Proceedings of the 2023 International Conference on Software Engineering and Machine Learning
DOI: 10.54254/2755-2721/27/20230202

A study on how to improve the accuracy of auto error correction from voice to text

Yumo Fan

University of Rochester, Rochester, NY, 14627, US

yfan26@u.rochester.edu

Abstract. Voice recognition application has been widely used in people's daily lives. Usually, if people find it inconvenient to directly play the audio in public, or it is too noisy for them to listen to an audio message, they would probably use the voice recognition application in their mobile devices to translate the voice into text, so that they can understand the message clearly. However, there are still various errors and problems occurring during the usage of this common application, for instance, it can be hard sometimes to translate the sound into words correctly. This usually happens when the speaker speaks too fast and pronounces unclearly. Besides, some other factors, such as environmental noise, transmission channel quality, and the radio equipment, would also cause this problem. This paper mainly analyzes the causes of the inaccurate translation problem and some potential improvements to make this function more perfect. In conclusion, to solve this problem, the parity check matrix is a good way, since it can check whether the digits behind each word are still correct. After doing so, the digits will be changed into the correct words syntactically. However, even though the words are syntactically correct after applying the parity check matrix, they might not be semantically correct. Therefore, Levenshtin distance and Latent Semantic Analysis can be used to analyze the hidden meanings of words and sentences, so as to find the best suitable words to change.

Keywords: Error correction, Voice-to-text, LSA, Levenshtin Distance.

1. Introduction

Voice-to-text function in electronic devices is typically used when the receivers are inconvenient to directly listen to the voice of others, especially when the voice message is quite long. The voice-to-text function can be helpful in this case, as receivers can process information by reading the text. However, automatic speech recognition (ASR) systems are prone to errors, particularly in inadequate environments [1]. Despite the latest development, misspellings and linguistic errors in the output text still exist in ASR systems [2]. The most probable reason for the occurrence of errors in the message can be the disruption of the noisy environment. Superior and accurate results can be attained if the recognition process occurs in a quiet location rather than a noisy open place. Additionally, it is important to note that the quality of input devices can also impact the SNR (Signal-to-Noise) ratio, in addition to the raw noise in the setting [2]. If this situation really happens, then the digits consisting of the information finally received on the other device would be incorrect, and what people can see may be nothing more than some special symbols if forcing to translate. For short voice messages, this problem would be acceptable, because people can often correctly guess what the others said, but when the voice message gets quite long, this problem would be serious since it can be difficult for people to successfully understand the information. This paper envisions several error correction techniques to reduce errors and improve the accuracy of ASR systems. Some techniques are manual, as they involve post-editing the recognized output transcript to correct misspellings. Others use enhanced acoustic mathematical models to improve the interpretation of the input waveform and prevent errors at early stages [3].

2. Error correction 1: For computers

The hardest point for correcting errors is that it is impossible to make any encoding process before the electronic devices of senders firstly get the message, because the very first message giver is the sender’s mouth, which may probably make the whole message wrong before the electronic devices firstly receive the message. Therefore, what the auto-correction can do is to “guess” what the sender really wants to say from the perspective of computers by using some algorithms. This section will mainly introduce some ways to improve the whole accuracy for error-correcting.

2.1. Background information

Whatever the message that the computer needs to store or transmit, they are all in the form of “1”s and “0”s. If a computer needs to store or transmit some information, it will use 1 and 0 to represent the information on the disk. When a computer receives the message containing 1 and 0, it will then automatically interpret them and use human language to present the whole message to the users. However, some errors would easily occur in such a way of storing and transmitting information, and the errors would become very hard to fix. Most errors resulting in an inaccurate voice-to-text translation mainly occur during the transmission process when the output information is not consistent with the input information. In theory, if the channel is "noiseless", every bit of information that goes in will come out unchanged without any issues. However, in practice, noise is typically added to the information, and as a result, introduced by the channel [4]. Digital signals are susceptible to noise during transmission, which can lead to errors in the binary bits as they travel from one system to another. By using the ASCII code, computers can easily interpret 1 and 0 into English letters or translate English letters into 1 and 0, but this way of information transmission also makes it possible that a small error can easily make the whole code unable to be interpreted. As a result of noise, a 0 bit may be changed to a 1 or a 1 bit may be changed to a 0 during transmission. For example, the letter “A” is “0100001” in the ASCII code. If only one 0 or 1 is changed, then the whole binary code can not be interpreted by the computer.

2.2. Parity bit

A parity bit, also known as a check bit, is an additional bit added to a binary code string. It is a simple form of an error-detecting code and is commonly applied to the smallest units of a communication protocol, such as 8-bit octets (bytes). However, it can also be applied to an entire message string of bits. After a byte of data is transmitted, a "Parity Bit" may be added by the transmitter for simple error checking. The receiver can then use the Parity Bit to check for errors [5].

Parity bits come in two variants: even and odd parity. They ensure that the total number of 1-bits in a string is either even or odd. With even parity, the bits with a value of 1 are counted, and if that count is odd, the parity bit is set to 1 to make the total count of 1s (including the parity bit) even. If the count is already even, the parity bit's value is 0. With odd parity, the coding is reversed. The parity bit is set to 1 if the count of 1-bits is even to make the total count of 1s odd. If the count is already odd, the parity bit's value is 0.

2.3. Parity check

During message transmission, noise or corruption may scramble the message data. To detect transmission errors, error-detecting codes can be added to the digital message. A common error-detecting code is the parity check, which adds additional data to the message. Before sending the message to the receivers, the senders will also make some adjustments to the whole content that it will send soon to make it not just purely messages with 1s and 0s, but also some "rules" of constituting the 1s and 0s, like which parity bit that it takes and how many 1s that the original parity bits should contain. If the code word received by the receiver does not meet the expected rules, the computer knows an error has occurred. In addition to the error-detecting code, the sender can include additional data to determine the original message from the corrupt message. This is known as an error-correcting code. Error-correcting codes use the same strategy as error-detecting codes, but can also identify the exact location of the corrupt bit. The parity check used in error-correcting codes detects errors and also determines the location of the corrupt bit. Once the corrupt bit is identified, its value is changed (from 0 to 1 or 1 to 0) to recover the original message. However, this is just the ideal situation where the error is easily located. Doing a parity check for each code word is very redundant and inefficient for each 8-bit code word, since in the real-world information-transmitting process, it is very common to transmit a few hundred of 8-bits binary code words each time in a short sentence. Therefore, finding another way to deal with such an enormous data set at once is very necessary.

2.4. Parity check matrix

Parity-check matrix of a linear block code C is a matrix that specifies the linear constraints that code words must satisfy, determining if a given vector is a code word and indicating the number of errors in the input code word. When using the parity check matrix, the user must first arrange all the n length code words into a matrix n*k.

The information theorem tells that a valid input code word must satisfy that the product of the parity check matrix and the transpose of the generator matrix is zero, and if it is not zero, then there are errors occurred. Also, the parity check matrix can detect how many errors occur in the given code words even though all those errors can not be fixed by only using the parity check matrix. Therefore, by multiplying the transpose of the generator matrix with the parity check matrix, it will finally get all zeros in the result matrix, which shows that the code word is valid and no errors occur in this code word. In order to solve the problem that the product is not zero and find a way to know which bit is incorrect, the result of the product of the parity check matrix and the transpose of the code word is the key. By comparing it with the parity check matrix, the final result matrix is the same as the n^th column of the parity check matrix. Therefore, the nth bit of the original code word is incorrect. If the final result matrix all consists of zeros, then it just shows the code word is valid, so there is no need to change anything. However, if the final result matrix can not match any column of the original code word, then this shows that there are at least two errors in the original code word, which is impossible for people to use the parity check matrix to correct. Therefore, other approaches should be considered in order to have a more accurate error correction.

In conclusion, the parity check matrix is an efficient way to correct single-bit errors. As the example shown above, if there is only one error in the code word, then calculating its parity check matrix and using the parity check matrix to multiply the transpose of the generator matrix is very easy. If the final result matrix can match the nth column in the parity check matrix, then it also shows that the nth bit in the original code word is wrong. However, if there are more than 1 error occurred in the original code word, using the parity check matrix would not solve the problem. Therefore, when encountering such a situation, another approach should be taken into consideration. Overall, the use of parity check matrices is a powerful and effective technique for detecting and correcting errors in digital communication systems. While it may not be suitable for all applications, the use of parity check matrices remains a valuable tool in the field of digital communication and signal processing.

3. Error correction 2: For human

The last section mainly introduces how to correct errors that occur in the code word in order to let the computer interpret the message it receives correctly. However, a more important problem for voice to text function is that it may not translate correctly into the one that the sender wants to express. The problem is that before the receiver gets the message, the sender would not know whether they correctly send the voice message due to many disruptions, like unclear pronunciation and environmental noise. This is not the same as text chatting, in which the sender can clearly see what they typed in. Therefore, even if the previous error correction stage accurately translates the code word into English letters, the English letters may originally be wrong, and some code words may originally be unable to translate by only applying the parity check matrix. That is why a more stable way to self-check the whole sentence is needed even if the previous error correction does not work as well. Therefore, this section will mainly discuss and analyze two ways to analyze what the senders originally want to say in order to eliminate the semantic errors as best as the whole auto-correction process can do.

3.1. Levenshtin distance

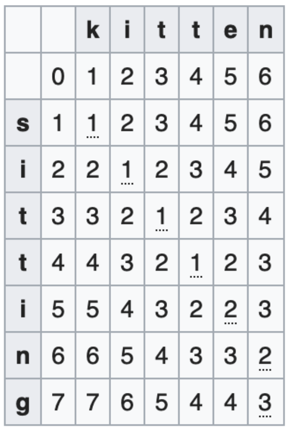
After the first transferring process, the word may still be weird and not correct lexically due to some happened interruption. The most general solution would be to pick the word that may be wrong and then compare it with the whole dictionary, and finally choose the word with the smallest difference as the suggested word to change. Before doing so, it is necessary to find a way of calculating the difference between two words, and using Levenshtin distance may be the best tool here.

Levenshtin distance is used to measure the dissimilarity between two strings, where a higher distance indicates a greater difference between the two words. The problem of searching for strings with differences is often referred to as string matching with k differences in literature [6]. Calculating the Levenshtin distance needs three methods: replace, delete, and insert, and each of them takes a maximum of 1 step to do, and then to perform those three methods of each of the strings and to see how many steps that one of them takes to be transformed to another. More generally and concisely, the Levenshtin distance is recursively defined and takes the below formula to do.

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min\begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases} \tag{1}$$

In order to better understand this formula, two strings should be in the form of a matrix, and the i, j above are the coordinates inside the matrix. The numbers of rows and columns of this matrix are decided by the length of those two characters. The next goal is to fill this matrix, and the Levenshtin distance is the number at the right bottom position of the matrix.

Now here is the example of calculating the matrix of Levenshtin distance between “kitten” and “sitting”. Since in the computational theory, a string’s index starts at 0, so kitten has the range of [0,5], where 0 represents k and 5 represents n, and the same for sitting. The empty string is special, since the number of steps for changing an empty string to any string will depend on the length of the string. Therefore, the first row below the kitten and the first column near sitting is the Levenshtin distance for transferring the empty string to each word, corresponding to 0 to 6 and 0 to 7. More importantly, the real worthy coordinates actually start at [1,1], because that is where the first letter of each word is. And each grid in this matrix represents the Levenshtin distance of the string between [0,i] and [0,j], and i, j represent the columns and rows. For example, the Levenshtin distance between [0,4] and [0,3], which is [4,3] in the matrix, is 2. This makes sense since [0,4] represents the kitten and [0,3] represents the sitten, and there is only a need to replace the k with s and then delete the last e, which only takes two steps. And the whole matrix is given below:



		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 1. The matrix for Levenshtin distance between kitten and sitten.

The next step is to find the word that has the minimum Levenshtin distance. Supposed that the whole English word dictionary can be the search base, and a hash table is created using the Levenshtin distance as the key, and the correct word from the search base as the content in the following list. After looping the whole dictionary, the whole hashtable can be accessed in the data base by using the key with the minimum Levenshtin distance. However, this method has a very large time complexity and space complexity, since it loops and stores many useless data which will all take time and space. In the reality, people typically search for the letter that they are sure in the word and look at those words in the dictionary with this letter one by one if they look similar. And if none of them looks like the right one, they will go to think about another letter in the words. Therefore, this search strategy can also be applied to a computer to search.

When people are normally talking, they typically would pronounce the first letter correctly, and then due to rapid talking speed and other phonetic phenomenons, the middle part or end of the word would be not quite clear. Therefore, the search base can be narrowed to those words with the same beginning letter and the second letter. And in fact, in the worst case, after being processed by the parity check matrix, the word will not be with all incorrect letters, and at least 1 or 2 letters are correct, which can be used as the search key. After determining the search key, the next step is pretty straightforward: loop over the whole dictionary that has the search key and calculate their Levenshtin distance. Noted here, it is unnecessary to store each word that has been searched with its Levenshtin distance, and actually, there is only a need to keep track of the word that has the current minimum Levenshtin distance, and that word, in the end, will become the word that is most similar with the original problematic word.

However, there is an obvious problem that the letters at front of the words can be incorrect. In fact, the more correct the front letters are, the less words need to be searched. If the correct letters occur at the back half of the word, there is no difference from searching the whole dictionary and it could be even worse. Therefore, doing some pre-processing steps to make the search process easier is pretty considerable.

In conclusion, Levenshtin distance can guarantee to find a correct and most similar word, but also there is a trade-off between the computational complexity and the efficiency. If the requirement is to find the most appropriate answer, the whole program has to sacrifice more time on searching, but if it cares less about accuracy, the whole program can reduce the computational time and give the answer fast. Overall, the Levenshtein distance algorithm is a powerful tool for measuring the similarity between strings and is widely used in many different fields. While it may not be suitable for all applications, it remains a valuable and effective approach for many types of string comparison tasks. Whatever it can find the answer by using Levenshtin distance, there is still the next step to complete the whole auto-correction step.

3.2. Latent semantic analysis (LSA)

After applying the Levenshtin distance and converting the incorrect words into correct ones, it still can't determine whether the words are correct semantically. Since the disruption in the process of transmission would change the original word structure and make it become a totally new word which would still be a correct word syntactically, but not semantically, finding an approach to somewhat improve this situation is important and Latent Semantic Analysis (LSA) may be the most suitable way to solve this dilemma. Latent Semantic Analysis (LSA) is a statistical method that extracts and represents the contextual meaning of words [7]. LSA can help find and substitute words by drawing connections between them based on their meaning in a text. To start, it is necessary to determine which words need to be corrected more, and this is easy to do here. Even if the word is corrected by the previous process, LSA can still be used to do more corrections on this word.

Suppose there is several sentence or paragraphs received, and the whole content is called a document. First, all the formats in this document should be removed, and the capital, if any, should be changed in this document into lowercase, and also any punctuation, meaning to change the whole document into pure word with space in order to make our next process more convenient in the pre-processing stage.

TF-IDF (term frequency-inverse document frequency) is utilized to calculate the weight of each word in a document, including the target word for replacement. This process generates a new representation of the document, which can be compared with other documents to determine their similarity [8]. TF-IDF needs many documents that can be analyzed. In this case, the best sample set may be the verbal content that many different people said because the topic is auto-correction from sound to text. Accordingly, the document set can be many paragraphs containing what people said and write. First, all the words in the original document will be arranged as terms. Then, each term's occurrence frequency will be calculated in each document, as the term-frequency process. After finishing this step, the IDF step simply calculates the logarithm of the number of documents that contain each term divided by the whole number of documents. Finally, the whole TF-IDF process is done by multiplying the term frequency with the inverse document frequency and normalizing the result, so here comes the weight for each term.

After the weight for each term is calculated, the document-term matrix is used to store the weight: the row represents each document, the column represents each term, and then the weight is inside it, so this matrix will show the weight for each term in each document. Since this matrix has a very high dimension, which will make it very hard to be analyzed directly, it will lower its dimension by cutting off some relatively irrelevant terms. SVD (Singular Value Decomposition) can be used for this purpose. It is a method that identifies and sorts the dimensions that produce the most variety in the data [9]. In short, SVD separates the original matrix into three matrices: U, sigma, and the transpose of V, and here is the specific formula:

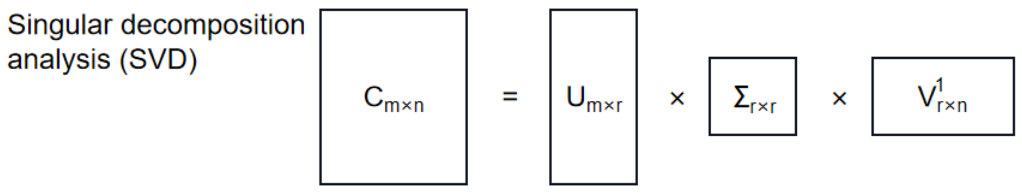


Figure 2. The SVD formula.

The specific steps of how to decompose a matrix into thee separate matrices are not shown here, since it will have much linear algebra matrix manipulation, but it can be done by computer using python. After those three matrices are obtained, the k value is decided by how large the original document is and the specific requirements, and typically k is to be set within 50-300. Again, the value of k depends on different situations and requirements, since the main purpose of k is to narrow down the three matrices, thus lowering the dimension for the original matrix. To be specific, only the first k column for the U matrix and the first k rows for the sigma and the transpose of the V matrix is kept, and then multiply them together again. Therefore, there will be an entirely new matrix with a lower dimension and the weight is updated for each term. However, there is a trade-off between preserving the original data and reducing computational complexity here. If very exact accuracy is not needed, and this whole process should be done quickly, then obviously k with a small value is practical here, because the listener may need the correct message urgently. Besides, k with a large number may be more suitable for some large scientific research, such as space exploration.

Finally, after the new matrix with a lower dimension is obtained, the next step is to analyze the similarities between terms and the relationship between terms and documents and to cluster documents. To obtain the value of similarity between reference documents and student documents, the Cosine Similarity Measurement is used [10].

$$\text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i-1}^{n} A_i B_i}{\sqrt{\sum_{i-1}^{n} A_i^2}\sqrt{\sum_{i-1}^{n} B_i^2}} \tag{2}$$

The basic idea for using cosine similarity is that if two terms are similar, then they will be similar in weight vectors. Therefore, if A is an incorrect term or the term needed to be changed in the given document, and B is correct in another document in the documents collection, then B can be used to substitute A. The cosine similarity will be within the range -1 to 1, and the higher the cosine similarity that A and B have, the more similar they are in semantics.

For a better auto-correction result, all the terms should be checked and not just those being changed by previous steps, but there is also a trade-off between the time complexity and the entire accuracy here. Checking all the terms may seem practical for small documents, like those not too long soundtracks, but for some long soundtracks may seem very time-consuming, since if each word is checked with another different word in the whole term set, the time complexity is the square of n which is the number of terms, and the reality may seems more complex when doing the calculation.

In conclusion, Latent Semantic Analysis (LSA) is a mathematical technique that analyzes word relationships in a text corpus. It works by identifying patterns of word co-occurrence and grouping similar words together to form "concepts" or "topics." LSA has been effectively applied in various domains such as natural language processing, information retrieval, and machine learning. One of the main advantages of LSA is its ability to identify hidden or latent relationships between words that might not be immediately obvious. It can also help to reduce the dimensionality of large text datasets, making it easier to analyze and visualize complex relationships between words. However, LSA has some limitations, such as its inability to handle polysemy (multiple meanings of a word) or to capture the nuances of language, such as idiomatic expressions. Overall, LSA is a useful method for analyzing and interpreting the meaning of large volumes of text. It is a field that is constantly evolving, with ongoing research and development, and it has proven to be effective in many applications, such as natural language processing, information retrieval, and machine learning.

4. Conclusion

In conclusion, this paper mainly introduces and analyzes two approaches of auto-correction from voice to text. The first approach is to use the parity check matrix to detect and correct the errors made in the process of transmission. In the process of transmission, the soundtrack may be disrupted by factors such as electronic disruption and environmental noise, and these disruptions could change the basic digits that consist of the soundtrack, thus making some parts or even the whole soundtrack unable to be compiled on the receiver's computer. Using the parity check matrix will guarantee that the final message is presented on the screen in a way that humans can understand. However, using the parity check matrix only does not always help convert the soundtrack into the correct message, since the words may be lexically correct after applying the parity check matrix but not semantically precise. In that case, Levenshtin distance and Latent Semantic analysis can be used to further correct the potential problematic words. Using Levenshtin distance can help guarantee the finding of the best similar word based on the whole structure of the original word, and using LSA can help guarantee the finding of a word in a document that best fits into the potential meaning. However, using all those three methods in an auto-correction process may have huge computational complexity, thus resulting in the failure of receiving the translated message on time. Therefore, future research can focus on the solution to reduce the computational complexity to make the whole process faster.

References

[1] Deng, L. and Huang, X. Challenges in adopting speech recognition. Communications of the ACM 47(1), 60-75 (2004).
[2] Bassil, Y. and Alwani, M. Post-Editing Error Correction Algorithm For Speech Recognition using Bing Spelling Suggestion. International Journal of Advanced Computer Science and Applications (IJACSA) 3(2), (2012).
[3] Rudnicky, A. Hauptmann, A. and Lee, K. Survey of Current Speech Technology. Communications of the ACM 37(3), 52-57 (1994).
[4] Van Lint, J. H. Coding theory. Springer-Verlag Berlin-Heidelberg-New York (1992).
[5] Rani, M. U. and Veni, R. S. A VLSI Based Design of Reduced Power On-Chip Coarse-Grained Network Processor. International Journal of Engineering Research and Applications 5(1), 50-53 (2015). ISSN 2322-0929.
[6] Navarro, G. A guided tour to approximate string matching. ACM Comput. Surv. 33, 31-88 (2001).
[7] Landaeur, T. K. Foltz, P. W. and Laham, D. An Introduction to Latent Semantic Analysis. Discourse Processes 25(2-3), 259-284 (1998).
[8] Ratna, A. A. P., Sanjaya, R., Wirianata, T. and Purnamasari, P. D. Word Level Auto-correction for Latent Semantic Analysis Based Essay Grading System. In Proceedings of the 15th International Conference on Quality in Research (QiR) and International Symposium on Electrical and Computer Engineering. IEEE. (2017). ISBN 978-602-50431-1-6.
[9] Landaeur, T. K. Foltz, P. W. and Laham, D. Learning Human-like Knowledge by Singular Value Decomposition: A Progress Report. In NIPS, Minneapolis (1997).
[10] Muflikhah, L. and Baharudin, B. Document Clustering using Concept Space and Cosine Similarity Measurement. In ICCTD, Kuala Lumpur (2009).