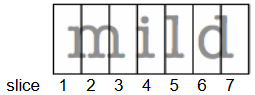
Our first word recognizer has no particular language model built in, but simply tries to produce the most likely interpretation of a sequence of character recognizer observations. We will illustrate how the algorithm works using the word contained in the image in figure 3.

This recognizer uses a simple 1-dimensitional dynamic programming algorithm where the objective function to be maximized is simply the sum of the scores for each character. Each cell in our table represents an endpoint – the optimal solution for the part of the word ending at the slice corresponding to that particular sub problem and a link back to the previous cell in the table representing the optimal solution for the part of the word prior to this letter. As our image has been divided into seven slices, we will have a table with 7 cells, as illustrated in figure 4.

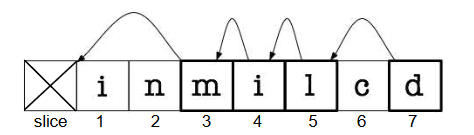
Since a letter is allowed to span up to three slices, there are 3 possible choices for the previous sub problem. For example, cell n could correspond to a character consuming only 1 slice (slice n), in which case it would ink back to cell n-1. If the letter were use up 2 slices (slice n and slice n-1), however, it would link back to the sub problem in the cell n-2. The score that gets stored in a particular cell is the sum of the local letter score and the score stored in the previous sub problem. We choose the letter that maximizes this sum.

The local character score is the probability score for the most likely character (as returned by the character recognizer), multiplied by a scaling factor that depends on how much the mist likely character’s average width differs from the width of the portion of the image being considered for this character.



**Figure 3.** Example word image and its slice. We will

use this example to illustrate our algorithms.



**Figure 4.** Dynamic programming table. Each cell in the table holds the

optimal solution of the problem ending at that point. Each solution is

expressed in terms of the optimal solution to a smaller sub problem

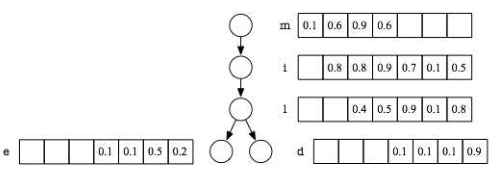
(as indicated by the arrow). In this example, the word is recognized as “mild”

When we have filled in the table completely, the last cell represents the optimal word for the entire sequence of observations. The optimal result can be re-constructed by following the links back from the last cell in the table.

**Dictionary Model**

The second word recognizer we implemented – which has given us the best results – is one that tries to find out which word in a dictionary is the most likely match for a given input image. We will first describe a version of the dictionary-based recognizer that simply scan linearly through the entire lexicon, evaluating the probability for each word, and outputting the word with the highest score. Then, we will describe an alternative organization that allows us to interleave the dynamic programming optimization with the dictionary traversal to compute the most likely word much more quickly.

In this case, the problem we are solving is somewhat different. Here, we use dynamic programming to find the best score we can get if we are forced to interpret a bit-map as a given word. We run this optimization for every word in the dictionary, and pick the word that gave the best score. This time, we have a 2-dimensional table to fill in. Again, each column in our dynamic programming table represents the sub problems ending at a particular slice in the input sequence, and each row of the table represents a letter from the word in question. Stored in this table location is a pointer back to the previous sub problem (the previous letter and the slice where that letter ends) as well as cumulative score (see figure 5). We use a similar local scoring method as the other word recognizer – the probability that the observation matches the letter implied by the current cell, times a scaling factor that depends on the slice width and the average width for the character. Again, the cumulative score is the score for the current cell plus the cumulative score for the previous partial solution. Once we finish filling in the table, the optimal score for the word is stored in the final (upper-right) cell. We then normalize this score by dividing by the number of letters in the word. Without this normalization, long word with relatively poorly-scoring letters can accumulate high score and beat out shorter words that have very good letter score – we want to maximize the score for each letter.



**Figure 6. Trie-based dictionary lookup.** As we visit each node in the tries, new

row of the dynamic programming table is created. We can reuse much of the

work when evaluating the score for “mile” and “mild”

Since many words in the dictionary share prefixes with other words, we are duplicating a lot of work by computing this shared information for each word. Consider the dynamic programming table used to find the score for the word “mild” example. We would like to share these identical rows when computing scores for words with common prefixes.

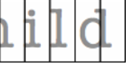
We employ a method similar to the method described by Lucas et al. [7] to speed up our dictionary search. To traverse the dictionary in an order that maximizes the amount of reused computation, we arrange the dictionary into a tree structure. Any node in the tree represents either a partial word or a complete word (or, both – “mild” is a word and also the prefix of “milder”). Now we can build up the dynamic programming table incrementally as we are traverse the dictionary. When we visit a node, we create a new “row” for this virtual table that corresponds to the letter represented by that node, and fill it in. The only context we need for this operation is the previous row, which we pass as a parameter to the recursive tree traversal routine. See figure 6 for an illustration for our tree traversal algorithm. If the node in question represents a full word, we can look at the last entry in the row to find the sum for the scores for the letters in that word. Again, we divide this sum by the length if the word to get our final word score. When the tree traversal finishes, we simply return the highest-scoring word we encountered.

We implemented two heuristics that speed up the computation immensely. First, we only visit the words starting with the letter that are likely to be the initial letter for the word. This optimization gives us a several-fold speedup, especially for words that begin with uncommon letters. As much greater speedup comes from pruning the search to avoid following paths that are unlikely to result in a high scoring word. If the score for the partial word at a given node (again, normalized by the number of letters) is worse than some threshold times the best score so far, we assume no matter how well the remaining letters of the word score, they will never be good enough to beat the best word we’ve seen so far. This second optimization gives us a quite dramatic speedup without noticeably compromising the result

i: character detected at this node

p: probability of correct detection

:remaining slices



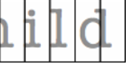
­­

Detected: ‘m’

P = 0.9532



Detected: ‘n’

P = 0.0467

Detected: ‘i’

P = 0.0001



Detected: ‘in’ P = .9995\* 0.0001

Detected: ‘ir’ P = .0005\* 0.0001



Detected: ‘in’ P = .9995\* 0.0001

Reference:

[7]: Lucas, S.M., Patoulas, G and Downton, A. C. 2003. “Fast Lexicon-Based Word Recognizer in Noisy Index Card Images”, In Proceedings ICDAR 2003 7th International Conference on Document Analysis and Recognition, Edinburgh, Scotland, August 3-6, 2003, Volume I: 462-466

[1]: Text Recognition of Low-resolution Document Images, Charles Jacobs, Patrice Y. Simard, Paul Viola, and James Rinker

**#Dictionary Model:**

The second word recognizer we implemented – which has given us the best results – is one that tries to find out which word in a dictionary is the most likely match for a given input image. We will first describe a version of the dictionary-based recognizer that simply scan linearly through the entire lexicon, evaluating the probability for each word, and outputting the word with the highest score. Then, we take the top 10 letters that are predicted in each layer of the tree (i.e. at a particular height) and store them in a priority queue. For example: We are using 3 slices for almost each letter, therefore for “mild”, we divide m into 3 slices. The predictor may predict 1st slice as ‘i’ with the probability of 0.0001. With second slice, the predictor may predicts it as ‘n’ with the probability 0.0467 and with the 3rd slice we get ‘m’ as the result with the probability of 0.9532.

We then take top 10 node with maximum probabilities and store them in a priority queue. Let us assume that the detected letter is ‘i’ and then we perform this division process. Again, we’ll have different letters detected. ‘ir’, ‘in’, ‘in’ in this case. We then multiply the probability of the character in the previous node and the probability of letter predicted in the second node. Then we again take the top 10 results and store them in priority queue.

At the nth layer of tree, we will have all the possible words that can be there and then we select the one with the maximum probability i.e. “mild” in this case.

Since the handwriting of doctors are very poor and sloppy. We will not discard the initial predictions of the word i.e. ‘mi’, ‘mil’ etc. because these can help us detect the correct prefix of the word and it will make task easier for our autocorrect to detect the correct word which is written there.

#General Case:

**#Dictionary Model:**

The second word recognizer we implemented – which has given us the best results – is one that tries to find out which word in a dictionary is the most likely match for a given input image. We will first describe a version of the dictionary-based recognizer that simply scan linearly through the entire lexicon, evaluating the probability for each word, and outputting the word with the highest score. Then, we take the top 10 letters that are predicted in each layer of the tree (i.e. at a particular height) and store them in a priority queue.

Consider a general case where the number of layers in the tree is ‘n’. We have taken the size of traversing window to be 1/3rd of the total width of character, so therefore each character will get sliced into approximately 3 parts. Consider the word that we are trying to recognize is “mild”.

At the first layer of the tree, character m will be sliced into approximately 3 parts. Each one predicting a letter with a definite probability, we connect those predicted letters to the root of the tree. Then we select top 10 predictions in that layer and add them to the priority list and rest of the nodes are discared making this process computationally efficient. Similarly we go on dividing the remaining word into slices and predict the next character. But the probability of the next node is equal to the prediction probability of current node times prediction probability of the current node. This help us determine the probability of the word that is going to be formed at the bottom of the tree. We continue this slicing and select top 10 probable words. We obtain our desired word at the bottom of the tree but we do not discard the predicted upper layers as they act as prefix for many words eg. “mild” acts like a prefix for “milder”. These predicted prefixes will help our autocorrect model to produce a better result in an efficient way.

Example: “mild”

|  |  |
| --- | --- |
| i, l, j, n, a, o, m | First layer prediction |
| in, ln, jn, nn, nr, an, ar, on, or, mi | Only top 10 predicted words |
| ….. |  |
| mild, milo, mill, miid, ….. upto 10 | Last layer prediction. We observe that maximum probability word is “mild”. And hence we opt it |