# Appendix

## Example to understand the notation

An example of the input data is shown in Table 1, in the form of a matrix *D*. It has six rows (corresponding to regions), denoted here by {*e*1*, e*2*, e*3*, e*4*, e*5*, e*6}. *D* has twelve columns, denoted here by {*f*1*, f*2*, f*3*, f*4*, f*5*, f*6}, in addition to one column for each row. The first six columns correspond to features, e.g., “South East” (which means the region is in South Eastern US) and “Was1\_high” (which means the region was high last week).

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | (South East) | (Mid- Atlantic) | (Was1- high) | (Was1- low) | (Was2- low) | (Was52-high) | (e1) | (e2) | (e3) | (e4) | (e5) | (e6) |
|  | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
|  | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|  | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
|  | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

Table 1: Sample input, represented as a matrix D of dimensions 6 × 12. The rows (corresponding to states) are elements of U, and the first six columns are features. The next six columns are added for computational reasons.

Let *U* = {*e*1*, e*2*, e*3*, e*4*, e*5*, e*6}. In the matrix D in Table 1, *D*2 = {*e*1*, e*4*, e*5} and

*D*5 = {*e*3*, e*4*, e*5} are examples of sets having features *f*2 and *f*5, respectively. Then, *S*(2*,* 5) is a clause, which represents the set *S*(2*,* 5) = *D*2 ∩ *D*5 = {*e*4*, e*5}. Note that the column *D*7 = {*e*1} is also a clause, represented by *S*(7).

We continue our example from Table 1. An example of a target set is the set of regions that have feature , i.e., . can be expressed as combinations of different kinds of clauses. For instance, . This representation has cost. Note that , so can also be expressed simply as ; this has cost . Finally, we can represent the target set as unions and differences of clause as , since , , and .

However, this representation has cost .

Continuing our example, let . Let us restrict ourselves on to the use of features in the clauses to represent in required form. Then, (alternatively, ) covers all elements of with just one extra element . Then, to exactly represent , there isn’t any negative clause to pick.

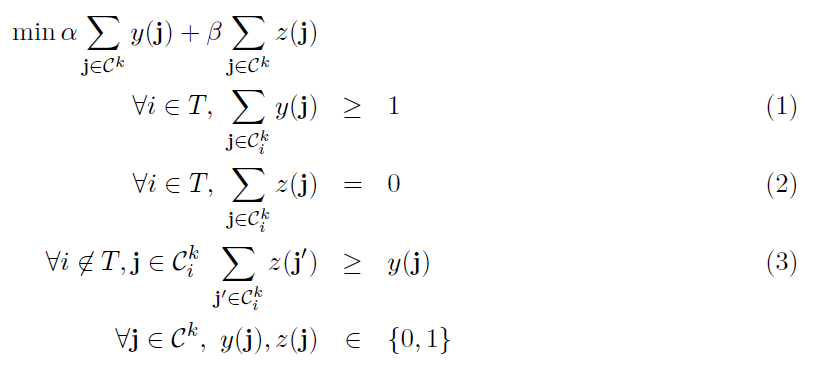
## Integer Program

The problem requires exploring over the space of all possible representations for a set , and choosing one that has the minimum cost. The problem has the additional requirement of ensuring that a large part of is represented. These are both computationally very hard. Formally, these problems are NP-complete, even when (i.e., each clause only has one feature. We refer to [18] for an introduction to this topic. Here, we solve these problems using integer programming, which is a powerful and general technique for solving combinatorial optimization problems. We describe our formulation as an integer program (IP), and how it is solved.

### A. Integer Program (IP) for Problem

We start by specifying the variables and the objective function. Recall the notion of a tuple of features , and a clause . For , let be an indicator variable for being used as a positive clause, i.e., in this case, and 0 otherwise. Similarly, is the indicator variable for being used as a negative clause.

We have the following integer program (IP):



We observe that this is a valid set of inequalities. The constraints (1) indicate that each element must be part of a positive clause. Since is the set of all clauses of size at most , containing element , this constraint implies that at least one of these clauses must be picked as a positive clause, which means must be 1 for some . No element must be part of a negative clause, else it will not be part of the representation. This is captured in the constraints (2), which ensures that for each . Finally, if there exists a clause containing an element , with , the solution must contain a negative clause to “remove” it. Therefore, we need for some . This is captured through constraints (3), which ensures that if for any , for some .

In our work, we focus on , since the descriptions become very complex and hard to interpret otherwise. The data matrix is constructed using the ILI data

for multiple seasons. The space of tuples is constructed from , and the resulting IP is solved using the Gurobi optimization software [19]. For the scale of problems considered here, this solver runs within seconds, and returns the variables which are set to 1. These are then used to construct the representations discussed in the Results.

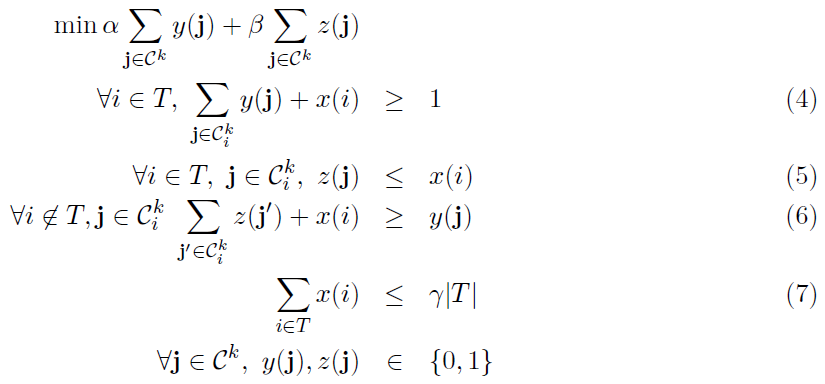
### Appendix B: IP for Problem

Recall that the gives a succinct representation for a set , which is “close” to . We formalize this by requiring that . We solve this by a slight modification of the above IP. Specifically, we have a variable for each element with the following semantics:

(1) if is not represented, i.e., , then , and

(2) if is represented, i.e., , then .

If neither of these conditions are satisfied, we have . Then, measures the difference between and . The following IP solves the problem.



The correctness of the above program follows on the same lines as that for .

The only difference is that if for some , that element does not have to be covered. Therefore, unlike inequality (1), is sufficient now. Similarly, could now be 1 for such an if . Therefore, we have inequalities (5) instead of (2). Finally, for , if it is covered by some , for , it does not have to be accounted for, if ; therefore, inequalities (6) have on the left side, in contrast with (3). We solve the above program in the same way, for different values of , which determines the variables set to 1. These are then used to construct descriptions for the set .