CS 539 Machine Learning

Project 7 - Genetic Algorithms
Chris Winsor

The Dataset

This experiment uses the Mushroom dataset from UC Irvine Center for Machine Learning Repository, available at http://archive.ics.uci.edu/ml/datasets/Mushroom

The dataset characterizes mushrooms with respect to whether they are edible or poisonous. There are 8124 samples. The 22 attributes describe observable characteristics of the mushrooms, for example cap, stem, population and habitat. All attributes are nominal unordered (categorical). There is a single class variable <edible, poisonous>

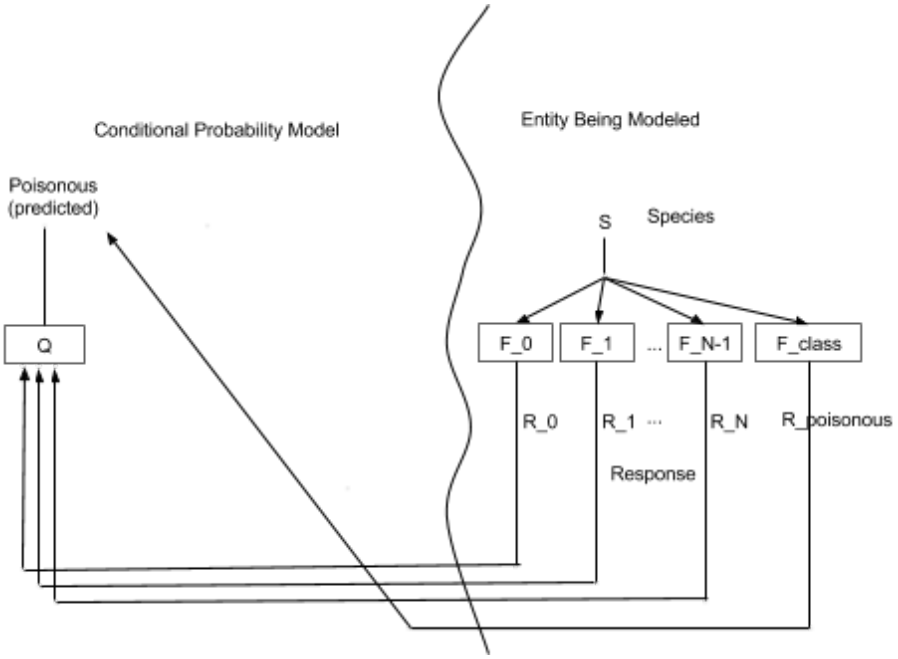
Guiding Questions

- Can we predict whether a mushroom is poisonous based on its observed attributes ?
- Are certain combinations of characteristics that are more apt to indicate that a mushroom is poisonous ? Can the mutation of a Genetic Algorithm search discover these easily ?

Code Overview

In this project we used Weka, and leveraged the Naive Bayes Network. An overview is shown in the Figure below. At the top level are two elements:

- Device Being Modeled
- Model (Naive Bayes Net using Genetic Algorithm Search)



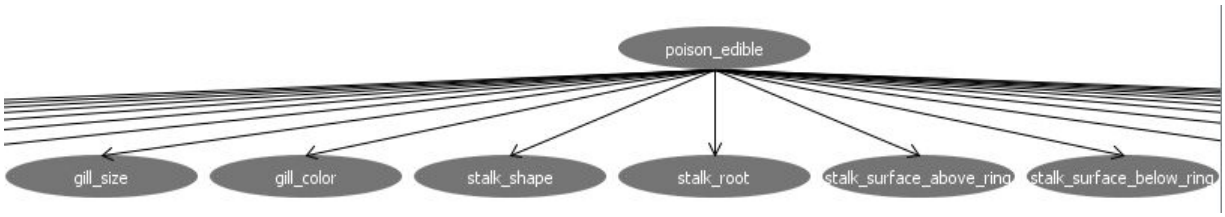
Entity Being Modeled

The Entity Being Modeled has been captured in the dataset. In the context of this paper we assume

- the Device Being Modeled is causal and stateless. Given stimulus S the Entity Being Modeled outputs response R_1, R_2 ... R_N and R_class in a deterministic manner based on functions F_1, F_2, ... F_N and F_class.
- There is a single class variable R_poisonous which must be predicted.

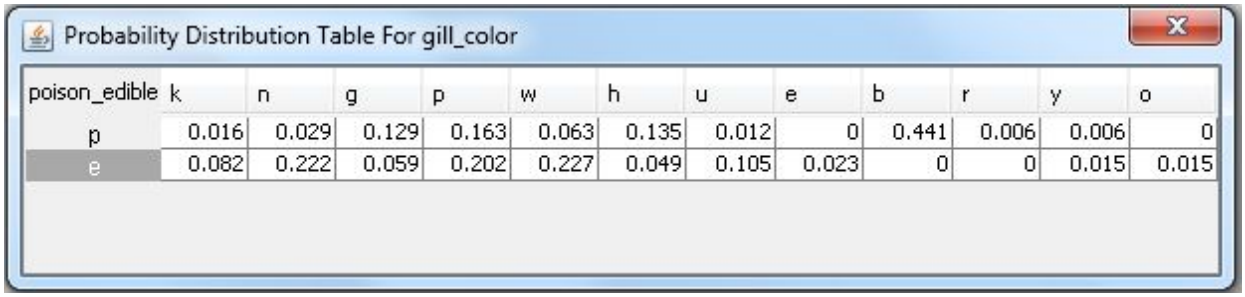
Conditional Probability Model (Q)

A statistical network with structure of a Naive Bayes Network is used to implement the Conditional Probability Model. The purpose of the Conditional Probability Model is to predict R_poisonous given R_1, R_2... R_N.

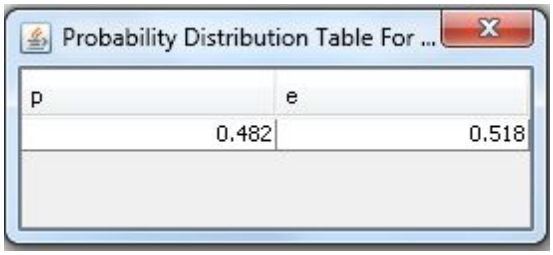


DNA for the Conditional Probability Model

The Conditional Probability Model has joint distribution table for each node, and a prior distribution probabilities for the root node. These joint and prior distribution tables constitute the DNA of the Individual. Values for these tables are established using the Genetic Algorithm search.



poison_edible	k	n	g	p	w	h	u	e	b	r	y	o
p	0.016	0.029	0.129	0.163	0.063	0.135	0.012	0	0.441	0.006	0.006	0
e	0.082	0.222	0.059	0.202	0.227	0.049	0.105	0.023	0	0	0.015	0.015



p	e
0.482	0.518

Individual

An Individual consists of 22 joint distribution tables and one prior probability table. These are captured in the Individual class and form the DNA for the Genetic Algorithm.

```
/**
 * Individual class
 * This class represents a single individual in the population
 * An individual has
 * one prior probability table (m_ClassDistribution)
 * and a set of joint probability tables (m_AttributeDistributions)
 *
 */
public class Individual {

    /** name for the individual */
    String name;

    /** probablity tables for attribute, class */
    protected DiscreteEstimatorWithAccess [][] m_AttributeDistributions;
    protected DiscreteEstimatorWithAccess m_ClassDistribution;
```

Population

The Population class establishes the Individuals that constitutes the population. It also establishes parameters for the search and has the buildClassifier() method which performs the search itself.

```
/**
 * Population class
 * This class represents the population of individuals
 *
 * Parameters include
 * the number of individuals, number of donors,
 * percentage of mutation and crossover to be applied
 * thresholds for fitness and maximum loops
 *
 * Methods include buildClassifier() which builds the
 * population and runs the Genetic Algorithm search
 *
 */
public class Population extends AbstractClassifier  {

    public static final int NUM_INDIVIDUALS   = 500;
    public static final int NUM_KEEPERS       =  10;
    public static final int PERCENT_MUTATION  =  30;
    public static final int PERCENT_CROSSOVER =  30;
    public static final int FITNESS_THRESHOLD =  98;
    public static final int MAX_LOOPS = 3000;

    /** Individuals that constitute the population */
    Individual [] individual = new Individual[NUM_INDIVIDUALS];

    /** fitness tracking (per-individual) */
    float [] fitness = new float [NUM_INDIVIDUALS];
```

Genetic Search Procedure

The buildClassifier() method performs the Genetic Algorithm search. Individuals are created with an initial random set of distributions. Then a loop is entered whereby mutation and crossover are selectively applied with the fittest individuals being the donors and less fit individuals the recipients.

```
    /**
     * Method buildClassifier
     * This method initializes the population and run the Genetic Search
     */
    public void buildClassifier(Instances instances) throws Exception {

        // Copy the instances
        m_Instances = new Instances(instances);

        // create the individuals
        for (int i=0; i<NUM_INDIVIDUALS; i++) {
            // create the individual
            Individual anIndividual = new Individual("individualName",instances);
            anIndividual.createNewRandomDistributions();
            individualsList.add(anIndividual);
        }

        // loop until we achieve fitness goal, or reach loop limit
        float maxFitness = 0;
        int loop = 0;
        Random randomGenerator = new Random();
        while ((maxFitness < FITNESS_THRESHOLD) && (loop < MAX_LOOPS)) {
            System.out.printf("--- loop %d ---\n",loop);

            // randomly apply crossover or mutation
            for (int recipient=NUM_KEEPERS; recipient<NUM_INDIVIDUALS; recipient++) {

                // crossover
                if (randomGenerator.nextInt(100) <= PERCENT_CROSSOVER) {
                    int donor = randomGenerator.nextInt(NUM_KEEPERS);
                    individualsList.get(recipient).crossover(individualsList.get(donor));
                }

                // mutation
                if (randomGenerator.nextInt(100) <= PERCENT_MUTATION) {
                    individualsList.get(recipient).mutate();
                }
            }

            // measure fitness
            for (int i=0; i<NUM_INDIVIDUALS; i++) {
                float tempF;
                tempF =  individualsList.get(i).fitness(instances);
                if (tempF > maxFitness) {
                    maxFitness = tempF;
                    System.out.printf("loop %d maxFitness %f from individual %d\n",
                            loop,maxFitness,i);
                }
            }

            // sort individuals
            Collections.sort(individualsList, COMPARTOR_INDIVIDUALS);
            loop++;
        }
    }
```

Fitness:

Fitness is measured by having the individual predict the test dataset. Fitness is a method of the Individual class. It measures the number of correctly predicted instances.

```
/**
 * fitness
 * Calculates the overall fitness of an individual, given a set of instances
 */
public int fitness(Instances instances) throws Exception {
    int numCorrect = 0;

    // Compute counts
    Enumeration enumInsts = instances.enumerateInstances();
    while (enumInsts.hasMoreElements()) {

        Instance instance = (Instance) enumInsts.nextElement();
        double [] result = distributionForInstanceFirstHalf(instance);

        // compare
        if (((result[0]<=result[1]) && (instance.classValue() == 0)) ||
            ((result[0]> result[1]) && (instance.classValue() == 1))) {
            numCorrect++;
        }
    }
    // squirrel it away
    historicalFitness = numCorrect;
    return (numCorrect);
}
```

Cross-Over and Mutation

Cross-Over and Mutation are methods provided using an extension to the Estimator class.
Mutation randomly chooses a symbol table from the Estimator and replaces it with random weights.
Cross-over takes as input a 'donor' Estimator and copies the symbol weights from the donor to its own table.

```
public class DiscreteEstimatorWithAccess extends DiscreteEstimatorMyCopy {

    /**
     * mutation
     * randomly choose one of the symbols and assign it a random value
     */
    public void mutate() {
        Random randomGenerator = new Random();
        int randomSymbol = randomGenerator.nextInt(m_Counts.length);
        double randomWeight = randomGenerator.nextInt(100);

        setSymbol(randomSymbol,randomWeight);
    }

    /**
     * crossover
     * given a donor distribution, copy its symbol weights to this distribution
     */
    public void crossover(DiscreteEstimatorWithAccess donar) {
        for (int i=0; i<m_Counts.length; i++) {
            setSymbol(i,donar.getCount(i));
        }
    }
```

Termination

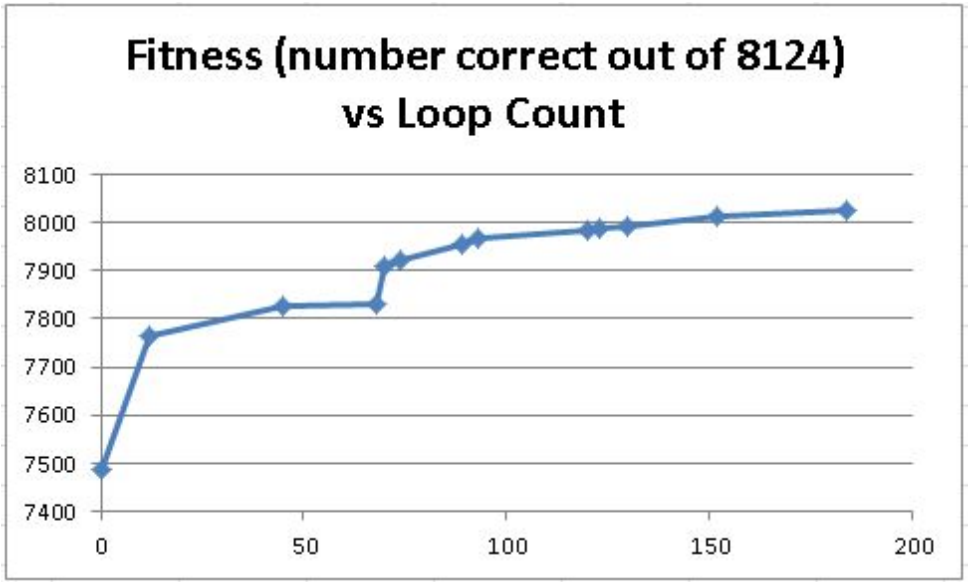
Termination is achieved if the Genetic Search reaches a pre-specified fitness threshold, or a maximum specified loop count is reached.

Summary

A run on the full mushroom dataset was performed using the following parameter setttings:

```
public static final int NUM_INDIVIDUALS   =    500;
public static final int NUM_KEEPERS       =     10;
public static final int PERCENT_MUTATION  =     80;
public static final int PERCENT_CROSSOVER =     80;
public static final int FITNESS_THRESHOLD =  10000;
public static final int MAX_LOOPS = 10000;
```

The results were as follows:



The maximum theoretical fitness is 8124 (the number of instances in the dataset).

An observation is that improvements appear to come in clusters. For example around loop #70 there were 3 improvements to performance which came quickly together. It is suspected a mutation resulted in one individual with a higher fitness, and that individual subsequently had descendents with this new 'gene' but which then contributed additional enhancements. A similar 'cluster' of improvements occur around loop #130.