# Report for the Assignment #4(LCS)

## 1. Dataset Information

In this assignment, the same dataset which was previously used in Assignment 1-3 is still being used in this assignment since the output comes from four different algorithms needed to be compared. This dataset contains 34 attributes, and 1 target value need to be classified. And the dataset is trying to explore how there are free electrons in the ionosphere are related with these attribute values. And the target value in this dataset are string values, like ‘b’ represents bad, and ‘g’ represents good.

## 2. LCS Algorithm

Learning classifier system is known as rule-based machine learning method and is also a combined system that combine a discovery component, like the genetic algorithm, and a learning component, like supervised learning, reinforcement learning and etc.

There are several key steps and parts in the LCS algorithm, and the details are as below:

- Environment: the environment is the source of the data that the LCS is going to learn from, it can be offline or online. Here in this assignment, it is the dataset mentioned in the first part, and I separated the shuffled dataset into a 4:1 part and use 4 to train and use 1 to test.
- Rule/Population: the rule in the LCS algorithm is like a relationship between the attributes values and the final prediction, which can be though as if state then predict, and the rule is typically represented as #1000#####1->1, here, 1 means the feature needs to equal to be 1, 0 means needs to be 0, # means either value for that attribute is fine. In this assignment, the 34 attributes are also represented in this way, but 0 here

means the attribute value is negative, 1 means the attribute value is positive and # means either positive or negative is fine. Then the rule can be represented as like 000####111##...00011###'b', where the last bit represents the target value.

- Matching: the matching step is the most important one in the LCS algorithm, it compares the value of instances between the rule and the training data, but it doesn't compare the target value, just the attributes. In other word, during the matching process, for each specific rule, go through all the train data, count the total number of matches for a specific rule, if there is no match for a rule, discard that rule. For example, matching means the for every position of a rule, if the corresponding index in the dataset is 0, it should have either 0 or #, if the corresponding index in the dataset is 1, it should have either 1 or #. All the rules that have matching are put in a set that names RulesSet.
- Covering: when there are no rules in the RulesSet, the covering process is very crucial that it generates new rules that at least has one match, and then add it to the RulesSet.
- Learning: In this process, the accuracy of rules in the RulesSet is calculated, and it is counted by correct#/match# of that rule.
- Subsumption: as previously said in the rule part, the rules can be represented as a combination of 0, 1 and #, where # means either value of 0 or 1 is fine. Therefore, there would be rules that like ####0 and 0000 are both in the rules set. If both accuracy of these rules are accepted, it is good to merge these two classifiers together. This can only happen we there is a rule is much more general that covers all the cases in the other rule and have similar accuracy.

- Genetic Algorithm: in this assignment, I have also implemented the crossover part and mutation part from the GA algorithm. For the cross-over part, two rules from the rules set is chosen from the rules set, and by generate one random index location, the left part from parent1 is combined with the right part from parent2. And the target value depends on the fitness score of parent1 and parent2. For the mutation part, select one random rule from the rule set, and generate a random index, if the current value in that index is either '0' or '1', mutated it to '#', if '#' mutated it to '0' or '1' based on whether it is even or odd. The reason why mutates to the '#' is to increase the generosity.
- Deletion: after fully explored and trained for several generations, need to clean the rules set, like discard rules that have accuracy less than certain value or merge rules that redundant together.
- After cycle through the above steps repeatedly, use the finalized rules set to predict the test set and get the accuracy value. And the test is unseen during the test step.

## 3. Implementation details

- Environment part:

```
# Read the dataset file and generate the list format, and separate them to
train set and test set
def readFile(path):
    df = pd.read_csv(path)
    dataset = df.iloc[:, :]
    dataset = dataset.values.tolist()
    random.shuffle(dataset)
    test = dataset[0::5]
    train = dataset[1::5] + dataset[2::5] + dataset[3::5] + dataset[4::5]
    #train = dataset[:]
    #test = train
    return train, test
```

In the code above, the dataset set is being separated into the 4:1 part and 4 used as training one and 1 used as the testing one

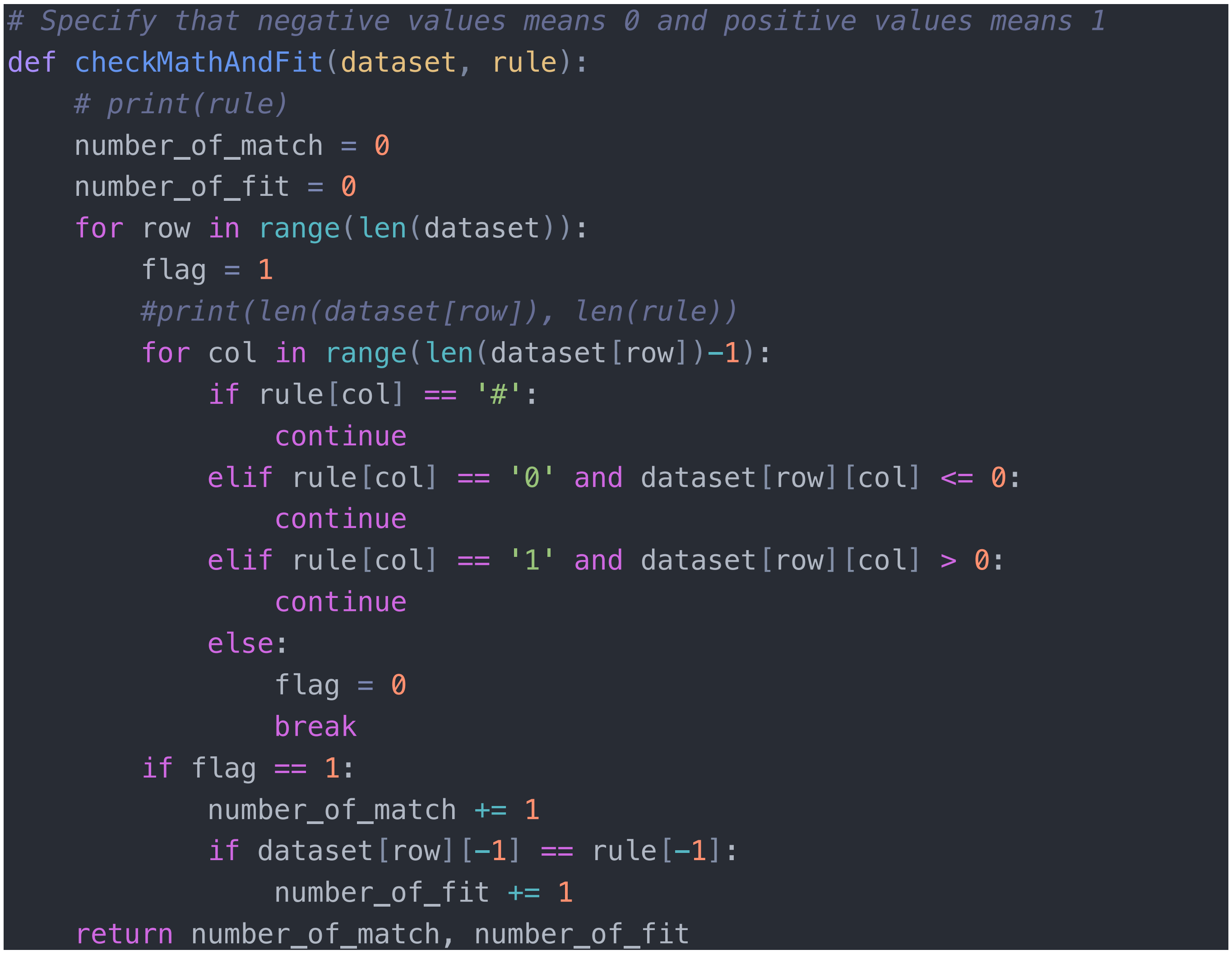
- Rule population:

```
# Based on the ramdon assignment of 0 ,1 and #, generate the rule
def generateRule(zeros, ones, wildcards):
    new_rule = ['0'] * zeros + ['1'] * ones + ['#'] * wildcards
    random.shuffle(new_rule)
    flag = random.randint(0, 4)
    if flag % 2 == 0:
        new_rule.append('b')
    else:
        new_rule.append('g')
    return new_rule
```

```
# Generate a random rules set of size 20
def generateRulesSet(number_of_instance):
    Rules = set()
    while True:
        if len(Rules) < 20:
            zeros = random.randint(0, 10)
            ones = random.randint(0, 10)
            wildcards = number_of_instance - zeros - ones
            if wildcards < 0:
                continue
            else:
                rule = generateRule(zeros, ones, wildcards)
                rule = tuple(rule)
                if rule not in Rules:
                    Rules.add(rule)
        else:
            break
    return Rules
```

In the code above, 20 initial population rules are generated, and for each rule generated, random number of 0, 1 and # is generated and used to create the rule. And the rule is shuffled in order to increase the randomness at the beginning.

- Matching:



```python
# Specify that negative values means 0 and positive values means 1
def checkMathAndFit(dataset, rule):
    # print(rule)
    number_of_match = 0
    number_of_fit = 0
    for row in range(len(dataset)):
        flag = 1
        #print(len(dataset[row]), len(rule))
        for col in range(len(dataset[row])-1):
            if rule[col] == '#':
                continue
            elif rule[col] == '0' and dataset[row][col] <= 0:
                continue
            elif rule[col] == '1' and dataset[row][col] > 0:
                continue
            else:
                flag = 0
                break
        if flag == 1:
            number_of_match += 1
            if dataset[row][-1] == rule[-1]:
                number_of_fit += 1
    return number_of_match, number_of_fit
```

In the code above, for each rule, go through the training set and find the number of matching and correct predicting in those matches. The returned values are stored in the local set and dictionary.

- Covering:

```
# Every generation, add some rules to the rules set
    def addRulesToSet(number_of_instance):
        if len(rules) >= 17:
            return
        temp = generateRulesSet(number_of_instance)
        for rule in temp:
            if rule not in rules:
                m_t, f_t = checkMathAndFit(dataset, rule)
                if m_t != 0 and f_t / m_t > 0.3:
                    rules.add(rule)
                    rules_fitness[rule] = [m_t, f_t / m_t]
        if len(rules) < 17:
            addRulesToSet(number_of_instance)
```

Like mentioned in the previous part, when there are not enough rules at the initial, use this function to generate more rules and add to the rules set. And make sure all the added rules have # of corrected predict / # of match > 0.3.

- Genetic Algorithm:

```
# Every generation, choose one rule in the rules set to
mutate
    def mutationStep():
        temp = list(rules)
        if len(temp) < 1:
            return
        index = random.randint(0, len(temp)-1)
        cur_rule = list(temp[index])
        loc = random.randint(0, len(cur_rule)-1)
        if cur_rule[loc] == '0' or cur_rule[loc] == '1':
            cur_rule[loc] = '#'
        elif cur_rule[loc] == '#' and loc % 2 == 0:
            cur_rule[loc] = '0'
        elif cur_rule[loc] == '#' and loc % 2 == 1:
            cur_rule[loc] = '1'
        if tuple(cur_rule) not in rules:
            m_t, f_t = checkMathAndFit(dataset, tuple(cur_rule))
            if m_t != 0:
                rules.add(tuple(cur_rule))
                rules_fitness[tuple(cur_rule)] = [m_t, f_t / m_t]
                if cur_rule[-1] == 'b':
                    return
                min_fit = f_t
                min_elem = tuple(cur_rule)
                for k, v in rules_fitness.items():
                    if v[1] * v[0] < min_fit and k[-1] != 'b':
                        min_elem = k
                        min_fit = v[1] * v[0]
                rules_fitness.pop(min_elem)
                rules.remove(min_elem)

    # Every generation, choose the best two rules in the rules set to crossover and generate the child
    def crossoverStep():
        temp = list(rules)
        if len(temp) < 2:
            return
        parent1 = random.randint(0, len(temp)-1)
        parent2 = random.randint(0, len(temp)-1)
        while parent2 == parent1:
            parent2 = random.randint(0, len(temp)-1)
        par1 = list(temp[parent1])
        par2 = list(temp[parent2])
        child = []
        index = random.randint(0, len(par1)-1)
        child = par1[:index] + par2[index:-1]
        '''
        for index in range(len(par1)-1):
            if par1[index] == '#' or par2[index] == '#':
                child.append('#')
            elif index % 2 == 0:
                child.append(par1[index])
            else:
                child.append(par2[index])
        '''
        fit_par1 = rules_fitness[tuple(list(rules)[parent1])][1]
        fit_par2 = rules_fitness[tuple(list(rules)[parent2])][1]
        if fit_par1 > fit_par2:
            child.append(par1[-1])
            # rules_fitness.pop(tuple(list(rules)[parent2]))
            # rules.remove(tuple(list(rules)[parent2]))
        else:
            child.append(par2[-1])
            # rules_fitness.pop(tuple(list(rules)[parent1]))
            # rules.remove(tuple(list(rules)[parent1]))
        if tuple(child) not in rules:
            m_t, f_t = checkMathAndFit(dataset, tuple(child))
            if m_t != 0:
                rules.add(tuple(child))
                print(f_t/m_t)
                rules_fitness[tuple(child)] = [m_t, f_t / m_t]

                if child[-1] == 'b':
                    return

                min_fit = f_t
                min_elem = tuple(child)
                for k, v in rules_fitness.items():
                    if v[1] * v[0] < min_fit and k[-1] != 'b':
                        min_elem = k
                        min_fit = v[1] * v[0]
                rules_fitness.pop(min_elem)
                rules.remove(min_elem)
```

As mentioned in the previous part, in both the mutation part and the crossover part, the parent is being random chosen from the rules set and the mutated parent is being random chosen from the rules set. And for the cross over index and the mutated index, the location is also being randomly generated. After generating the new child, add it to the rules set and remove the one with lowest fitness.

- Subsumption and Deletion:

```
# Check the subset of a string
    def isSub(x, y):
        for index in range(len(x)):
            if x[index] == y[index]:
                continue
            elif y[index] == '#':
                continue
            else:
                return False
        return True

    # Remove the redunt or over-fitting rules from the
rules set
    def cleanRule():

        temp = set()
        for k, v in rules_fitness.items():
            if v[1] < 0.8 and k[-1] == 'g':
                temp.add(k)
            if v[1] * v[0] < 50 and k[-1] == 'b':
                temp.add(k)
            if v[1] < 0.85 and v[0] > 50:
                temp.add(k)
        for rule in temp:
            rules_fitness.pop(rule)
            rules.remove(rule)

        cur = list(rules)
        D = dict()
        for i in range(len(cur)-1):
            for j in range(i+1, len(cur)):
                if isSub(cur[i], cur[j]):
                    # print("aaaaaa")
                    if cur[j] in D:
                        D[cur[j]].append(cur[i])
                    else:
                        D[cur[j]] = [cur[i]]
                elif isSub(cur[j], cur[i]):
                    # print("bbbbbb")
                    if cur[i] in D:
                        D[cur[i]].append(cur[j])
                    else:
                        D[cur[i]] = [cur[j]]
                else:
                    # print("hhhhhh")
                    if cur[i] not in D:
                        D[cur[i]] = []
                    if cur[j] not in D:
                        D[cur[j]] = []
        New_rules = set()
        for k, v in rules_fitness.items():
            if k in D:
                New_rules.add(k)
        New_fitness_rules = dict()
        for k in D:
            # print(k)
            New_fitness_rules[k] = rules_fitness[k]
        return New_rules, New_fitness_rules
```

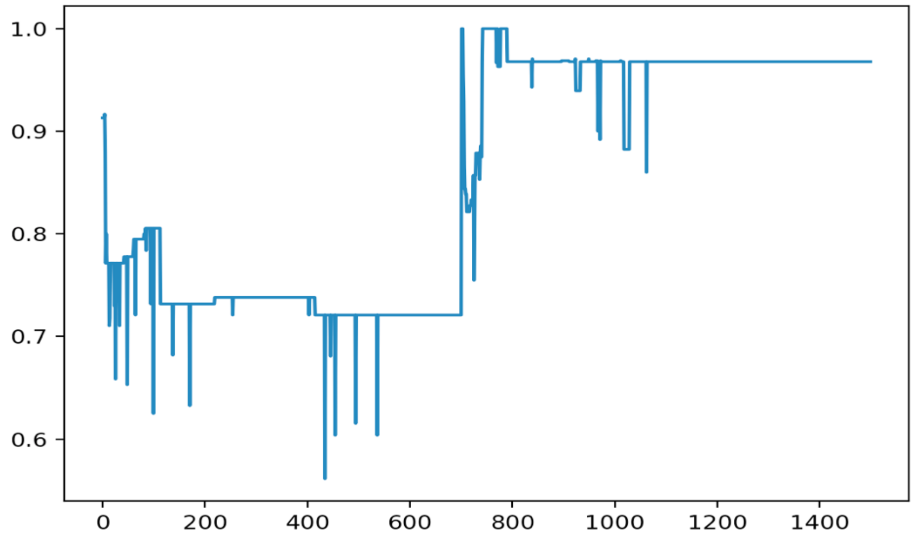
In this dataset, there are 4:1 of target value of ‘g’ and ‘b’, therefore, the restriction of the checking condition for the rules with target ‘g’ and ‘b’ is a little different. And for the subsumption, every two rules in the rules set are being chosen to test whether they are a general format of the other, if yes, the subset is being discarded and only the superset is being kept.

- Overall, the steps mentioned above are cycled for 1500 generations, and for each generation, when the size of rules set is less than 10, call the update function to add more random rules that have at least 0.3 fitness. And when the cycles is about half through, start the deletion and subsumption process for every generation. The reason why choose to start these two steps lately is because all the rules at beginning are randomly generated which would have large error for the value of the assignment and the number of target value of ‘g’ and ‘b’ has large difference and the target value for all the rules I generated are randomly assigned. In each generation, use current rules set to predict the accuracy of the test set, and save them for later plot.

## 4. Output of the self-created LCS algorithm

```
{('#', '#', '1', '#', '#', '#', '1', '1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '1', '#', '#', '#', '#', '#', 'g'): [107, 0.8691588785046729], ('#', '#',
'1', '#', '1', '#', '1', '1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '1', '#', '#', '#', '#', '#', 'g'): [106, 0.8773584905660378], ('1', '#', '1', '#', '1',
'#', '1', '1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', 'g'): [128, 0.90625], ('#', '#', '1', '#', '#', '#', '1', '1', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'g'): [135, 0.8592592592592593], ('#', '#', '#', '#', '1', '#', '1', '1', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '1', '#', '#', '#', '#', '#', 'g'): [106,
```

**0.8773584905660378], ('#', '#', '#', '#', '1', '#', '1', '1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', 'g'): [134, 0.8656716417910447], ('1', '#', '#', '#', '1', '#', '1', '1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', 'g'): [128, 0.90625], ('#', '#', '1', '#', '1', '#', '1', '1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', 'g'): [133, 0.8721804511278195], ('1', '#', '#', '#', '1', '#', '1', '1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '1', '#', '#', '#', '#', '#', 'g'): [102, 0.9117647058823529], ('1', '#', '1', '#', '#', '#', '1', '1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', 'g'): [130, 0.8923076923076924]}**



The above picture is the (# of corrected prediction / # of matched case in the test case) in each generation, it is clear that it at first is high since at the beginning, the size of rules set is not enough for my set of limit, and the update rules are being performed, all the rules being added are having this fitness higher than 0.3, which makes it is big at initial. And then it drops, dur to the crossover and mutation steps are performed, which may lead to a bad fitness, but the number of matches for the rules in the rule set is increasing. After about 700, where I start the deletion process and subsumtion process, the accuracy is increasing suddenly since many rules that similar to each other are merged together and fitness that fail my set of limitation is being dropped. There are several specific reasons:

- Number of matched sample of rules in the test set is dropping, this is because some rules with frequency less than my set is being discarded, like there are only about 70 data that has target value of ‘b’, and the sampling and shuffling process makes the training sample for target value of ‘b’ become less. In addition, due to the randomness of generating rules, it makes the finding of correct and strong rules for predicting ‘b’ needs much more generations.
- The value of these 34 instances are all between -1 and 1, in this assignment, I simply separated them by negative and positive. But it should be better to use specific way for each attribute, like some attributes have just 0 or 1, some have negative and positive but prefer one much more than the other, therefore, separate the region into more detailed pieces, like 1 represent positive to 0001 means >0.2, >0.4, >0.6 and >0.8.
- LCS is not really good for this dataset, since this dataset is really all numeric value and is not balanced, most datasets are preferred to get a target value of ‘g’, even use all ‘#’ for the rules attributes and ‘g’ for the target value would get about 67% accuracy.

## 5. Comparison with the output from GA, ES, PSO and Backpropagation

Compare the result with the Backpropagation, the output of the Backpropagation is much better than the LCS algorithm, there are several reasons:

1. Backpropagation would compute the gradient of the loss function with respect to the weights of the network for a single output. It is very efficient in minimize the loss of Neural Networks and it always gives the right direction of changing. And in LCS, only several rules are being generated, number of attributes in the rules would also affect its performance. For this dataset, number of rules in the LCS and the way used to represent the rule is highly related with the performance.

Compare the result with the PSO, the output of PSO is much better than the LCS algorithm, the reasons are:

1. The PSO also use the Neural Networks to give weight for each attribute to the Neural Network, it is less efficient than the Backpropagation but since it still uses the global optimal solution and local optimal solution as a guide, it still leads the particle to a relative correct direction to the optimal solution.
2. There is no mutation and crossover in the PSO algorithm, which would not add complexity to the population.
3. In the LCS in this assignment, each attribute is being given a direct value, like 0 for negative and 1 for positive, which makes each attribute seems binary, but this dataset is not suit for using this representation. Cut the range of intervals into more pieces would increase its performance, like the decision tree.

Compare the result with the GA and ES, the output of LCS is similar to those two, reasons are:

1. All these three algorithms contain the strategy and thinking of mutation and crossover steps, and in case of sticking in the local optimal solution, not all the parents and children can be survival to the next generation due to the selection strategy.

2. For the crossover part, even two parents with best fitness are chosen to perform 1-point crossover, the result doesn't permit to have a good fitness value. Especially, in this LCS algorithm, when two rules are crossover, their target value may even not be the same, assign child based on the fitness is also meaningless, like Rule ##001111'g' with fitness 0.9 and Rule 1111##00'b' with fitness o.6 would not make ##00##00 to predict 'g' more.
3. For the mutation part, this would make rules set forget its optimal solution and sometimes guide a wrong direction and make this exploration no effort.

Overall for this assignment with LCS algorithm, there are several things can be further discussed:

1. Try to find a classification problem with binary value for each attribute with almost balanced numbers with LCS algorithm to check whether it is just because this dataset is not good for the LCS algorithm.
2. As mentioned several times before, try to create a unique way of represent each attribute based on its real condition, like attribute 1 and 2 in this dataset only have 1 or 0, and others have value from -1 to 1, and different attributes have different weight concentration in certain numeric range. Therefore, using 1 bit to represent a class attribute is not precise enough.