Learning Shuffle Ideals

The Problem

You are building a machine that takes a set of strings as input, is told which of those strings match an unknown rule, and tries to come up with the implicit rule being used. In a general sense, this could be any rule like: The string starts with the character ‘a’. Or, the string is of length 10. Or, the string is a palindrome; that is, it is the same forward and backward. If a string matches the rule, it is a positive example, and otherwise it a negative example.

Ideally, one would be able to create a machine that can figure out any such arbitrary rule with perfect accuracy, while only needing relatively few example inputs. This is a rather lofty goal, however, so we will concern ourselves with devising an algorithm for the machine to be able to deduce a specific kind of rule, or concept, with reasonable probability of success: the shuffle ideals.

Shuffle ideals are sets of strings that all contain some designated string (the shuffle string) as a subsequence. This means that the characters of the shuffle string appear, in order, within the sample string, with interpolation of other characters allowed. For example, if our shuffle string was “ace”, then aceb, aaced, baecbe, bdbabaaddbced would all count as strings in the associated shuffle ideal, while cac, bbadcd, and dcc would not. The goal, then, would be to find an algorithm for our machine that would take in these positive and negative example strings and be able to determine the original shuffle string.

Our inspiration

This question of finding such an algorithm was inspired by a question from biology: Say we had several different people who all shared some physical trait that we suspected was the result of some genetic sequence in their DNA. Our goal would be to look at the relevant sections of DNA code and find some specific subsequence of nucleotides that they all shared. The problem is that this subsequence might not necessarily be contiguous (i.e. it may not be a substring, where all its characters appear in order without interpolation); that is, other characters may appear between the salient nucleotides as the result of random insertions. Our algorithm would then be able to find the shared subsequence, where interpolation is allowed, among these different DNA strings.

The Problem, rigorously defined and with additional constraints

We here define the original problem statement somewhat more precisely, with more terminology. There is an alphabet of some finite set of characters, such as \{a, b, c, d\}. The alphabet is designated with the symbol \( \Sigma \), and any string of characters of length \( n \) is considered a member of \( \Sigma^n \): a concatenation of length \( n \) of members from \( \Sigma \). A concatenation of two strings \( u \) and \( v \) refers
to a string consisting of the characters of \( u \) in order, followed by the characters of \( v \) in order. Such a concatenated string is denoted \( u \cdot v \), or simply \( uv \).

In the context of machine learning, there is a term called \textit{probably approximately correct (PAC)}, which denotes that a particular algorithm or learning method gives answers that are close to the correct ones (\textquotedblleft approximately\textquotedblright correct), with high probability (\textquotedblleft probably approximately\textquotedblright correct). In mathematical terms, there exist some \( \delta, \varepsilon > 0 \) such that for some correct answer \( x \) and machine\’s answer \( y \):

\[
\text{Pr}(|y - x| > \varepsilon) < \delta
\]

An efficient and PAC algorithm for the general case of shuffle ideals has proved to be fairly computationally complex, so we further constrain the conditions with the following:

1) All input strings are of some fixed length \( n \).
2) The length of the shuffle string is known by the machine.
3) The input strings are drawn from a uniform distribution; that is, each individual character of each input string is selected uniformly at random from the alphabet.

\textbf{Our starting point and goal}

Significant work has already been done on devising a relatively successful algorithm for such a constrained class of shuffle ideals, as in Angluin, Aspnes, Eisenstat and Kontorovich [1]. A version of their proposed algorithm runs as follows:

1) Look at the first characters of all of the positive examples, count the number of occurrences of each, and return the most frequently found character \( u_1 \) as the hypothesis for the first character of the shuffle string.
2) In each positive example, find the first character after the first appearance of \( u_1 \). As in Step 1, tally these occurrences and return the most frequently found character \( u_2 \) as the hypothesis for the second character of the shuffle string.
3) Repeat Step 2 for each of \( u_3, \ldots, u_n \).

Their paper proves that \( u_1u_2\ldots u_n \) is probably approximately equal to the actual shuffle string. We aim to find a way to extend and adapt their algorithm for a broader range of concepts, namely those in which each character in the shuffle string may be designated to be not necessarily a specific character, but any subset of the alphabet. For example, rather than requiring an \( \textquoteleft a\textquoteleft \), followed later by a \( \textquoteleft c\textquoteleft \), and then another \( \textquoteleft c\textquoteleft \), the shuffle string could allow first an \( \textquoteleft a\textquoteleft \) or \( \textquoteleft b\textquoteleft \), then a \( \textquoteleft b\textquoteleft \), \( \textquoteleft c\textquoteleft \), or \( \textquoteleft d\textquoteleft \), and then a \( \textquoteleft c\textquoteleft \). (In relation to the biological inspiration for this problem, this would be analogous to making our algorithm more robust in that it allows single-point mutations.) Kontorovich, Cortes, and Mohri [2] have explored a solution to this problem using support
vector machines, which we are unlikely to use in our approach, but their work may nevertheless serve as some inspiration.

**Our plan and deliverables**

We hope to explore the problem using a combination of conjecture, simulation, and mathematical proof. Deliverables will consist of a paper documenting our approaches and results, as well as any relevant code simulating our proposed algorithms.

**References**