APPLIED DATA SCIENCE PROJECT 4 SECTION 2, GROUP 3

Papers: D-3, C-1

Samuel Kolins (sk3655), Sheng Wang (sw3224), Colleen Wang (yw3177), Jiaxi Wu (jw3588), Wanyi Zheng (wz2409)

PROJECT GOALS

- Inputs
  - Ground truth text: 100 text files of varying lengths (usually thousands of words or more)
  - Output from the **Tesseract** OCR algorithm, all of its mistakes intact
- **Goal**: Run Tesseract output through two algorithms, *error detection* and *error correction*, in order to detect and restore errors to more closely match to the ground truth text.
  - Devise or use an evaluation metric to record performance of the algorithms.

ERROR DETECTION (D-3)

- Error detection is a binary classification problem. That is, we wish to determine which tokens are garbage and not garbage.
  - Garbage: tokens with enough errors as to be rendered indecipherable to those who can read English (example: "mdf5tb;/v")
  - Not garbage: tokens that might be *incorrect*, but potentially correctable into a valid English word or phrase (example: "Cplumvia Umiwrsity")
  - Correct tokens are obviously not garbage
- To solve this problem, we created a feature list based on the paper and ran them through a support vector machine (SVM).

FEATURE LIST

1. The length l of the input string (token).
2. The number of vowels v and consonants c in the token, as well as the ratios $\frac{v}{l}, \frac{c}{l}, \frac{v}{c}$ $(c \neq 0)$.
3. The number of special non-alphanumeric symbols s and the ratio $\frac{s}{l}$.
4. The number of digits d and the quotient $\frac{d}{l}$. Note that $d + v + c = s$.
5. The number of lowercase (uppercase) letters low (upp) and the ratios $\frac{low}{l}, \frac{upp}{l}$.

FEATURE LIST

6. Let the length of the maximal sequence of identical characters be denoted by m. If $m \geq 3$, record $\frac{m}{l}$ for this token. Otherwise, record 0.
7. Let α be the number of alphanumeric characters in the token. If $\alpha > s$, record 1 for this feature. Otherwise, record 0.
8. If the input token contains a subsequence of ≥ 6 directly consecutive consonants, record 1. Otherwise, record 0.
9. After deleting the first and last characters in the token, if the resulting infix has at least two non-alphanumeric characters, record 1. Otherwise, record 0.

FEATURE LIST (BIGRAMS)

- In addition to the nine basic features, we added a few more advanced features as well.
- **Bigrams.** We compiled a frequency list L_B of the (consecutive) bigrams as they appear in the ground truth data (normalizing to lowercase). Using these numbers, as well as some data easily computed from the input token, we compute the following:

$$\beta = \frac{1}{kn}\sum_{i=1}^{n} b_i$$

- Here, n is the number of bigrams in the token, b_i is the frequency of the ith bigram in L_B, and $k = 10{,}000$ is a scaling factor. Garbage tokens have smaller β's on average.

FEATURE LIST

- **Most frequent symbol.** Let i be the number of occurrences of the most frequently occurring symbol (doesn't matter what the symbol is). If $i \geq 3$, record $\frac{i}{l}$; if $i \leq 2$, record 0. This feature is similar to Feature 6 and is motivated by the observation that most garbage tokens repeat the same symbol multiple times.
- **Non-alphabetical symbol ratio.** Let L_1 and $L_2 = l - L_1$ represent the number of alphabetical and non-alphabetical symbols in the token respectively. Record $\frac{L_2}{L_1}$.

LEVENSHTEIN DISTANCE?

- The D-3 paper uses one final feature called Levenshtein distance which we did not implement because of its negative effect on computation time; it is noted in the paper as single-handedly boosting performance ratings by a few percentage points.
- Essentially, the Levenshtein distance between two strings is the minimum number of character insertions, deletions, or substitutions necessary to turn one string in another. As an example, the words $string$ and $train$ have a Levenshtein distance of 3.

$$string \rightarrow \_tring \rightarrow trin\_ \rightarrow train$$

- A token is plausibly correctible if the Levenshtein distance does not exceed a certain predetermined bound. Garbage tokens tend to not be plausibly correctible.

SVM PERFORMANCE

```
[[16297  2093]
 [ 3459  8580]]
             precision    recall  f1-score   support

          0       0.82      0.89      0.85     18390
          1       0.80      0.71      0.76     12039

avg / total       0.82      0.82      0.82     30429
```

ERROR CORRECTION

- The correctly classified not-garbage tokens are funneled to the error correction algorithm for attempts at being corrected.
- Typically, error correction algorithms use either a dictionary-based algorithm or matrices of precise n-gram probabilities to make their corrections.
  - In the former case, each token is compared to a dictionary. If it matches, the token is considered correct. If not, it is corrected to whatever the closest dictionary word is (perhaps using Levenshtein distance). It's possible that there is no singular best word choice, in which case the token is flagged as an uncorrected error.
  - The latter case uses n-gram probabilities to decide where the errors are and corrects them to the most probable n-gram available (perhaps also under the condition that the fewest character substitutions are made).
- Ideally, all detected errors at the n-gram level are corrected, but this almost never happens. Minimizing false negatives (undetected, uncorrected errors) is best then.

POSITIONAL BINARY N-GRAMS

- The problem with the previous two methods is that they both require a great deal of computation time and storage space. To minimize these issues at only a small cost to overall performance, we will be using **positional binary n-grams**.
- Rather than record the precise probability of an n-gram appearing, we can instead simply record a 1 if the probability of the n-gram appearing is non-zero and 0 otherwise. Each entry in the positional binary n-gram then only requires a single bit of storage.
- Most of these arrays (matrices if $n = 2$) are extremely sparse; according to the C-1 paper, only 40 percent of the 676 English letter pairs appear consecutively in any word.

POSITIONAL BINARY N-GRAMS

- The "positional" part refers to the fact that we will need a different array for each set of positions. For a word of length l, this translates to $\binom{l}{n}$ total positional binary arrays.
- Each array has 26^n entries in it, meaning the total number of bits needed is $\binom{l}{n} \cdot 26^n$.
- For six-letter words, positional binary trigrams only require $351{,}520$ bits (≈ 44 KB) of storage!
- These arrays also can be read-only since the algorithm we used never modifies them, so computation requires even less RAM than normal positional n-grams would.

A WORD OF CAUTION!

- If a comparison analysis finds that the token contains an n-gram with a zero entry in the corresponding positional binary array, then we can conclude an error has occurred.
- But *the converse is not necessarily true!*
- For example, consider the garbage word **SUT**. If our corpus is the English dictionary, each of the positional binary bigrams (**SU_**, **S_T**, **_UT**) are non-zero entries because of the words **SUN**, **SAT**, and **CUT** respectively. We could, however, use a positional binary trigram (on 3-letter words) to detect **SUT**. A dictionary would be more efficient in this case though.

FINDING & FIXING ERRORS

- Suppose we want to use positional binary trigrams to find errors in a six-letter word. We will assume that there are no more than two errors in a word we wish to correct; anything more than that is too complicated to correct and is not attempted by our algorithm.
- We first wish to return the list of positional trigrams that return an error. Suppose that list looks like this (in table form):

1	2	3	4	5	6
X	X	X			
X		X	X		
X		X		X	
X			X	X	

FINDING & FIXING ERRORS

- If there is only one error, it must be in all of the listed trigrams. Therefore, in this case, the singular error must be in position (1).
- However, there could be two errors. To determine this, we check to see that, if we take the union of just two columns (and discard the rest), there is an X in each row. Therefore, in addition to (1, #) where # is a wild card, the error positions could also be (3, 4) or (3, 5).

1	2	3	4	5	6
X	X	X			
X		X	X		
X		X		X	
X			X	X	

FINDING & FIXING ERRORS

- Here's a more complicated example:

1	2	3	4	5	6
X	X		X		
X		X		X	
X				X	X
	X	X			X
		X	X	X	
			X	X	X

- The only possibility (for two or fewer errors) is (2, 5).

FINDING & FIXING ERRORS

- Once the candidate error positions are determined, the algorithm runs through all the possible allowable alternatives (using the same positional binary n-grams to determine what is legal) and makes a correction if there is a single viable choice. If not, the error is flagged but remains uncorrected.
- Compared to the dictionary algorithm, this results in many more uncorrected *flagged* errors, but the number of *unflagged* uncorrected errors (false negatives) is about the same, which is enough to make the computational efficiencies worthwhile.
- It's possible that the algorithm cannot precisely determine what the correct error position is (if there's only one error and only bigrams are used). In this case, we can check the possible corrections anyway to see if only one correction between the two locations is viable; if so, we make that correction. This is called ***positional determination by elimination***.

CORRECTION PERFORMANCE

	Tesseract	Tesseract w/ post-processing
Word-wise recall	60.4%	63.7%
Word-wise precision	60.4%	63.7%
Character-wise recall	?	96.6%
Character-wise precision	?	93.4%

THANKS FOR LISTENING!