Minimum Edit Distance on a Probabilistic String

Presented to the S. Daniel Abraham Honors Program

in Partial Fulfillment of the

Requirements for Completion of the Program

Stern College for Women

Yeshiva University

May 23, 2023

Dahlia Schwartz

Mentor: Professor Alan Broder, Computer Science

Abstract:

Optical character recognition, or OCR, is a method used to convert images of typed or handwritten text into machine-encoded text. Oftentimes, text can be illegible or worn out, and therefore ambiguous. In these situations, OCR models can output a probabilistic string, or sequence of characters, with a ranking of several less likely options as well. In order to quantify how dissimilar the output string is from another string, Levenshtein distance, or other edit distance algorithms are used. These algorithms count the number of operations required to convert one string into another. The possible operations that can be performed in most edit distance algorithms consist of inserting a character, deleting a character, and replacing a character. The smaller the Levenshtein distance between two strings, the more similar the strings are to one another. There are various edit distance algorithms, each with their own run-time, efficiency, and readability, however, most of these algorithms do not take probabilistic strings into account. This paper's contribution is to survey prominent methods of calculating the minimum edit distance of two strings and to evaluate how each method takes run-time, efficiency, and the ability to work with probabilistic strings into account. This will help automate the process to find missing and extra letters in scrolls that were copied from the Masoretic Text, which can help Biblical researchers.

Introduction:

Optical Character Recognition

Optical character recognition is useful for extracting information from documents such as medical records, legal texts, and historical documents. It can be used to process hand-written forms, convert a printed book into an electronic book, or read license plates. OCR, however, can cause some errors that cause the extracted information to be ambiguous (Hládek et al., 2017)[1]. OCR works by scanning a document and converting it into binary data. The software can then scan the image to differentiate between the text and the background. Before the actual process begins, there is some preprocessing that needs to take place. First, the OCR model needs to clean the image to remove theoretical errors in the reading process. Some steps it takes to clean the image include aligning the text if it was originally unaligned, removing any specks that should not be read as part of the text, and smoothing out the edges of the images (1978)[2].

Pattern matching and feature extraction are two types of algorithms that OCR software might use for text recognition. In pattern matching, a glyph, or character image, is isolated and compared to another glyph. In feature extraction, the glyphs are decomposed into features, such as lines or loops, and these features are used to find the best match with other glyphs (ibid).

There are different types of OCR technologies that have different uses. The simple OCR engine stores many different fonts and image patterns as templates and uses pattern-matching algorithms to compare text images, one character at a time. Optical word recognition is when the system matches the text one word at a time. Intelligent character recognition software is similar to the way in which humans read text. The machines are trained by a machine learning software, which helps preprocess the image many times so that it can recognize different image attributes.

---

[1] Hládek, D., Staš, J., Ondáš, S. *et al.* Learning string distance with smoothing for OCR spelling correction. *Multimed Tools Appl* 76, 24549–24567 (2017). https://doi.org/10.1007/s11042-016-4185-5

[2] "What Is OCR." *Amazon*, 1978, aws.amazon.com/what-is/ocr/.

Intelligent word recognition is similar to intelligent character recognition but processes the text one word at a time (ibid).

The Significance of Analyzing Torah Scrolls

Torah scrolls are written by professional scribes, or *sofrim* in Hebrew. The Torah is considered a blueprint by which all Jews are supposed to live and consists of a plethora of commandments that all Jews must observe. God expects Jews to abide by the standards of the Torah. In order for the Torah scrolls to be considered *kosher*, they have to pass an abundance of requirements. Some examples of these requirements are that Torah scrolls must be written on the hide of a *kosher* animal, using the dominant hand of the *sofer*, in black ink. There are requirements regarding the amount of distance required between letters in the *sefer Torah* and rules about how much space required between words. These requirements convey the significance of Torah scrolls and how much thought, effort, and time goes into each one. Because of these strict guidelines, it can be quite informative to analyze and compare various *sifrei Torah*. Comparing and contrasting one scroll with another can help determine when they were written, what traditions their respective *sofrim* held, and the location of where the scrolls were written.

Rabbi Ephraim Kaspi's mission using OCR and minimum edit distance

Rabbi Ephraim Caspi, an Israeli scholar, researches *sifrei Torah*, or *Torah* scrolls, from various time periods in order to analyze the change in tradition and style. The styles of letters and words in *sifrei Torah* over multiple generations have the ability to tell a story about the way in

which tradition may change or get lost over time. The Masoretic Text is the authoritative Hebrew text of *Tanach*, or the Hebrew Bible. As part of Caspi's mission to compare *sifrei Torah*, it is important to recognize whether any letters are missing, added, or replaced from the Masoretic Text. In order to calculate whether there is a change in character from the Masoretic Text to the current *sefer Torah* being analyzed, it would be helpful to use OCR to convert the current text into machine-readable text and then to use a minimum edit distance algorithm to compare the machine-readable text to the Masoretic Text.

Why minimum edit distance is important

Minimum edit distance is important in many sects of computer science, such as natural language processing. When implementing an automatic spell check, minimum edit distance would be used to compare the misspelled word that was typed to a word in the dictionary with the smallest distance. When analyzing various *sifrei Torah* and whether they have any *chaseirot*, missing characters, or *yeteirot*, additional characters, minimum edit distance is especially important. Minimum edit distance can be used to compare the text from a *sefer Torah* that was copied from the Masoretic Text with the Masoretic Text itself. If there is an extra *vav* or a missing *yud* in the *sefer Torah* being analyzed, the minimum edit distance algorithm would be able to detect the difference by calculating the minimum number of operations needed to convert the text from the *sefer Torah* into the Masoretic Text.

What will be addressed in this paper

This paper will begin with an outline of prior research on this topic. There have been other attempts at converting Hebrew text to machine-encoded text, but none of them are perfect for the goal of my project. It will define the concept of running a minimum edit distance algorithm on a probabilistic string and why it is inherently different from running a minimum edit distance on a certain string. It will explain why the likelihood of each character should contribute to the cost of the edit operation. Next, this paper discusses various edit distance algorithms. It outlines Levenshtein distance and its applications, recursive and iterative implementations of the algorithm, and evaluates efficiency. It then covers weighted Levenshtein distance, which is the implementation that is most relevant to probabilistic strings because the costs of different edit operations can be specified as input parameters. It then bridges the gap between weighted Levenshtein distance and edit distance with probabilistic strings.

1. Prior Research

1.1 Tesseract

Tesseract is an OCR engine that was originally created in the 1980s by Hewlett-Packard as proprietary software but then became open-source in 2005 (2023)[3]. Tesseract was created under the Apache 2.0 license, which is a free software license that has minimal restrictions on how the software can be used. Because Tesseract is under the Apache license, users can use the

[3] Tesseract-Ocr. “Tesseract-OCR/Tesseract: Tesseract Open Source OCR Engine (Main Repository).” *GitHub*, github.com/tesseract-ocr/tesseract.

software for any purpose, modify it, distribute it, and not have to pay royalties (2023)[4]. Tesseract code can be viewed on GitHub, which includes open issues for Tesseract, the latest version of their software, and planning documentation. Tesseract was originally created to be used from the command-line interface, but third parties have since created GUIs, or graphical user interfaces, to go along with it. It supports a variety of languages, and programmers can use an API to extract printed text from images (2006)[5].

1.2 Using Tesseract with Hebrew Text

In December 2012, Adi Oz and Vered Shani worked to develop a project called 'Hebrew OCR with *Nikud'*. Their goal was to write a program that takes a Hebrew text file without *nikud*, or signs used to represent vowels in Hebrew, as input, and return a Hebrew text file with *nikud*. They felt that the use of *nikud* was decreasing and they wanted to help preserve the Hebrew language by working to better represent its text. They used Tesseract and found that without *nikud*, it was pretty accurate. With *nikud*, however, they found that Tesseract outputted a text with many typos. Attached below are pictures of a text using Tesseract with and without *nikud* (Oz Et al., 2012)[6].

[4] "APACHE LICENSE, VERSION 2.0." *Welcome to the Apache Software Foundation!*, www.apache.org/licenses/LICENSE-2.0.
[5] "Tesseract User Manual." *Tessdoc*, tesseract-ocr.github.io/tessdoc/.
[6] Oz, Adi, and Vered Shani. *Hebrew OCR with Nikud*, www.cs.bgu.ac.il/~elhadad/hocr/.

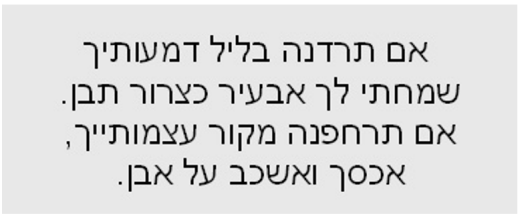


Figure 1. A Hebrew text image without *nikud* and Tesseract's output; typos highlighted in red (ibid)

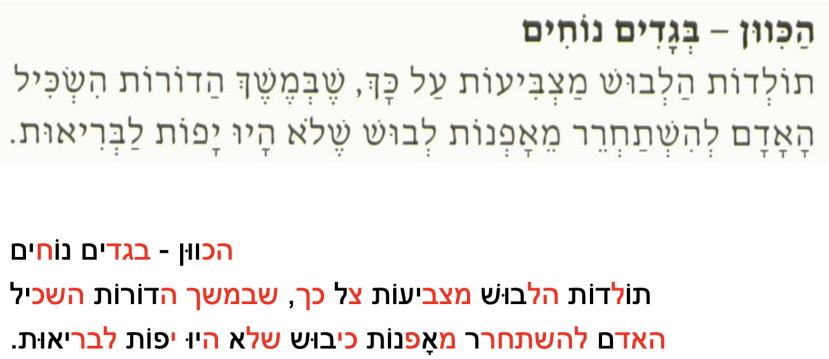


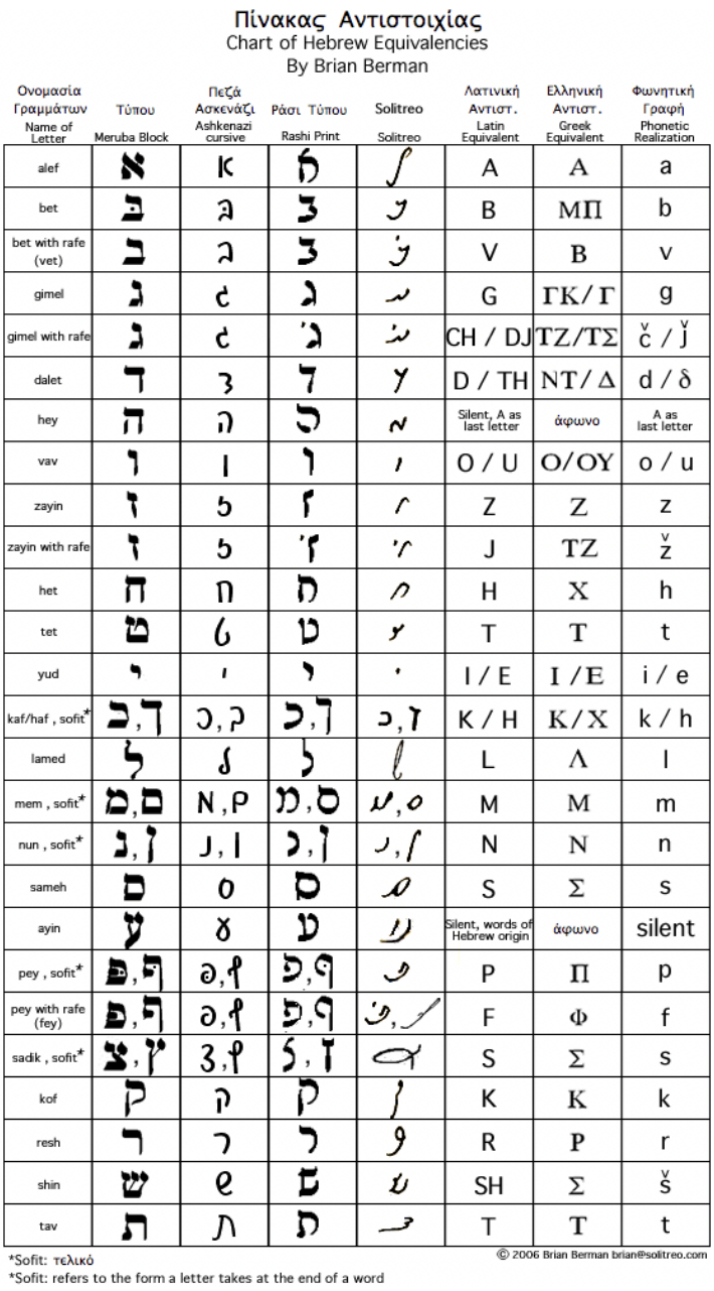
Figure 2. A Hebrew text image with *nikud* and Tesseract's output; typos highlighted in red (ibid)

Although they trained Tesseract to read Hebrew text with *nikud* more accurately, they were working with clear text that was not damaged. This model will not work for ancient Torah Scrolls that have been damaged or blurred over time.

1.3 Technion - OCR for Old Torah Manuscripts

In 2020, under the supervision of Ori Bryt, Tal Stolovich and Ohad Kimelfeld worked on a project that involved OCR and old Torah manuscripts. More specifically, they worked with Solitreo font, which is a cursive font in the Hebrew alphabet, and was common to Sephardi Jews prior to Israel's establishment (2020)[7]. Attached below is a comparison of Solitreo font with other forms of Hebrew text.

[7] "Optical Character Recognition (OCR) for Old Torah Manuscripts." *Signal and Image Processing Lab.*, sipl.eelabs.technion.ac.il/projects/optical-character-recognition-ocr-for-old-torah-manuscripts/.



Πίνακας Αντιστοιχίας
Chart of Hebrew Equivalencies
By Brian Berman

Ονομασία Γραμμάτων Name of Letter	Τύπου Meruba Block	Πεζά Ασκενάζι Ashkenazi cursive	Ράσι Τύπου Rashi Print	Solitreo Solitreo	Λατινική Αντιστ. Latin Equivalent	Ελληνική Αντιστ. Greek Equivalent	Φωνητική Γραφή Phonetic Realization
alef	א	א	א	𐤀	A	Α	a
bet	בּ	בּ	בּ	𐤁	B	ΜΠ	b
bet with rafe (vet)	ב	ב	ב	𐤁	V	Β	v
gimel	ג	ג	ג	𐤂	G	ΓΚ/Γ	g
gimel with rafe	ג	ג	ג׳	𐤂	CH / DJ	ΤΖ/ΤΣ	č / ǰ
dalet	ד	ד	ד	𐤃	D / TH	ΝΤ/Δ	d / δ
hey	ה	ה	ה	𐤄	Silent, A as last letter	άφωνο	A as last letter
vav	ו	ו	ו	𐤅	O / U	Ο/ΟΥ	o / u
zayin	ז	ז	ז	𐤆	Z	Ζ	z
zayin with rafe	ז	ז	ז׳	𐤆	J	ΤΖ	ž
het	ח	ח	ח	𐤇	H	Χ	h
tet	ט	ט	ט	𐤈	T	Τ	t
yud	י	י	י	𐤉	I / E	Ι / Ε	i / e
kaf/haf , sofit*	כ,ך	כ,ך	כ,ך	𐤊	K / H	Κ/Χ	k / h
lamed	ל	ל	ל	𐤋	L	Λ	l
mem , sofit*	מ,ם	מ,ם	מ,ם	𐤌	M	Μ	m
nun , sofit*	נ,ן	נ,ן	נ,ן	𐤍	N	Ν	n
sameh	ס	ס	ס	𐤎	S	Σ	s
ayin	ע	ע	ע	𐤏	Silent, words of Hebrew origin	άφωνο	silent
pey , sofit*	פ,ף	פ,ף	פ,ף	𐤐	P	Π	p
pey with rafe (fey)	פ,ף	פ,ף	פ,ף	𐤐	F	Φ	f
sadik , sofit*	צ,ץ	צ,ץ	צ,ץ	𐤑	S	Σ	s
kof	ק	ק	ק	𐤒	K	Κ	k
resh	ר	ר	ר	𐤓	R	Ρ	r
shin	ש	ש	ש	𐤔	SH	Σ	š
tav	ת	ת	ת	𐤕	T	Τ	t

*Sofit: τελικό
*Sofit: refers to the form a letter takes at the end of a word

Figure 3. Solitreo font, shown in the fifth column, in comparison to other forms of Hebrew text[8]

[8] *Solitreo.Com*, solitreo.com/chart.

The goal of this project was to automate the conversion of Solitreo manuscripts to a digital Modern-Hebrew text document. Their model had high recall and accuracy but had trouble differentiating between noise, punctuation, and parts of letters (2020)[9]. In other words, the percentage of words in the original text correctly found by their OCR model was high, but their model was not good at ignoring unwanted behaviors within the data. This model would have been useful to convert a Biblical text to machine-encoded text, however, it can only be used on Solitreo font, which is different from that of *sifrei Torah*.

1.4 Prior Research at Yeshiva University

Alyssa Aharon, class of 2022 at Stern College for Women, wrote a thesis on identifying anomalous letters in Torah scrolls. Instead of manually sifting through many Torah scrolls to locate any *otiyot meshunot*, anomalous letters, the goal was to build a prototype for a deep learning model that would automatically locate *otiyot meshunot* without manual labor. The model was trained using various Torah scrolls and OCR, which was used to translate written text into its corresponding digital letters. Images of Torah scrolls were obtained from the Museum of the Bible in Washington, DC, and used those images as data in her model. High-quality images were used from the museum in order to build a classifier that would label each written letter with a number that would correspond to its position in the alphabet. A threshold was then set in order to determine whether a letter fails to fall under any category with enough confidence, which would determine that letter as an anomaly (Aharon, 2022)[10].

---

[9] "Optical Character Recognition (OCR) for Old Torah Manuscripts." *Signal and Image Processing Lab.*, sipl.eelabs.technion.ac.il/projects/optical-character-recognition-ocr-for-old-torah-manuscripts/.
[10] Aharon, Alyssa. *Deep Torah Learning: A Deep Learning Approach to Identifying Anomalous ...*, repository.yu.edu/handle/20.500.12202/8249.

In order to accomplish the goal of identifying *otiyot meshunot*, OCR was used to detect outliers that deviated from the trend of regular letters. Relative to the set threshold, when the prediction of a certain letter was uncertain, it would be labeled as an anomalous letter. A new dataset had to be created, as no pre-existing dataset contained precisely what was needed to detect anomalous letters. Pre-existing datasets either consisted of whole words, as opposed to individual letters, or modern Hebrew script, which looks drastically different from the handwritten letters written in Torah scrolls. Images from eight different *sifrei Torah* were used to gather her data. In order to process the images of *sifrei Torah*, the algorithm used OpenCV, an optimized computer vision library. Once the images were processed, the letters needed to be labeled so that they could be identified by the model. A letter value was assigned to each image of a letter so that it could be identified properly. Because many dots of ink were falsely detected as letters when the Torah scroll images were segmented, each image had to be manually sifted through in order to determine whether it was falsely detected as a letter or whether it was a true letter, and if so, which one. As Aharon noted, it would be more efficient in the future to further limit the threshold for what constitutes a letter by decreasing the range of acceptable letter size (ibid). Had this been done, there would be fewer falsely detected letters. Attached below is an image of the steps that took place during the processing stage. From left to right, this image consists of the original image, the grayscale image, the binarized image, and the image with bounding rectangles.

Figure 4. The four stages necessary for preprocessing and segmenting an image (ibid)

Throughout the processing stage, approximately 2,000 images of letters were used. Because some letters are used more frequently than others, there was a drastic range in the ratio of how many images existed of each letter in the Hebrew alphabet. If more data was used, there would be less room for error, as there would be more images of each letter, including the more rarely used letters. A deep learning model was built using TensorFlow, an open-source library for machine learning, together with Keras, a high-level neural network library that runs on top of TensorFlow. The results of the model showed that it was less certain when trying to classify anomalous letters than it was when trying to classify typical letters. The level of uncertainty was

measured by an activation function. This function would output an activation level that would indicate how likely it is that the current image of a letter matches with a target letter. If the activation level was low, that meant that it was less likely that the image matched the target letter (ibid). A decision was made that it would be better to falsely identify typical letters as anomalous letters, and simply exclude them than to falsely identify anomalous letters as typical letters, and miss out on crucial data. An interface was created using Jupyter Notebook that allows for a user to upload an image of a *sefer Torah* and then outputs a class label, along with the likelihood of that letter belonging to that class (ibid). After concluding her thesis about her prototype, Aharon stated,

> In order for this project to be practically implemented on a large scale, several improvements would be necessary. The current product is limited to images of single letters, assuming that a full text is preprocessed and segmented. A more developed product would be able to automatically perform the preprocessing steps on a raw Torah image. Assuming that this is possible, this product would aid the research involving *otiyot meshunot*, leading to greater understanding of origins and traditions of particular Torah scrolls. Additionally, the data is currently limited to data of a very specific source and format, and the model must be trained on a much larger sample in order to generalize to any Torah images. As another possible avenue for further development, this technology can potentially be used to detect errors in a *sefer Torah* by identifying abnormalities in the text. The ultimate goal in this realm would be to contribute an open-source version of a computerized checker for *sifrei Torah* (as well as *mezuzot* and *tefillin,* which are more ubiquitous and abide by similar standards) that can be utilized and understood by the public (ibid).

Although this model has room for improvement, it demonstrates the idea that OCR models have the ability to output probabilistic characters. The activation level represents the likelihood that the character being outputted by the OCR model is accurate. Using the activation level, the model can output a ranking of the most likely characters, which can then be strung together into a probabilistic string.

1.5 OCR With Probabilistic Output

When a text is blurry, unclear, or damaged, OCR outputs generally contain more errors than average. In a study done by students at the University of Wisconsin-Madison, it was found that in order to minimize errors in OCR output, OCR programs, such as Google Books, can output probabilistic models. These probabilistic models can capture alternative strings in the recognition process.[11] Google Books, for example, includes an OCRopus tool that can define a probability distribution over all possible strings that could be shown in a specific image (ibid).

1.6 Connection Between Prior Research and My Own

As noted, many of the prior OCR models for converting Hebrew text images to machine-encoded text have flaws that would inhibit them from aiding in my research. One model is limited to Solitreo font, one model expects images of better quality than that of ancient Torah scrolls, and one model requires too much manual labor. I plan to suggest a minimum edit distance algorithm that can be used on probabilistic strings. After using output from an OCR model and inputting it into my minimum edit distance algorithm on probabilistic strings, it will be possible to identify when a letter in a particular Torah scroll is added, removed, or altered. These discoveries can lead to a more vast understanding of when and where the scroll was written and what traditions were prominent at the time of origin.

[11] Kumar, Arun, and Christopher R´e. *Probabilistic Management of OCR Data Using an RDBMS - Stanford University*, cs.stanford.edu/people/chrismre/papers/HazyOCR_TR.pdf.

2. Existing Minimum Edit Distance Algorithms

2.1 Introduction

Existing minimum edit distance algorithms do not accept probabilistic strings as input. While they can output the minimum distance required to convert an inputted string to a target string, they do not work on strings with uncertain predictions. In order to create an algorithm that would work on probabilistic strings, several changes would have to be made to the existing models.

2.2 Levenshtein Distance

In 1965, Vladimir Levenshtein, a Soviet mathematician, designed a metric for measuring the difference between two strings, called Levenshtein distance. In the naive recursive implementation of edit distance, the Levenshtein distance between two strings, *a* and *b*, is outputted by the following guidelines.

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) & \text{if } a[0] = b[0], \\ 1 + \min \begin{cases} \operatorname{lev}\left(\operatorname{tail}(a), b\right) \\ \operatorname{lev}\left(a, \operatorname{tail}(b)\right) \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) \end{cases} & \text{otherwise} \end{cases}$$

Figure 5. Naive recursive implementation of the minimum edit distance algorithm, where $|a|$ represents the length of string a, and $|b|$ represents the length of string b (2023)[12]

Recursion is a method that helps solve computational problems by depending on solutions to smaller instances of the same problem. In the naive recursive implementation, the Levenshtein distance between "kitten" and "sitting" would evaluate to three because that is the minimum number of edits required in order to convert the former string into the latter. An example of the three edits to convert the string "kitten" into the string "sitting" would be substituting the "k" for an "s", substituting the "i" for an "e", and inserting a "g" at the end of the string (ibid).

The Levenshtein distance has several upper and lower bounds that limit its range. First, it needs to be greater than or equal to the absolute value of the difference between the lengths of the two input strings. Additionally, it must be less than or equal to the length of the longer string. The edit distance can only equal zero if the two strings are exactly equal. Another limitation on the bounds of Levenshtein distance comes from the triangle inequality. The triangle inequality in mathematics states that for any triangle, the sum of the lengths of any two sides must be greater than or equal to the length of the third side. Applying this principle in mathematics to Levenshtein distance would require that the Levenshtein distance between two strings cannot be greater than the sum of the Levenshtein distances between each of those strings and a third string (ibid).

[12] "Levenshtein Distance." *Wikipedia*, 11 May 2023, en.wikipedia.org/wiki/Levenshtein_distance.

2.3 Applications of Levenshtein Distance

Levenshtein distance is useful in processes like spell checkers and systems that correct optical character recognition. It is also helpful for quantifying linguistic distance, which measures how different two languages are from one another. The higher the mutual intelligibility between two languages, the higher the linguistic distance is between them.

2.4 Alternatives to Levenshtein Distance

There are other edit distance metrics that consist of a different set of edit operations than that of the typical Levenshtein distance. One such metric is the Damerau-Levenshtein distance, which includes an additional edit, transposition. Transposition allows two adjacent letters to swap places. Another such metric is the longest common subsequence, or LCS. LCS is similar to the typical Levenshtein distance but does not allow substitution. The Hamming distance only allows substitution, and therefore can only be used on two strings of the same length. The Jaro distance only allows for transposition. In each metric, every edit operation is assigned a cost, which can be infinite, and the edit distance equals the sum of the costs of the minimum edit operations necessary (ibid).

2.5 Recursive, Haskell Implementation

Haskell is a coding language designed for research and teaching purposes. It is statically typed, has type-inference, and lazy evaluation. One computation of edit distance uses Haskell to

write a recursive algorithm that is simple, yet inefficient. This algorithm takes two strings, *s* and *t*, and their lengths, and outputs the Levenshtein distance between them (ibid). The figure below outlines the algorithm.

```haskell
lDistance :: Eq a => [a] -> [a] -> Int
lDistance [] t = length t -- If s is empty, the distance is the number of characters in t
lDistance s [] = length s -- If t is empty, the distance is the number of characters in s
lDistance (a : s') (b : t') =
  if a == b
    then lDistance s' t' -- If the first characters are the same, they can be ignored
    else
      1
        + minimum -- Otherwise try all three possible actions and select the best one
          [ lDistance (a : s') t', -- Character is inserted (b inserted)
            lDistance s' (b : t'), -- Character is deleted  (a deleted)
            lDistance s' t' -- Character is replaced (a replaced with b)
          ]
```

Figure 6. Haskell, recursive implementation of edit distance (ibid)

In this algorithm, the Levenshtein distance of the same substrings is calculated many times, causing it to be inefficient. In order to be more efficient, the Levenshtein distance should not be calculated more than once. Instead of calculating the same distances multiple times, the distances of all possible suffixes can be stored in an array, a data structure that contains elements (ibid). It would be quick to look up the distance between any possible suffixes of the two strings simply by indexing into the 2D array.

2.6 Wagner-Fischer Algorithm

The Wagner-Fischer Algorithm for edit distance is iterative, as opposed to recursive, which means that it consists of a sequence of instructions. It also uses dynamic programming, which helps simplify a problem by breaking it down into smaller steps. This algorithm calculates edit distance by creating a matrix that holds the edit distances between all prefixes of the first and second strings. The last value to be computed is the minimum edit distance which can always be found in the bottom rightmost position of the matrix (ibid). The figure below shows the code for the algorithm.

```
function Distance(char s[1..m], char t[1..n]):
  // for all i and j, d[i,j] will hold the distance between
  // the first i characters of s and the first j characters of t
  // note that d has (m+1)*(n+1) values
  declare int d[0..m, 0..n]

  set each element in d to zero

  // source prefixes can be transformed into empty string by
  // dropping all characters
  for i from 1 to m:
      d[i, 0] := i

  // target prefixes can be reached from empty source prefix
  // by inserting every character
  for j from 1 to n:
      d[0, j] := j

  for j from 1 to n:
      for i from 1 to m:
          if s[i] = t[j]:
            substitutionCost := 0
          else:
            substitutionCost := 1

          d[i, j] := minimum(d[i-1, j] + 1,                   // deletion
                             d[i, j-1] + 1,                   // insertion
                             d[i-1, j-1] + substitutionCost)  // substitution

  return d[m, n]
```

Figure 7. Wagner-Fischer Algorithm for minimum edit distance (ibid).

An example of the matrix that would be created from the Wagner-Fischer algorithm is shown below, where the dotted underline indicates that an edit operation was performed.

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Figure 8. Matrix created by Wagner-Fischer algorithm in order to compute the minimum edit distance between the two strings, “kitten” and “sitting” (ibid)

This figure shows that the minimum edit distance between “kitten” and “sitting” is three, as that is the number located and the bottom rightmost position in the matrix.

2.7 Weighted Levenshtein Distance

Weighted Levenshtein Distance is the concept that not all edit operations should be of equal value. Perhaps substituting the letter "O" for the number zero, which looks very similar, should cost less than substituting two letters that look completely different. Maybe substituting two letters that are located near each other on the QWERTY keyboard should cost less than substituting two letters that are located further apart on the QWERTY keyboard. Because of this concept, David Su, under an MIT license, created a library that allows users to specify their desired weights for each edit operation. The bulk of the algorithm is written in Cython, a programming language that combines Python and C, which allows for the algorithm to run at an optimized speed (Su, 2016)[13].

Included in this library are three functions. One function is called weighted_levenshtein.levenshtein and calculates the Levenshtein distance between two strings. This function takes the first string being compared, the second string being compared, the cost of inserting a specified character into one string, the cost of deleting a specified character from one string, and the cost of substituting a specified character from one string as input parameters. If any of the costs are not specified, it defaults to one (ibid). Another method included in this library is weighted_levenshtein.optimal_string_alignment, which calculates the optimal string alignment between two strings. This method includes the same input parameters as the weighted_levenshtein.levenshtein method, but also includes an optional parameter for the cost of transposing a specified character with another adjacent character, with a default value of one (ibid). The third method included within the weighted Levenshtein library is weighted_levenshtein.demerau_levenshtein. This method calculates the Damerau-Levenshtein distance between two strings and takes the same input parameters as the method that calculates

[13] Su, David. "Welcome to Weighted-Levenshtein's Documentation!¶." *Weighted*, weighted-levenshtein.readthedocs.io/en/master/.

optimal string alignment (ibid). Attached below is a usage example after installing the weighted Levenshtein library and invoking different methods.

```
import numpy as np
from weighted_levenshtein import lev, osa, dam_lev


insert_costs = np.ones(128, dtype=np.float64)  # make an array of all 1's
insert_costs[ord('D')] = 1.5  # make inserting the character 'D' have cost 1.5 (instead of 1)

# you can just specify the insertion costs
# delete_costs and substitute_costs default to 1 for all characters if unspecified
print lev('BANANAS', 'BANDANAS', insert_costs=insert_costs)  # prints '1.5'

delete_costs = np.ones(128, dtype=np.float64)
delete_costs[ord('S')] = 0.5  # make deleting the character 'S' have cost 0.5 (instead of 1)

# or you can specify both insertion and deletion costs (though in this case insertion costs don't matter)
print lev('BANANAS', 'BANANA', insert_costs=insert_costs, delete_costs=delete_costs)  # prints '0.5'


substitute_costs = np.ones((128, 128), dtype=np.float64)  # make a 2D array of 1's
substitute_costs[ord('H'), ord('B')] = 1.25  # make substituting 'H' for 'B' cost 1.25

print lev('HANANA', 'BANANA', substitute_costs=substitute_costs)  # prints '1.25'

# it's not symmetrical! in this case, it is substituting 'B' for 'H'
print lev('BANANA', 'HANANA', substitute_costs=substitute_costs)  # prints '1'

# to make it symmetrical, you need to set both costs in the 2D array
substitute_costs[ord('B'), ord('H')] = 1.25  # make substituting 'B' for 'H' cost 1.25 as well

print lev('BANANA', 'HANANA', substitute_costs=substitute_costs)  # now it prints '1.25'
```

Figure 9. Usage example of weighted Levenshtein library (ibid)

As shown above, this usage example specifies the cost of various methods. For example, it declares that the cost of deleting the character ‘S’ should cost 0.5, instead of the default value of one.

3. Probabilistic Strings

3.1 Bridge Between Probabilistic Strings and Weighted Levenshtein

The premise of weighted Levenshtein can be applied to probabilistic strings. In order to use probabilistic strings, the weighted Levenshtein algorithm would need to consider the costs of the different possible character evaluations. The costs for the different possible character evaluations should inversely correlate with the likelihood that the suggested character is accurate. A machine learning model can return several possible characters with their corresponding activation levels. The activation level represents how likely it is that the suggested character is truly the character that is being read from the text. If the character is more likely to be accurate, for example, if it was physically clear in the Torah scroll and not damaged, it should cost less for the Levenshtein algorithm to use that character than another, less likely character. Because the activation level can be a number up to 100, it makes sense to divide the value by 100 when deciding how much it should cost to use, so that it can represent a percentage. In other words, the cost of using a specific character should evaluate to one minus the activation level divided by 100.

3.2 Representation of the Probabilistic String

The probabilistic string would need to be represented as a list of dictionaries. A dictionary is a data structure that stores data values in key-data pairs and has constant lookup time. Constant lookup time means that an element in the dictionary can be accessed or retrieved

in the same amount of time, regardless of the size of the collection. In other words, the time required to find an element does not increase as the number of elements in the dictionary grows. To implement minimum edit distance on a probabilistic string, it would make sense to use a list of dictionaries, where each element in the list corresponds to a detected character. The key in the element would signify a detected character, and the data would represent the likelihood that the detected character is accurately read, a number between zero and one.

An example of how this list of dictionaries could be represented in the code is outlined by: [{'H':.7, '5':.2, 'N': .1}, {'C':.3, 'E':.2}]. The OCR algorithm would output this list, and that would mean that the first character could be an 'H', with .7 percent accuracy, or a '5', with .2 percent accuracy, or an 'N', with .1 percent accuracy. The second letter could be a 'C', with .3 percent accuracy, or an 'E', with .2 percent accuracy, and so on.

The runtime complexity of this algorithm would be closely related to the runtime complexity of the weighted Levenshtein algorithm because indexing into a list has constant lookup time and indexing into a dictionary inside of a list also has constant lookup time. A dictionary has constant lookup time because it uses a hash function to map keys to array indices. The hashing function takes the key as input and produces a unique hash code, which is used as the index to store or retrieve the corresponding value. This hashing process allows for efficient storage and retrieval of key-data pairs. Because the runtime of the Levenshtein distance between two strings of length n is approximately $(\log n)^{O(1/\epsilon)}$, where $\epsilon > 0$ is a free parameter to be tuned (2023)[14], it would likely be similar for the same algorithm to be performed on probabilistic strings, where the likelihood of each character is specified in a dictionary, with constant lookup time.

[14] "Levenshtein Distance." *Wikipedia*, 11 May 2023, en.wikipedia.org/wiki/Levenshtein_distance.

3.3 Pseudocode for the Algorithm

Attached below is a screenshot of the code for weighted Levenshtein with some comments on what would be different when performing minimum edit distance on a probabilistic string. In the weighted Levenshtein algorithm, there is a two-dimensional array, or matrix, that contains 128 rows by 128 columns. Substitute_costs[i, j] represents the cost of substituting ASCII character i with ASCII character j. ASCII, or American standard code for information interchange, is a character encoding format for text data. Each letter in the alphabet and numeric or other type of character is represented by one of the 128 values (Loshin, 2021)[15]. In an algorithm for probabilistic strings, the matrix would need to be modified. As noted, there would be a dictionary that contains a mapping of each recognized character with its corresponding percentage of accuracy. The modified code is signified in the orange block comment at the bottom of the figure. There would need to be a method that substitutes the cost of the edit operation based on the activation levels, as indicated below.

[15] Loshin, Peter. "What Is ASCII (American Standard Code for Information Interchange)?" *WhatIs.Com*, 9 Sept. 2021, www.techtarget.com/whatis/definition/ASCII-American-Standard-Code-for-Information-Interchange#:~:text=ASCII%20(American%20Standard%20Code%20for%20Information%20Interchange)%20is%20the%20most,additional%20characters%20and%20control%20codes.

```
cdef DTYPE_t c_levenshtein(
    unsigned char* str1, Py_ssize_t len1,
    unsigned char* str2, Py_ssize_t len2,
    DTYPE_t[::1] insert_costs,
    DTYPE_t[::1] delete_costs,
    DTYPE_t[:,::1] substitute_costs) nogil:
    """
    https://en.wikipedia.org/wiki/Wagner%E2%80%93Fischer_algorithm
    """
    cdef:
        Py_ssize_t i, j
        unsigned char char_i, char_j
        DTYPE_t ret_val
        Array2D d

    Array2D_init(&d, len1 + 1, len2 + 1)

    Array2D_0_at(d, 0, 0)[0] = 0
    for i in range(1, len1 + 1):
        char_i = str_1_get(str1, i)
        Array2D_0_at(d, i, 0)[0] = Array2D_0_get(d, i - 1, 0) + delete_costs[char_i]
    for j in range(1, len2 + 1):
        char_j = str_1_get(str2, j)
        Array2D_0_at(d, 0, j)[0] = Array2D_0_get(d, 0, j - 1) + insert_costs[char_j]

    for i in range(1, len1 + 1):
        char_i = str_1_get(str1, i)
        for j in range(1, len2 + 1):
            char_j = str_1_get(str2, j)
            if char_i == char_j:  # match
                Array2D_0_at(d, i, j)[0] = Array2D_0_get(d, i - 1, j - 1) # It's equal, no cost added, go diagonally down to next letters in each word
            else:
                Array2D_0_at(d, i, j)[0] = min(
                    Array2D_0_get(d, i - 1, j) + delete_costs[char_i],
                    Array2D_0_get(d, i, j - 1) + insert_costs[char_j],
                    Array2D_0_get(d, i - 1, j - 1) + substitute_costs[char_i, char_j]
                    '''
                    What would be done here is remove the previous line of code, and instead insert the following 3 lines:

                    Array2D_0_get(d, i - 1, j - 1) + substitute_cost(list_of_dictionaries[number_letter_in_torah], 0)
                    Array2D_0_get(d, i - 1, j - 1) + substitute_cost(list_of_dictionaries[number_letter_in_torah], 1)
                    Array2D_0_get(d, i - 1, j - 1) + substitute_cost(list_of_dictionaries[number_letter_in_torah], 2)

                    Assuming that the following function exists:

                    cdef float substitute_cost(dictionary_type dictionary_of_particular_letter_in_torah, int n_most_optimal_substitution):
                        predicted_accuracy = dictionary_of_particular_letter_in_torah[n_most_optimal_substitution]
                        if predicted_accuracy is None:
                            return 1
                        return 1 - predicted_accuracy

                    '''
                )
```

Figure 10. Weighted Levenshtein algorithm with modifications for probabilistic strings

3.4 Example of a Probabilistic String

As a simple example, it can be helpful to walk through the process of comparing a probabilistic string with a theoretical Masoretic Text. Using English letters, for the sake of

simplicity, let the Masoretic Text read “TO”. Portrayed below is the input probabilistic string in relation to the Masoretic Text, where the numbers in parentheses represent the activation levels of the top two guesses of the OCR model divided by 100.

Masoretic Text:	T	O
Top guess:	O (.7)	D (.8)
Second guess:	T (.2)	C (.15)

One option for the minimum edit distance algorithm is to accept the ‘T’ as the first character and to insert an ‘O’ as the second character. Accepting the ‘T’ would cost .8, because that is the difference of 1 minus .2, and inserting an ‘O’ would cost one because that is the default cost of insertion. The total cost would therefore be the sum of .8 and one, which equals 1.8. An alternative option for the edit distance algorithm would be inserting a ‘T’ as the first character, and accepting the ‘O’ as the second character. Inserting a ‘T’ would cost one, because that is the default cost of insertion, and accepting the ‘O’ would cost .3 because that is the difference of 1 minus .7. The total cost would therefore be the sum of one and .3, which equals 1.3. The second option is lower than the first option, which means that the minimum edit distance algorithm would choose that solution.

A more complex example of minimum edit distance on a probabilistic string is shown below.

Masoretic Text:	T	O	R	A	H
Top guess:	O (.9)	R (.9)	A (.9)	H (.9)	K (.9)
Second guess:	T (.1)	O (.1)	R (.1)	A (.1)	H (.1)

The minimum edit distance algorithm would take each probability into account and determine whether it is more efficient to insert a character, delete a character, or substitute a character. One possibility is to accept the second guess for every character. That would result in a minimum edit distance of (1-.1) + (1-.1) + (1-.1) + (1-.1) + (1-.1) = 4.5. An alternate possibility is to insert a 'T' as the first character and then accept the top guess for the next four characters. This would result in a minimum edit distance of (1) + (1-.9) + (1-.9) + (1-.9) + (1-.9) = 1.4. The latter option has a lower edit distance, and therefore would be chosen by the algorithm.

4. Conclusion

This concept of using an algorithm that allows for different costs to be specified depending on the edit operation being done is very useful for Biblical research. When dealing with *sifrei Torah* from different time periods, it is common for the scrolls to be damaged or worn out. As a result, it is likely that an OCR model being used to read an image from a particular scroll into a string will output a probabilistic string. It is crucial, however, that the likelihood of accuracy for a particular character being suggested should be taken into consideration when

calculating the cost of that edit operation. It would make sense for the edit operation to cost less when accepting a character that is more likely accurately read than it would to accept a character that is less likely accurately read.

This idea helps with the objective to find *chaseirot* and *yeteirot* in *sifrei Torah*. The minimum edit distance algorithm on probabilistic strings would find the most efficient way to match the probabilistic string with the Masoretic Text. If a character needed to be added to the probabilistic string, that would signify a *yeter*, or a missing character, in the *sefer Torah*. On the other hand, if a character needed to be removed from the probabilistic string, that would signify a *chaser*, or an additional letter, in the *sefer Torah*.

The costs of accepting each character from the OCR output will inversely correlate with the likelihood that they were accurately read. The costs will be specified in the minimum edit distance algorithm and will be stored in a Python dictionary.

Something that would need to exist for this algorithm to work is an improved version of a machine learning OCR model. Some prior models required a significant amount of manual labor in order to separate the true characters from extra ink on the paper. They only worked on single characters, but could not yet accept an entire image of a *sefer Torah* as input. Other models were limited to specific fonts, different from that of Torah scrolls. Overall, no model was able to handle worn out, ancient text, and accurately output a machine-encoded string. Once the OCR model is perfected, that would allow for a minimum edit distance algorithm on probabilistic strings to work at an efficient level. It would save Biblical researchers a tremendous amount of time if they did not have to manually read through every letter in a *sefer Torah* and identify whether there is an additional or missing letter from the Masoretic Text. This concept can be

applied to any text that has damaged letters or text that is worn out. Using minimum edit distance on probabilistic strings is an algorithm that can be useful to many researchers and has the potential to discover hidden meaning behind ancient texts.

Works Cited

Aharon, Alyssa. *Deep Torah Learning: A Deep Learning Approach to Identifying Anomalous ...*, repository.yu.edu/handle/20.500.12202/8249. Accessed 14 May 2023.

"APACHE LICENSE, VERSION 2.0." *Welcome to the Apache Software Foundation!*, www.apache.org/licenses/LICENSE-2.0. Accessed 14 May 2023.

Hládek, Daniel, et al. "Learning String Distance with Smoothing for OCR Spelling Correction - Multimedia Tools and Applications." *SpringerLink*, 7 Dec. 2016, link.springer.com/article/10.1007/s11042-016-4185-5.

Kumar, Arun, and Christopher R´e. *Probabilistic Management of OCR Data Using an RDBMS - Stanford University*, cs.stanford.edu/people/chrismre/papers/HazyOCR_TR.pdf.

"Levenshtein Distance." *Wikipedia*, 11 May 2023, en.wikipedia.org/wiki/Levenshtein_distance.

Loshin, Peter. "What Is ASCII (American Standard Code for Information Interchange)?" *WhatIs.Com*, 9 Sept. 2021, www.techtarget.com/whatis/definition/ASCII-American-Standard-Code-for-Information-Interchange#:~:text=ASCII%20(American%20Standard%20Code%20for%20Information%20Interchange)%20is%20the%20most,additional%20characters%20and%20control%20codes.

"Optical Character Recognition (OCR) for Old Torah Manuscripts." *Signal and Image Processing Lab.*, sipl.eelabs.technion.ac.il/projects/optical-character-recognition-ocr-for-old-torah-manuscripts/. Accessed 15 Dec. 2022.

Oz, Adi, and Vered Shani. *Hebrew OCR with Nikud*, www.cs.bgu.ac.il/~elhadad/hocr/. Accessed 13 May 2023.

Solitreo.Com, solitreo.com/chart. Accessed 14 May 2023.

Su, David. "Welcome to Weighted-Levenshtein's Documentation!¶." *Weighted*, weighted-levenshtein.readthedocs.io/en/master/. Accessed 14 May 2023.

"Tesseract (Software)." *Wikipedia*, 2 Apr. 2023, en.wikipedia.org/wiki/Tesseract_(software).

"Tesseract User Manual." *Tessdoc*, tesseract-ocr.github.io/tessdoc/. Accessed 14 May 2023.

Tesseract-Ocr. "Tesseract-OCR/Tesseract: Tesseract Open Source OCR Engine (Main Repository)." *GitHub*, github.com/tesseract-ocr/tesseract. Accessed 14 May 2023.

"What Is OCR." *Amazon*, 1978, aws.amazon.com/what-is/ocr/.