Sasha Arditi Linguistics 5410

Minimal Feature Algorithm Documentation

The formulation of minimal feature sets for phonological inventories is a fundamental linguistic task. A standardized set allows the language to be described at a fundamental level, and substantially reduces phonological rules: in addition, it allows phonological rules of any language to be described.

The minimal feature algorithm described here is different from that in *Chen, Holden*. They seek to find a minimal set of positive and negative features F that can distinguish subset S_s from a phonetic inventory S when $S_S \subseteq S$. Alternatively, this minimal feature algorithm seeks to find a minimal set of features that can be used to distinguish any pair of phonemes p_a,p_b in a phonetic inventory $p_a, p_b \in S$. Nevertheless, it is pertinent to briefly cover certain concepts discussed in *Chen, Holden*, as their minimization problem shares computational difficulties with ours.

An important similarity between these problems is that, at face value, they are not computationally feasible. They are NP-hard problems, requiring assessments of distinctiveness (in theory) for all possible subsets of the total number of features F, requiring assessments of $2^{|F|}$ feature sets. Chen and Holden point out an optimization for their algorithm: Once a feature set of size k which fulfills its task is identified, only sets of less than size k need to be looked at, as one of equal or greater size does not improve our solution. This optimization also works for our minimal feature algorithm.

This optimization does not guarantee feasibility for our algorithm: however, given the scope of this algorithm, concerns about feasibility may be reduced. Chen and Holden's algorithm considers that a universal feature system could have upwards of 400 features, quoting the size of the P-base 3 resource. We will only be concerning the 30-feature excel sheet used in Phonology 5410. This limits the number of features, thus limiting the time complexity of our algorithm($30 \leq |F|$, this the runtime is $\leq 2^{|30|}$). In addition, other optimizations are used in my

code which further minimize the number of features that will be examined: these optimizations generally are able to reduce more features when the inputted phoneme set is smaller, and when the inputted phoneme set is less diverse (ie a set of 10 vowels vs a set of 10 random phonemes from the excel sheet). These optimizations will be discussed in the documentation section, along with the core brute-force algorithm, and an example test case

Documentation

The link: https://github.com/sashera1/minimalFeatureAlgorithm/blob/main/src/Main.java documents the code. The Following is a documentation of the algorithm, by what each major function does: certain functions which are less self explanatory are discussed in additional depth.

processCSV() this function processes the csv (https://github.com/sashera1/minimalFeatureAlgorithm/blob/main/src/FullTest2.csv) and stores its data. The specific set of phonemes we want to find the minimal set for, P is read from the first line of the csv and stored in a list. The next line of the csv is the set of all features from the excel sheet, with the first element empty. All the following lines start with a phoneme describable with the features, and then '+' '-' or '0' for each feature. All features and all phonemes are stored in lists, and all feature values for phonemes are stored in a matrix of length |features| by |all phonemes|. *It is pertinent to remember that different feature values for a phoneme are stored in a row, and values of a feature for different phonemes are stored in a column.*

filterMatrixByPhonemes() this function gets rid of all the rows in the matrix which correspond to a phoneme *not* in the subset of phonemes we want to find the minimal description for. All future references to the phoneme set refer to this subset of phonemes, not all the phonemes in the csv originally in the excel file.

getRidOfZeros() this function makes all 0's in the matrix into -'s instead.

removeUniformColumns() this function removes from the matrix all columns which are all positive or negative. This is an important step in reducing the complexity of our problem

without losing any information. Any column with uniform values cannot provide any distinctive comparison, as any two phonemes in our matrix cannot be distinguished by the feature represented in this column.

removeIdenticalAndInverseColumns() this function removes columns which are identical or inverse to an already existing column in the matrix. *Logic:* Imagine a column A with feature values for phonemes. Imagine column B, identical to column A, cannot provide any new distinctive information, as any comparison which can be made in B can be made in A, and any comparison which cannot be made in B cannot be in A. Imagine column C, for which each feature value is the inverse of that of A. Any comparison between feature values of phonemes [-,-] becomes [+,+], and [+,+] becomes [-,-]: thus any non-distinctive comparison remains non-distinctive. Any comparison [+,-] becomes [-,+] and [-,+] becomes [+,-]: thus any distinctive comparison remains distinctive. So for a column, any identical or inverse columns cannot add any new information. The **invert()** function assists this function by inverting a column: then, when our removal function is comparing two columns to see if they're the same, it can also compare a column to the second column's inverse.

findNecessaryColumns() this function returns a list of "necessary" columns, or features. It iterates through all possible combinations of rows, and for each combination, it checks if there is only a *single* possible way to distinguish these two columns. The logic for this is that, if two phonemes can only be distinguished by a single feature, this feature is absolutely necessary to include in any solution to our program.

findMinimalDescriptiveSet(necessaryColumns) this function is the core of our algorithm. It essentially performs a brute force analysis of all possible subsets of our set of features, checking if the subset is descriptive. This function iterates through the size of the current feature set with a *for* loop, starting at the size of the set of necessary columns. Within this iteration, the method **generateSubsets()**, which it turn relies on the recursive method **generateCombinations()**, returns all subsets of a certain size so that our primary function can

iterate through them in another, nested, for loop, ignoring sets which do not contain all necessary features, and calling the method **isDescriptive(subset)**, (which iterates through all combinations of phonemes, returning true only if all combinations are contrastive with the given set of features). Whenever the isDiscriptive method returns true, the subset is immediately returned, as only subsets of equal or greater size will be iterated through.

Sample Input

Let us input the selected phonemes i,ɪ,e,ɛ,a,ɔ,o,ʊ,u as the first row in the csv file. After removing uniform columns, we reduce our feature set from 30 to 8 (such a large reduction is expected as we inputted only vowels). Removing identical/inverse columns further reduces our feature set to 5. 2 of these features are formulated to be necessary regardless of formulation. Ultimately, when we run our final optimized brute-force algorithm, only 5 subsets are checked for descriptiveness before a correct solution, [high, tense, labial, low], is reached. I tried running this same problem on our algorithm without any of the optimizations *except* for one, the act of going from smaller to larger datasets and ending the search once a descriptive set had been found, and thus required 30,030 subsets to be checked for descriptiveness.

Sources

Chen, Holden, The Computational Complexity of Distinctive Feature Minimization in Phonology

Armik Mirzayan, PhonFeatures-IPA-AM-based-on-sources-v02