=0, omega=0.5):
  """Our generalization of the above algorithms."""
  assert nu >= 0
  assert tau >= 0
  assert kappa >= 0
  assert omega >= 0 and omega <= 1
  x, w0, w1, p0, p1, _, _, d0, d1 = preliminaries(n, x)
  v0 = clip((p0 * nu * tau**2 + d0) / (p0 * nu + w0))
  v1 = clip((p1 * nu * tau**2 + d1) / (p1 * nu + w1))
  f0 = -d0 / v0 - w0 * np.log(v0) + 2 * (w0 + kappa *      omega)  * np.log(w0)
  f1 = -d1 / v1 - w1 * np.log(v1) + 2 * (w1 + kappa * (1 - omega)) * np.log(w1)
  return argmax(x, f0 + f1), f0 + f1
```

Algorithm 1: Reference Python 3 code for our GHT algorithm and the baseline algorithms it generalizes: MET, Otsu's method (here implemented in two equivalent ways), and weighted percentile.

```
import numpy as np

clip = lambda z: np.maximum(1e-30, z)

def GHT_forloop(n, x=None, nu=0, tau=0, kappa=0, omega=0.5):
  """An implementation of GHT() written using for loops."""
  assert np.all(n >= 0)
  x = np.arange(len(n), dtype=n.dtype) if x is None else x
  assert np.all(x[1:] >= x[:-1])
  assert nu >= 0
  assert tau >= 0
  assert kappa >= 0
  assert omega >= 0 and omega <= 1

  n_sum = np.sum(n)
  nx_sum = np.sum(n * x)
  nxx_sum = np.sum(n * x**2)

  max_score, n_c, nx_c, nxx_c = -np.inf, 0, 0, 0
  for i in range(len(n) - 1):
    n_c += n[i]
    nx_c += n[i] * x[i]
    nxx_c += n[i] * x[i]**2
    w0 = clip(n_c)
    w1 = clip(n_sum - n_c)
    p0 = w0 / n_sum
    p1 = w1 / n_sum
    d0 = np.maximum(0, nxx_c - nx_c**2 / w0)
    d1 = np.maximum(0, (nxx_sum - nxx_c) - (nx_sum - nx_c)**2 / w1)
    v0 = clip((p0 * nu * tau**2 + d0) / (p0 * nu + w0))
    v1 = clip((p1 * nu * tau**2 + d1) / (p1 * nu + w1))
    f0 = -d0 / v0 - w0 * np.log(v0) + 2 * (w0 + kappa *      omega)  * np.log(w0)
    f1 = -d1 / v1 - w1 * np.log(v1) + 2 * (w1 + kappa * (1 - omega)) * np.log(w1)
    score = f0 + f1

    # Argmax where the mean() is used for ties.
    if score > max_score:
      max_score, t_numer, t_denom = score, 0, 0
    if score == max_score:
      t_numer += x[i]
      t_denom += 1
  return t_numer / t_denom
```

Algorithm 2: Reference Python 3 code for our algorithm GHT written using only for-loops. This is likely less efficient than the implementation provided in Algorithm 1, and is only provided to demonstrate how this algorithm can be implemented in a single sweep over the histogram (ignoring the calls to `np.sum()`), and to ease integration into existing implementations of Otsu's method or MET.

```
import numpy as np
from tensorflow_probability import distributions as tfd

def sichi2_var(n, resid, nu, tau):
  """Posterior estimate of variance for a scaled inverse chi-squared."""
  return (nu * tau**2 + np.sum(n * resid**2)) / (nu + np.sum(n))

def GHT_prob(n, x=None, nu=0, tau=0, kappa=0, omega=0.5):
  """An implementation of GHT() using probability distributions."""
  assert np.all(n >= 0)
  x = np.arange(len(n), dtype=n.dtype) if x is None else x
  assert np.all(x[1:] >= x[:-1])
  assert nu >= 0
  assert tau >= 0
  assert kappa >= 0
  assert omega >= 0 and omega <= 1

  n_sum = np.sum(n)
  lls = np.zeros(len(n) - 1)
  for i in range(len(lls)):
    n0, n1 = n[:(i+1)], n[(i+1):]
    x0, x1 = x[:(i+1)], x[(i+1):]
    w0 = clip(np.sum(n0))
    w1 = clip(np.sum(n1))
    p0 = clip(w0 / n_sum)
    p1 = clip(w1 / n_sum)
    mu0 = np.sum(n0 * x0) / w0
    mu1 = np.sum(n1 * x1) / w1
    var0 = sichi2_var(n0, x0 - mu0, p0 * nu, tau)
    var1 = sichi2_var(n1, x1 - mu1, p1 * nu, tau)
    lls[i] = ((np.sum(n0 * (np.log(p0) + tfd.Normal(mu0, np.sqrt(var0)).log_prob(x0)))
             + np.sum(n1 * (np.log(p1) + tfd.Normal(mu1, np.sqrt(var1)).log_prob(x1))))
             + tfd.Beta(kappa * omega + 1, kappa * (1 - omega) + 1).log_prob(np.minimum(p0, 1-1e-15)))
  return np.mean(x[:-1][lls == np.max(lls)]), lls
```

Algorithm 3: Reference Python 3 and Tensorflow Probability [1] code for GHT written in terms of probabilities as described in the paper. This is a highly inefficient way to implement this algorithm, but is provided to allow the reader to verify the equivalence between the efficient GHT implementation presented earlier and this Bayesian formulation, and to enable future work that extends this approach.