Kim Soffen
Lisa Wang
Diane Yang
Amna Hashmi

NOTE: post-implementation annotations are highlighted yellow

CS51 Project Draft Specification

Brief Overview:

We would like to build a font recognition software that can take a scanned text-only document, recognize all the letters on said document, and make the document searchable.

Feature List:

1. Given a single letter, recognize what letter it is
   This will use a neural network similar to the one implemented in CS181 Homework 2 (which recognizes handwritten digits). This is described in detail in the below "technical specification" section. We are currently unsure where we will get our training set. We were unable to find a database of standardized typed letters online, so we may be creating it ourselves (unless we manage to find a suitable dataset that already exists). We don't think this set will have to be too large; there are a finite number of fonts (and an even smaller subset of these are actually used) so the data set would not be too large for us to construct ourselves.
   This was accomplished through our neural network, which has a ~95% success rate. We ended up implementing this through CS181 HW 2 just as we expected
2. Given a word, pick out the divisions between letters.
   Given a word, we will detect for columns of white pixels, and draw the line there. There will be two of these lines drawn between each character: the first instance of a white column, and the last one before the first black pixels of the next character appear. This will put each character in its own box, and each space between characters in its own box. If it is a character box, we will then standardize it by adding white pixels around it such that it is the size the aforementioned neural network is trained to process. If the box is too large, we will use the Python image library resize function to shrink down the letter, and then add white pixels to get the proper dimensions. With the boxes of spaces, we will look at the horizontal size as compared to the horizontal size of the letters to determine if it's the space between words, which would need to be stored as a character, or if it's between two characters in the same word, in which case it would be disregarded.
   This was implemented in preprocessing (character_extraction.py). Our process is very similar to what we outlined here except that our output does not differentiate between spaces and characters (we rely on the neural network to make that distinction), and we used a different method for finding spaces. We realized that it might not be practical to use the average width of previous letters to guess the width of a space (for example,

what if our document started with “I am...”? The second character is a space, but the only previous character that it would have to go off of is the “I”.) We instead experimented with using the height of characters as a metric to guess the width of spaces. We ended up using height/2 as the threshold.

3. Given a document, pick out where the letters are on the document (versus whitespace)
   First step will be to create horizontal lines of text. This will be done in a similar way as described above. We will look for rows of all white pixels, and draw a line there. This step will leave us with many rows of characters. Then we will take off the margins on each side by deleting the first and last instance of black pixels in each row. This will leave us with one long column. This will then be split vertically as described above.
   This was implemented in preprocessing (character_extraction.py). The process is pretty much the same except that we do not bother to take off the margins since the process described in bullet point 2 already takes care of this.
4. Take a scanned document and correct for imperfections
   Upon intaking the scanned image, we will increase the contrast so whitespace looks whiter and font is darker. This will allow the pixel detection to be more accurate since it will have more definitive blacks and whites. This will use Python’s image library.
   There may be issues w/ black specks or the document being slanted b/c of how it was scanned, but we suspect the letter recognition will still function, since a slanted letter with specs still looks more like that letter than any other letter.
   Dealing with documents with images is out of the scope of this project; we will assume all black on the page is text.
   If we have time, we’ll deal with if the scanned image is crooked.
   This was not implemented; we are only equipped to work with perfect or nearly perfect .pngs (i.e. flat and unblemished with horizontal lines of text)
5. Make a search function (likely command line: input word, output char #s where it appears)
   We are going to store the contents of the document as a list of characters. (since python lists are able to be up to ~537 million elements, and we will not be considering documents with more than that many characters). Given this list, we will do a linear search for the first character in the query, and then match the next list item with the next query character, etc, until a match is made. We will get the index of the location of the search query and store it in a list of these indices. This list will be returned as the location of the query throughout the document. Alternatively, python has built-in search functions that can find queries in .txt files. When we get to this point, we will compare efficiency between using this function over a .txt file (described in feature #7), or using the list of characters that already exists.
   This is implemented in search.py. However, the implementation was a bit simpler because python is sufficiently high level that we did not have to manuall iterate through a list. We are also only finding the first instance, not every instance of the search.
6. Make search function case-insensitive
   A simple extension of the above function. Will make the entire list of characters lowercase, and will make the query all lowercase. Thus, the new query will match every instance of that query in the new list, regardless of case.

This is implemented in search.py by making all characters lowercase

7. Output characters into a text document.

Transform the aforementioned list of characters into a .txt document which we output.

This is implemented in erm.py, which outputs searchable.txt by concatonating the characters output by the neural network into a string.

8. Make this document visibly searchable

You can type in a search query to the command line, and it will output a .txt file with each instance of that query in all caps.

We did not implement this

Our minimum complete project will cover #1,2,3, and 7 listed above. #4,5,6, and 8 are extensions that we hope to, but may not, get to.

We implemented #1,2,3,5,6,7

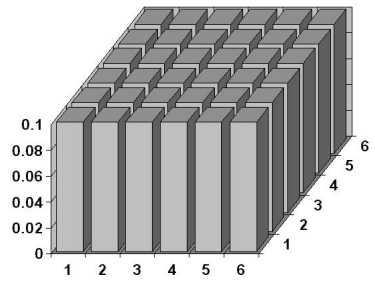
Technical Specification:

We're not quite sure how to modularize this, or if it even makes sense since our project is largely sequential: first you take in the document, split it into rows, then split into columns, then send it through the neural network to determine what letter it is, and then compile all these letters into a .txt document. This is how we implemented it (erm.py). We could potentially make each of these functions a module, but it seems like then, each module would only have one function, which makes the whole thing seem kind of pointless. We would really appreciate some feedback on if/how to modularize our project. We put each type of object (ie: node) into a class in the neural network implementation, and all of the preprocessing code was encapsulated into a class. As far as the implementation of each function/module listed here, the neural network is detailed below, and all other functions are detailed above in the function list.

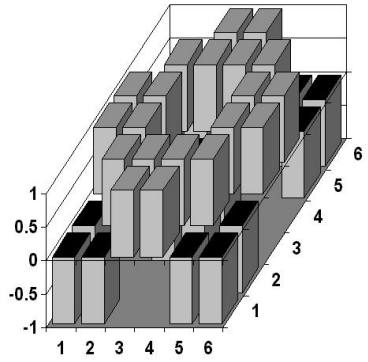
We will be using a neural network algorithm in order to recognize the characters in the scanned document. For the neural network, we will have a class for about 95 nodes (including upper and lowercase letters, numbers, and symbols) which will make up the hidden layer that processes information between the input and output, each of which will be trained to recognize their respective character. Any input will go through the nodes, which will generate an output of whether or not the input matches the character.

So first to train the system, we will define squares for each character, say 40 by 40 pixels. We ended up going with 20x20 pixels to balance increased precision with larger files yet slower run time with larger files. This 2D array of pixels will be converted into a one-dimensional array of size 1600. Each black pixel will be assigned an integer value of 1 and every white pixel will be assigned an integer value of -1. Each pixel will be fed into each node and will generate a weight between 0 and 1. The nodes will be fed into an output layer which will have a single node that will recognize the character. We're basing this algorithm off of one detailed in http://www.oocities.org/santosh_ganti/nn.html.

So as we begin, for the character, the initial weights will look something like



So for the number 0, after the training phase, the weights for say node 0 should look like this.



To summarize, the values at input node, which represent each pixel, (1 for black and -1 for white) will be multiplied by the weights. Then, each node will output the sum of all these products and the closer the output of a particular node is to 1, the more likely the input corresponds to that character. The final node then determines which character the input most corresponds to based on the outputs of the hidden layer.
We misunderstood what each layer did. What we interpreted as the ‘output layer’ was actually a different function all together, and what we interpreted as the ‘hidden layer’ was actually the output layer. We added in a real hidden layer since the variables aren’t linearly separable.
So for the neural network, we’re going to have a class for weight, and a class for the node. The class for weight will just have a value. The class for the node will have attributes, inputs, weights (a list of weights for each input node), a raw value (the linear combination of weights and input signals), and a final value (the output of this node). We’ll also have a class for the neural network itself which will have functions for getting the weights, adding nodes, computing the raw values, and determining whether or not the character corresponds to the input.
We also had classes for neural network framework, and encoded framework

Then for the actual reading of the document, there will be a function to get the list of image objects which will be determined by the pixel size of the square we use.
This is found in datareader.py

What’s Next?:
Language: Python!
Dataset: find one or make one if DNE online

We ended up making one
Repository: GitHub --> make sure everyone can pull/push properly
Frameworks: probably don't need one; if it does come up that we need one, we'll figure it out then
Set up all module signatures by next week.