Implementation and Testing of Classifier-Guided Sampling Algorithm

Dau Ku

NC State University

Table of Contents

Introduction

The goal of this project is to implement Classifier-Guided Sampling (CGS) method in solving engineering optimization problems (Backlund, Shahan, and Seepersad, 2014). The CGS method uses the knowledge/guidance of machine learning classifiers to sample the potential good designs and to find the optimum design using the least amount of expensive evaluations. In this project, we implemented the CGS method with various Bayesian network classifiers on the two optimization problems.

Bayesian Network

In the CGS method, classifiers play the most important role. The performance of the classifier will decide whether the result will converge or not. According to Backlund, Shahan, and Seepersad (2014), they proposed using Multinomial Naïve Bayesian as the classifier for the CGS method. In this part, we will elaborate more details about Bayesian network.

- Introduction of Bayes' theorem:
  In statistics, Bayes' theorem describes the probability of an event based on some prior knowledge.

$$f_{\Theta|X}(\theta|x) = \frac{f(x_1, \dots, x_n|\theta) * f_\Theta(\theta)}{\int f_{X,\Theta}(x,\theta)\, d\theta} \dots (1)$$

$$f_{\Theta|X}(\theta|x) \propto f(x_1, \dots, x_n|\theta) * f_\Theta(\theta) \dots (2)$$

  - $f_{\Theta|X}(\theta|x)$ is often called Posterior Distribution. In the context of twenty-item problem, $f_{\Theta|X}(\theta|x)$ means the probability of being a good design given the selected items (design attributes), which is exactly the result given by the prediction results given by the trained classifier.
  - $f(x_1, \dots, x_n|\theta)$ is often called likelihood. Likelihood is a probability of getting the observed data. In the context of the twenty-item problem, likelihood means the probability of getting the design attributes, selected items, given the class of the design.
  - $f_\Theta(\theta)$ is often called Prior Distribution. In the context of twenty-item problem, prior is the prior knowledge about the probability of being a good design.

- $\int f_{X,\Theta}(x,\theta)\, d\theta$ is often called normalizing constant, which ensure the function is a probability density function. Since the normalizing constant doesn't depend on the parameter θ, we can write the Bayes' theorem as equation (4): posterior is proportion to likelihood*prior.

- Naïve Bayesian Network:

  naïve Bayesian network is a special case of Bayesian network. The naïve part means that naïve Bayesian network assume each data point only depends on the parameter of prior distribution and each data point is independent with each other. This assumption of independence makes the calculation of likelihoods become every simple.

$$Under\ the\ assumption\ of\ independence:$$

$$f(x_1, \ldots, x_n|\theta) = \prod_{i=1}^{n} f(x_i|\theta) \ldots (3)$$

  If we assume each X are independent with each other, then the likelihood can be calculated as the product of the function of Xs given the prior. The figure. 1 shows the structure of naïve Bayesian network.

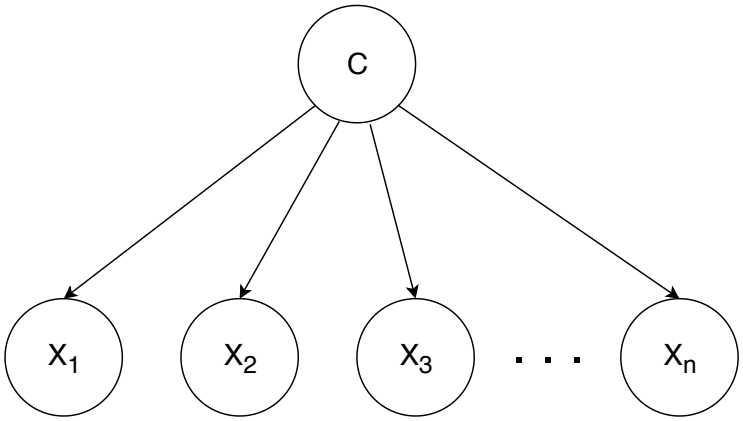

Figure 1. Naive Bayes

The classifier-guided sampling method

a. Prepare design candidates

Before implementing the CGS method, we start with preparing our design candidates and formulate our objective function based on the types of the problem. In this project, we are only solving problems with discrete design variables, therefore, we always begin the CGS method by

creating the design space; generating all the possible combinations of design variables. Once the design space is created, we can sample the potential good designs according to the guidance of the classifier.

b. First iteration

1. First expensive evaluation:

   The first iteration of the CGS method begins with randomly selecting Ntr (number of training points in the initial training set) numbers of design points and evaluating their objective function values.

2. Assign Labels:

   In the study of Backlund, Shahan, and Seepersad (2014), they suggested using Bayesian Network as the classifier of the CGS method. One advantage of choosing Bayesian Network is that Bayesian Network gives us the posterior probability of being a good design given the prior; we then not only can label a design as good or bad for the purpose of machine learning, but also these probabilities also help us to bin the cheap points, predicted results by the classifier, in the later CGS step. We assign labels based on the initial class threshold, Tc0. For maximization problems, we assign a design as a good design if its objective function value is greater than Tc0, whereas we assign a design as a good design if its objective function value is less than Tc0 in minimization problems. Here, Tc0 is set such that only 5% of the designs in the initial expensive evaluation pool can be assigned as a good design.

3. Classifier training and predicting:

   We train our Bayesian Network classifier with the labeled designs and their corresponding design attributes. Once the classifier is trained, we predict the labels for all the designs whose objective function values are not evaluated yet in the design space. These labels predicted by the classifier are called cheap points, which is the essence that makes CGS method shine: these cheap points generated by the classifier will guide us to only sample the candidates that have high probability of being a good design so we only spend computational powers on the potential good designs.

c. Sequential loops until meet the termination criterial

The process of sequential iterations is fundamentally no difference with the first iteration, however, in order to rapidly converge to the global optimum, Backlund, Shahan, and Seepersad

(2014) proposed some modification regarding how to assign labels and how to select designs for sequential expensive evaluations.

1. Binning cheap points:

   The first step of the loop is to sample the designs that we will evaluate in the current iteration from the cheap points we generated from the first iteration. However, here we don't randomly sample designs like we did in the first iteration. Instead, we will utilize the information provided by the classifier. Specifically, we want to sample the designs that are classified as good by the classifier. Therefore, we categorize the cheap points into three categories: 1. High-class high-certain points (HH points) 2. Low-class high-certain points (LH points) 3. Uncertain points. We categorize a cheap point as a high class with high certainty if that point is classified as a good design with the posterior probability greater than 0.6. Likewise, we categorize a cheap point as a low class with high certainty if that point is classified as a bad design with the posterior probability greater 0.6. Lastly, we categorize an uncertain point if both probabilities of being good or bad for that design are less than or equal to 0.6.

2. Sample Ns designs for the expensive evaluation:

   Unlike the first iteration, we can sample a different amount of designs to evaluate in the sequential iterations. Here, Ns is the number of points to be sampled during each iteration. Selecting Ns number of points, ideally, we want to sample as many HH points as possible, since HH points are more likely to be a good design according to the classifier. However, there are two reasons we can't always do that. Firstly, we don't always have enough HH points in the pool, especially during the first couple of iterations, since we only feed very few good designs to the classifier, so the classifier tends to classify everything as a bad design in the early stage of the algorithm. Secondly, in order to increase the accuracy of the classifier gradually, we also need to select some uncertain points to feed the classifier, so the classifier can explore more the design space. Hence, we need to introduce another parameter: Phs, percentage of high-class high-certainty points in Ns. Phs controls how many percentages of Ns need to be HH points. Consequentially, there will be total three scenario in sampling Ns designs from cheap points:

   a. The sum of numbers of HH points and uncertain points is less than Ns: this is the only scenario we will sample from LH points, since it's not enough even we include all the HH points and the uncertain points, so we need to fill the rest with LH point.
   b. The sum of numbers of HH points and uncertain points is greater than Ns while the number of HH points is less than Ns*Phs: in this case, although the sum of numbers of HH points and uncertain points is greater than Ns, the number of HH points still doesn't meet Ns*Phs, therefore, we select all HH points and fill the rest with uncertain point.
   c. The sum of numbers of HH points and uncertain points is greater than Ns while the number of HH points is larger than Ns*Phs: here, we have the number of HH points more than the total numbers of HH points we will select in that iteration, therefore, we randomly sample Ns*Phs numbers of HH points from all the HH points and fill the rest with uncertain points.

3. Expensive evaluations in the sequential iterations:
   Like what we did in the first iteration, we evaluate the objective function value of each point in the sampled pool and record the best optimum objective function value in this iteration.
4. Assign labels in the sequential iterations:
   Here, we are also going to assign the label based on Tc like what we did in the first iteration. However, instead of using the same Tc to assign the label, we will update Tc according to the performance of the algorithm. As we mentioned before, making some parameters flexible during the procedure improves the performance of the algorithm and Tc is one of the flexible parameters. The update rules for Tc are as follows:
   a. If number of HH points is greater than Ns, we should make Tc more stringent. Therefore, we will increase (for maximization problems) or decrease (for minimization problems) the Tc for the next iteration by half the difference between the current class threshold and the current best known solution: $T_{c,j+1} = T_{c,j} - 0.5(T_{c,j} - f^*), where\ f^*\ is\ the\ current\ best\ known\ solution$

b. If number of HH points is less than Ns, we should make Tc more tolerant. Therefore, we will decrease (for maximization problems) or increase (for minimization problems) the Tc for the next iteration by 5% of the difference between the current class threshold and the initial class threshold:

$$T_{c,j+1} = T_{c,j} + 0.5\left(T_{c,0} - T_{c,i}\right), where\ T_{c,0}\ is\ the\ initial\ class\ threshold$$

5. Retrain the classifier and generate cheap points:

Again, we train the classifier with designs' labels and their design variables. Notice, we have total Ntr + Ns numbers of data that we have evaluated the objective function values at this point, so be sure to train the model using all the data we have evaluated. Likewise, as the procedure iterates, we will have more and more evaluated data to feed the classifier, so the classifier should theoretically become more accurate as the iteration goes on. After we retrained the classifier, we will use it to predict the label for those designs whose objective function values are not yet evaluated. Once the labels are predicted, the new batch of cheap points are generated and then we can go back to the step one to start a new loop until we met the termination criteria.

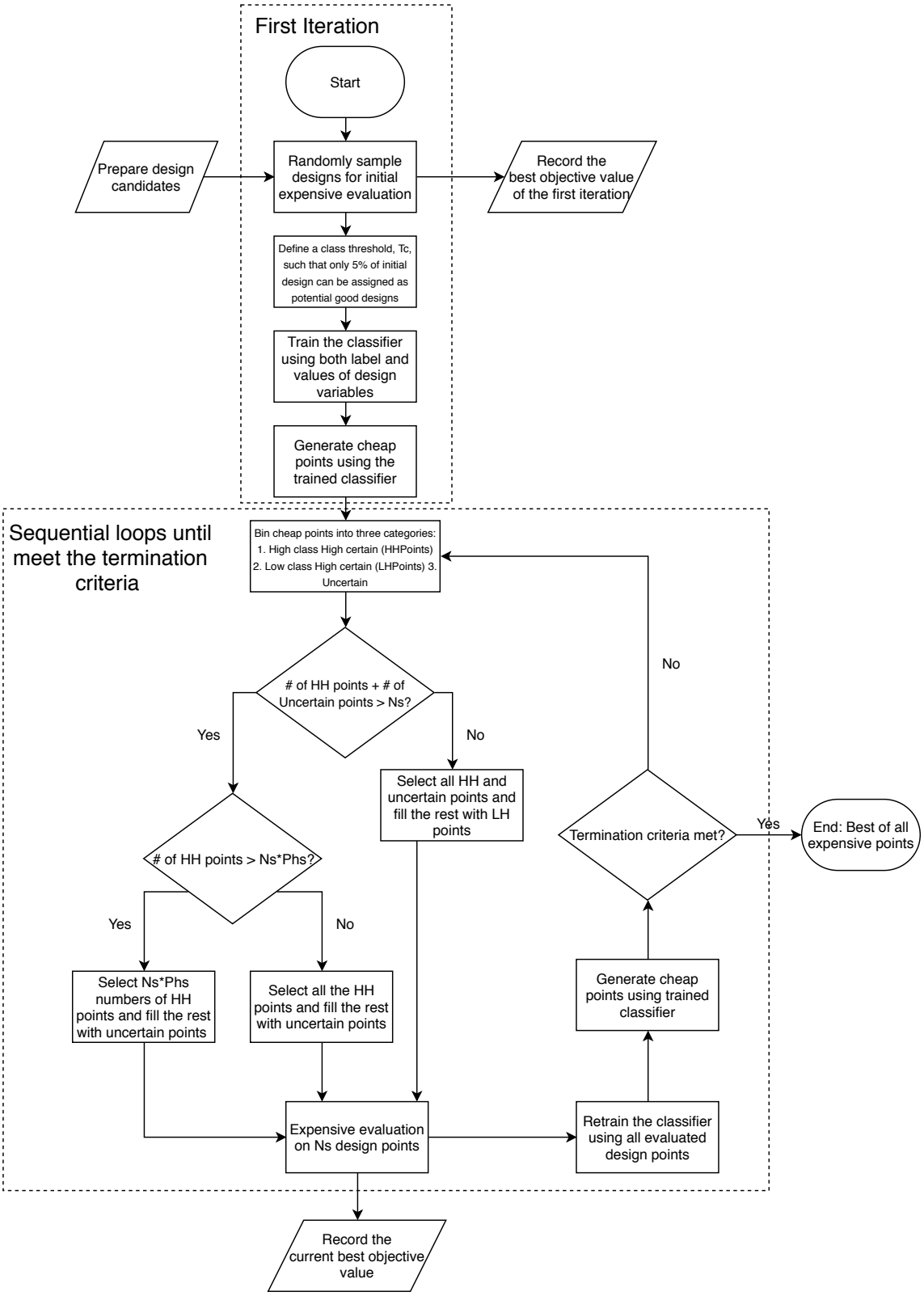

Figure 2, Classifier-guided sampling method

Testing problem1: Twenty-item Knapsack problem

a. Problem introduction

The objective of the knspsack problem is to select items from total 20 items that will maximize the combined value while the total weight doesn't exceed a certain limit. Table 1 shows the weight and the value for each item (Backlund, Shahan, and Seepersad, 2014).

Table 1. Twenty-item problem parameters

Item	Weight (w_i)	Value (v_i)
1	94	3
2	70	41
3	90	22
4	97	30
5	54	45
6	31	99
7	82	75
8	97	76
9	1	79
10	58	77
11	96	41
12	96	98
13	87	31
14	53	28
15	62	58
16	89	32
17	68	99
18	58	48
19	81	20
20	83	3

Therefore, the problem can be formulated as:

$$Maximize\ V(x) = \sum_{i=1}^{n} v_i x_i, \qquad x_i \in (0,1) \ \ldots(4)$$

$$Subject\ to \sum_{i=1}^{n} w_i x_i \ \leq W, \qquad x_i \in (0,1) \ldots(5)$$

b. Parameters

For this particular problem, we set our parameter as the follows: Ntr = 100, Ns = 50, and Phs = 0.9

c. Procedure

We totally ran the CGS algorithm 50 times so we can have enough data to evaluate the average performance. Each execution of the CGS algorithm has 1200 times of objective function evaluation and then we average over the 50 best function value for each iteration.

d. Results and comparison

- Multinomial naïve Bayesian:

  According to Backlund, Shahan, and Seepersad (2014), they proposed using multinomial Naïve Bayesian as the classifier for the CGS method.

  - Algorithm performance:

    Figure 3. shows the performance of the CGS method using multinomial naïve Bayes as the classifier. In this figure, the blue line is plotted by connecting the averages of best objective function value at each iteration and the intervals represent the 10% percentile and the 90% percentile. As we can see, even though multinomial naïve Bayesian is suggested by Backlund, Shahan, and Seepersad (2014), the algorithm not only never reached the true global optimum but also getting further away from the global optimum.

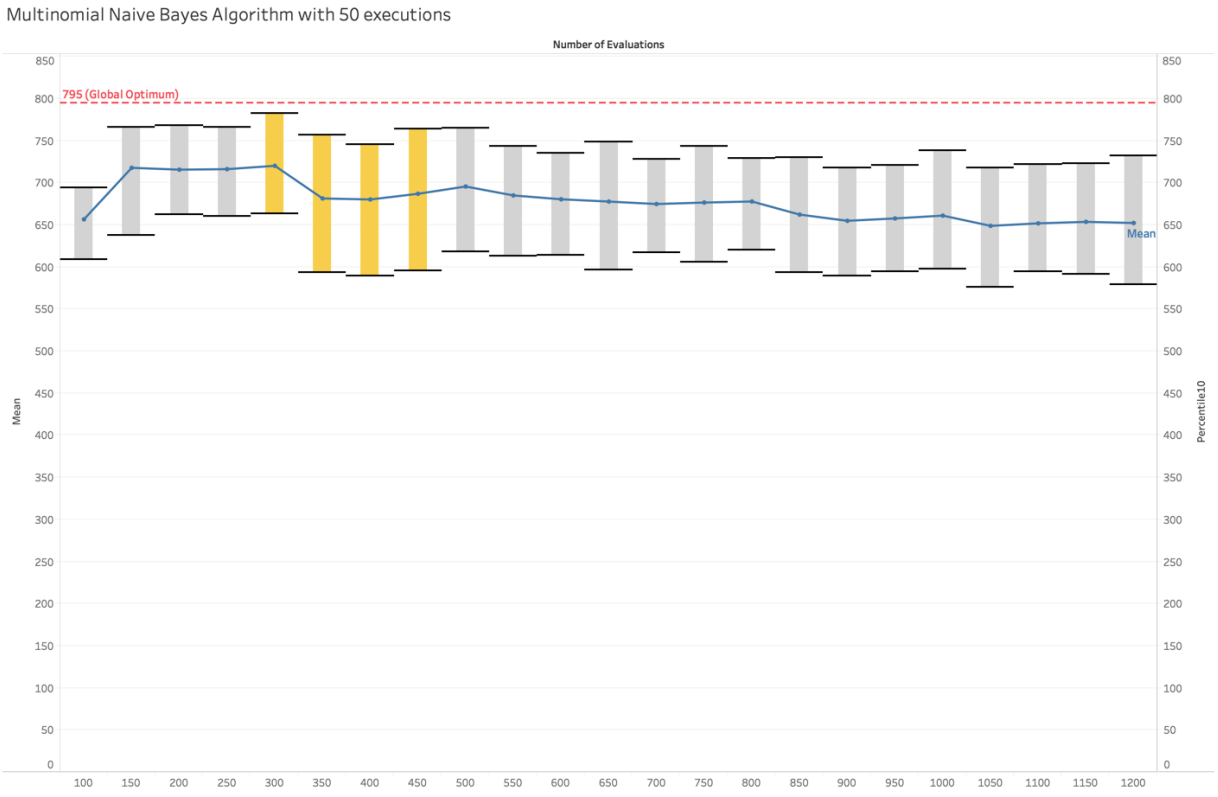

Figure 3. Twenty-item problem result by multinomial naïve Bayes

  - Classifier performance:

    As we mentioned above, the CGS algorithm relies heavily on the guidance provided by the classifier, so we believe the poor performance of the algorithm is

due to the inaccuracy of the multinomial naïve Bayesian classifier. Therefore, we then evaluated the performance of the classifier based on the true positive rate, which is often called recall score. The reason why we use recall score instead of the overall accuracy is due to the nature of the CGS algorithm: in the CGS algorithm, our data is always imbalanced since we always allow only few designs to be considered a good design, for example, we set the Tc0 in the way that only 5% of the design in the initial pool can be assigned to be good, so inevitably the majority of the data we feed into the classifier are bad designs which cause the classifier tends to classify everything as bad designs. Therefore, the right metrics we should be paying attention to is how likely the classifier can identify the good designs correctly. To evaluate the performance of the classifier we implemented 5-folds cross-validation to calculate the average recall score at each iteration. Figure 4 shows the averages of recall score at each iteration and the overall recall score. As we can see each iteration has a pretty low recall score and the overall average is only 0.1815. This is the evidence that the multinomial naïve Bayesian classifier is not suitable for this problem.

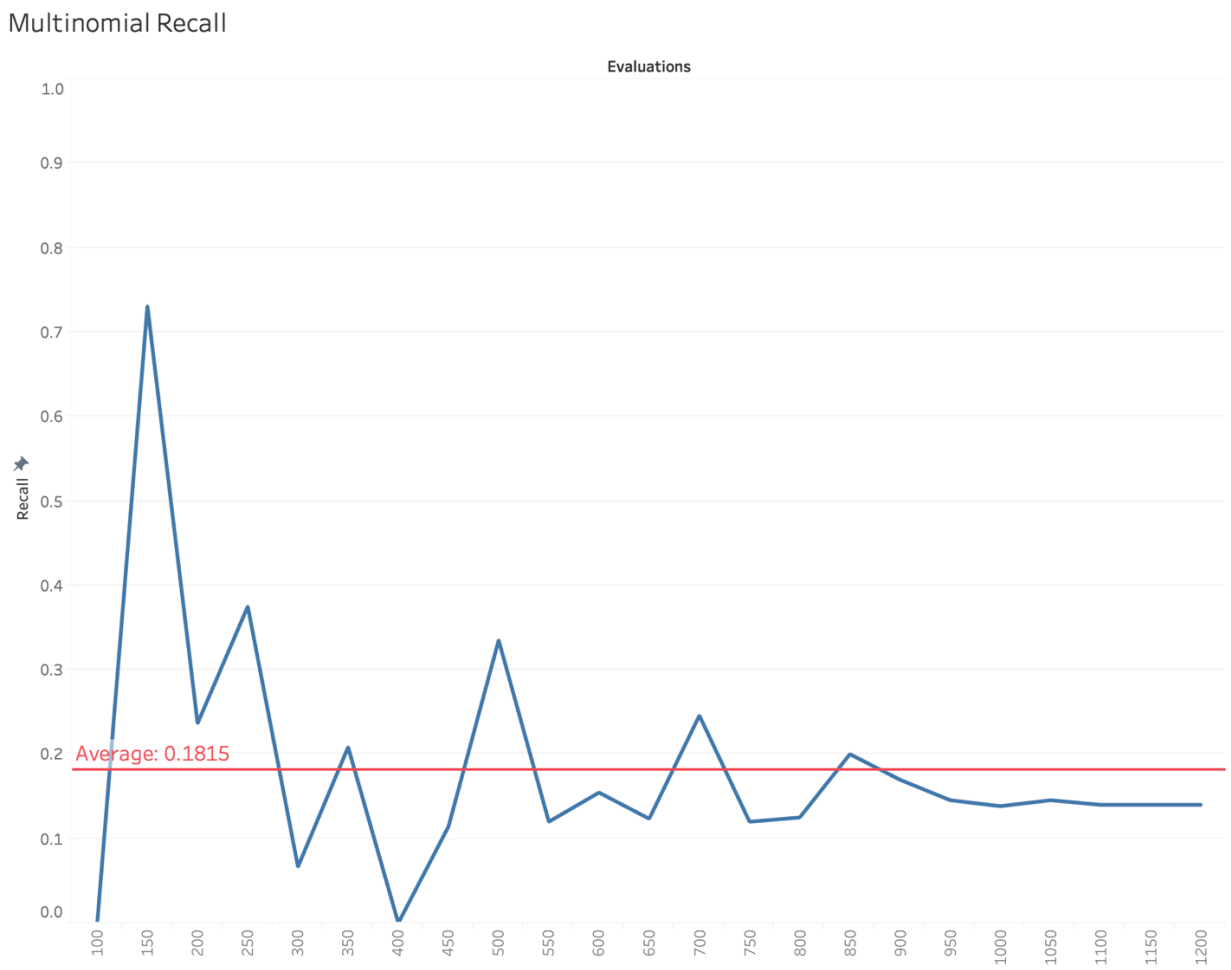

Figure 4. Recall scores using multinomial naïve Bayes

- Bernoulli naïve Bayesian:

  After knowing that the multinomial naïve Bayes doesn't work well with the twenty-item problem, we noticed that instead of multinomial distribution, our data follows a Bernoulli distribution; each design attribute represents including or excluding a particular item, which means each design variable can only take either 1 or 0. Therefore, we are sure that each of the design variable follows a Bernoulli distribution. According to the documentation of naïve Bayes in Scikit-Learn of Python, Bernoulli naïve Bayes classifier is suitable for the data such that there are multiple features however each one is assumed to be a binary-valued variable. This description matches our data perfectly, so we decided to switch to Bernoulli naïve Bayesian classifier.

    - Algorithm performance:

Figure 5 shows the performance of the CGS method using Bernoulli naïve Bayes as the classifier. From the figure 5 we can see that the algorithm reaches the global optimum at 300 function evaluations and gradually moved away from the optimum. We might be confused by this phenomenon of moving away from the optimum but remember how we construct the CGS algorithm. We sample the potential designs based on the guidance of the classifier and evaluate the sampled design, however, once a design is selected and evaluated, we will exclude them during the sampling process. Therefore, once those good designs are selected based on the guidance of the classifier, the rest of the designs are not as good as them so we will see the result moving away from the optimum.

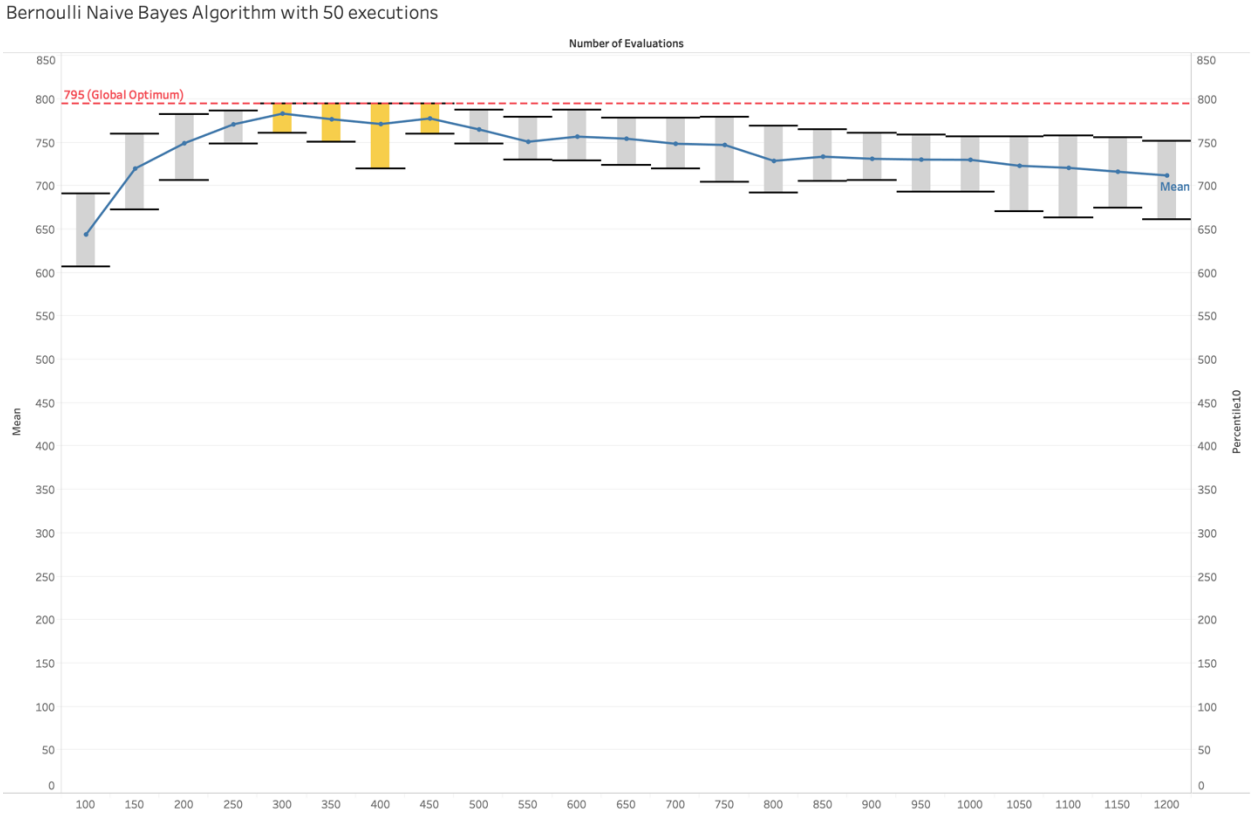

Figure 5. Twenty-item problem result by Bernoulli naïve Bayes

- Classifier performance:

  Although we have found the global optimum using Bernoulli naïve Bayesian as the classifier, we can still double check the recall score to see whether Bernoulli truly performed better than multinomial in this problem. From the Figure 6 we see that despite the first couple of iterations, the recall scores are between 30 to 40%

and the overall average is 0.2778, which is far better than the multinomial case. Therefore, we can conclude that Bernoulli naïve Bayesian works better for the twenty-item optimization problem.

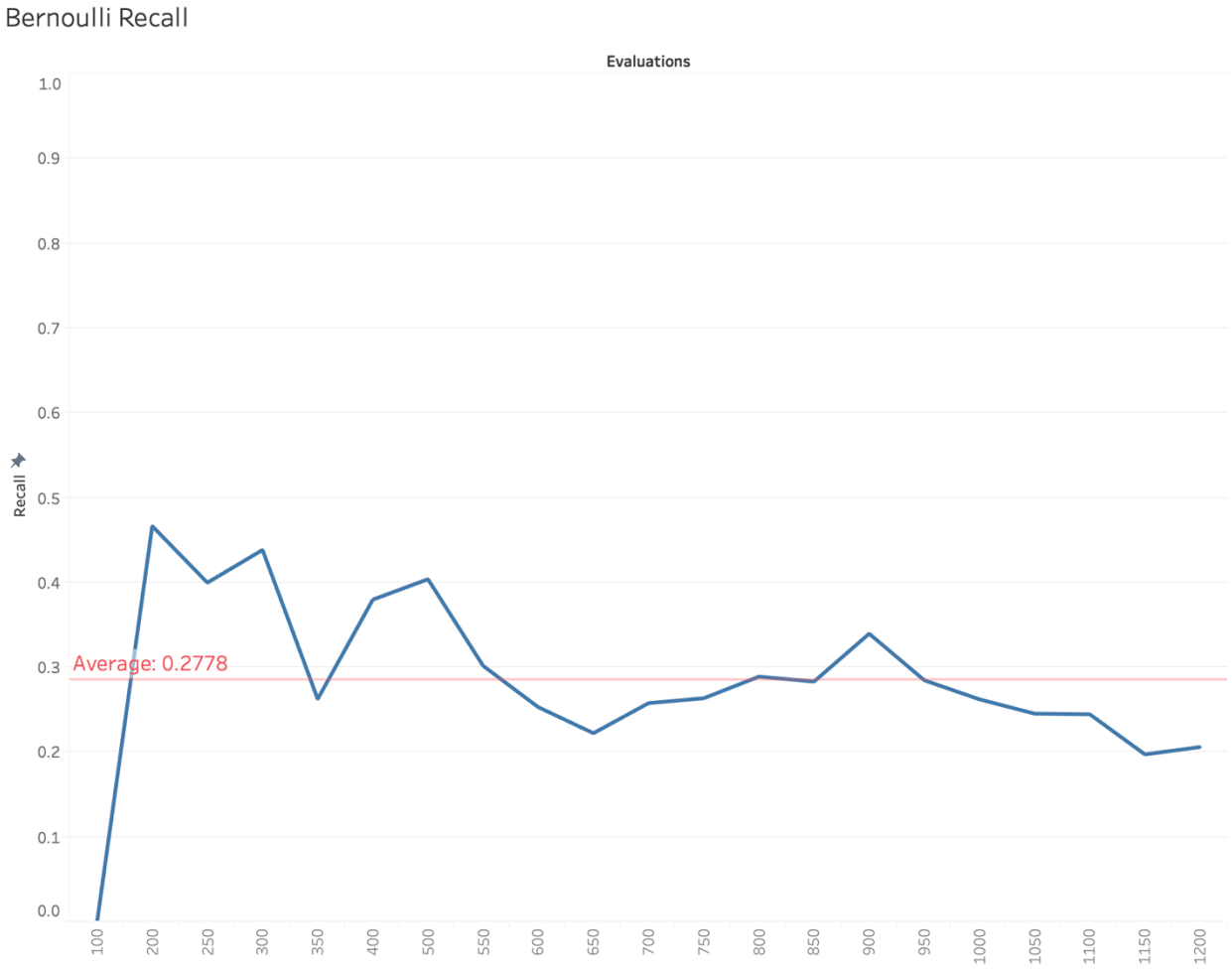

Figure 6. Recall scores using Bernoullil naïve Bayes

Testing problem 2: Eleven-city traveling salesperson problem (TSP)

a. Problem introduction

The objective of the TSP problem is to find the shortest possible tour that visits each city exactly once and return to the city of origin. The eleven cities' coordinates are given in Table 2 (Backlund, Shahan, and Seepersad, 2014).

Table 2. Coordinates for eleven cities

City	Abscissa	Ordinate
Origin	14.8	42.7
1	98.6	29.8
2	5.0	4.5
3	39.9	12.0
4	28.0	22.6
5	57.0	79.2
6	26.2	90.7
7	67.9	55.1
8	76.0	47.1
9	86.4	67.5
10	47.3	98.5

The traveling salesperson will begin at the city of origin and travel to every other city exactly once and return to the city of origin. Since the starting city and the ending city are both the origin city, the rest of possible permutation is exactly: $10! = 3628800$. Therefore, we are going to implement the CGS algorithm to find the shortest route from all the 3628800 possible routes.

The problem can be formulated as:

$$Minimize\ D(x) = d_0(x_1) + \sum_{i=1}^{n-2} d(x_i, x_{i+1}) + d_0(x_{10}) \ldots (6)$$

Where $d_0(x_1)$ is the Euclidean distance from the city of origin to the first city

$d(x_i, x_{i+1})$ is the Euclidean distance between cities x_i and x_{i+1}

$d_0(x_{10})$ is the Euclidean distance from the city of origin to the last city

b. Parameters

For this particular problem, we set our parameter as the follows: Ntr = 100, Ns = 100, and Phs = 0.5

c. Procedure

Again, we ran the CGS algorithm 50 time to evaluate the average performance. However, according to Backlund, Shahan, and Seepersad (2014), TPS problem is more complicated than the twenty-items problem, so for each execution of the CGS algorithm we evaluate total 8000 objective functions.

d. Results and comparison

The two classifiers we used in the testing problem 1 are both based on the assumption of independence. However, this is not the case in TSP optimization problem; there are high dependence between each design variables, since we design routes such that each city can only be visited exactly once. Therefore, Backlund, Shahan, and Seepersad (2014) proposed using augmented naïve Bayesian classifier and the structure is showed in Figure 7.

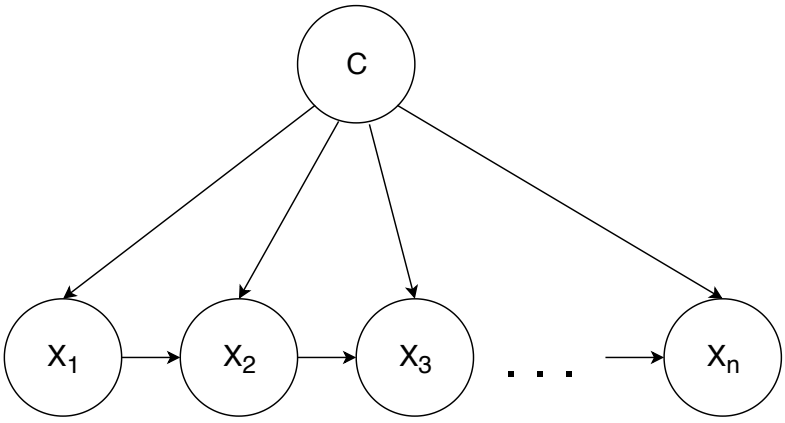

Figure 7. Augmented naïve Bayes

In augmented naïve Bayes, each design variable not only depends on the prior anymore. Instead, each design variable depends on the prior and the previous design variable. This dependence complicates the calculation of the likelihood in Bayesian theorem and, unfortunately, there are

very few packages that support the augmented naïve Bayes. Therefore, we still use the classifiers in Naïve Bayes family for this problem.

- Multinomial naïve Bayesian:
    - Algorithm performance:

      From Figure 8 we can tell that the CGS algorithm based on multinomial naïve Bayesian classifier is roughly a random search; the result never converges toward the global optimum.

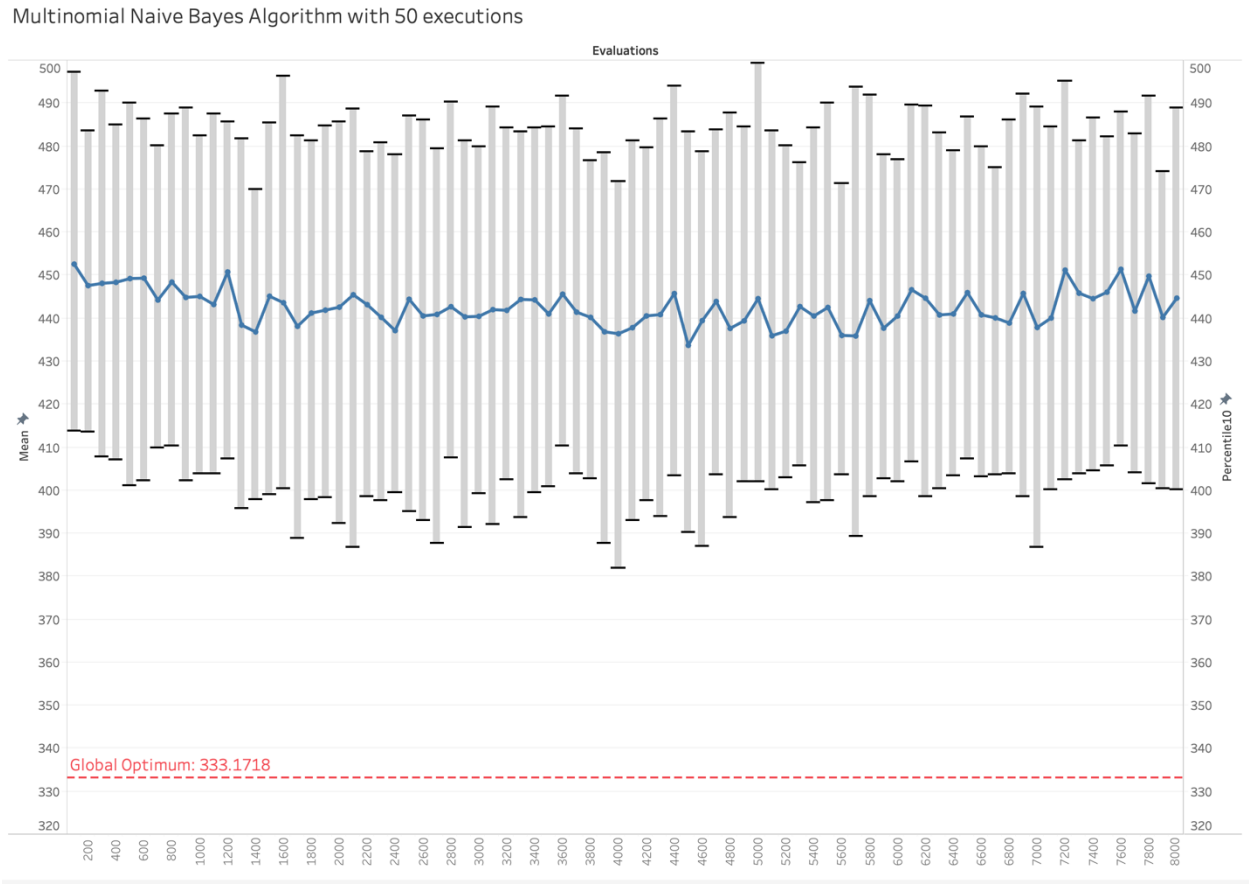

Figure 8. TSP result by Multinomial naïve Bayesian

    - Classifier performance:

      Here we can examine the accuracy of the multinomial naïve Bayesian classifier for this problem. We know that the algorithm never converges to the optimum, so we can expect the recall score for this case is very low. From the Figure 9 we see a zero-recall score at any iteration. This shows that the algorithm is indeed randomly picking designs.

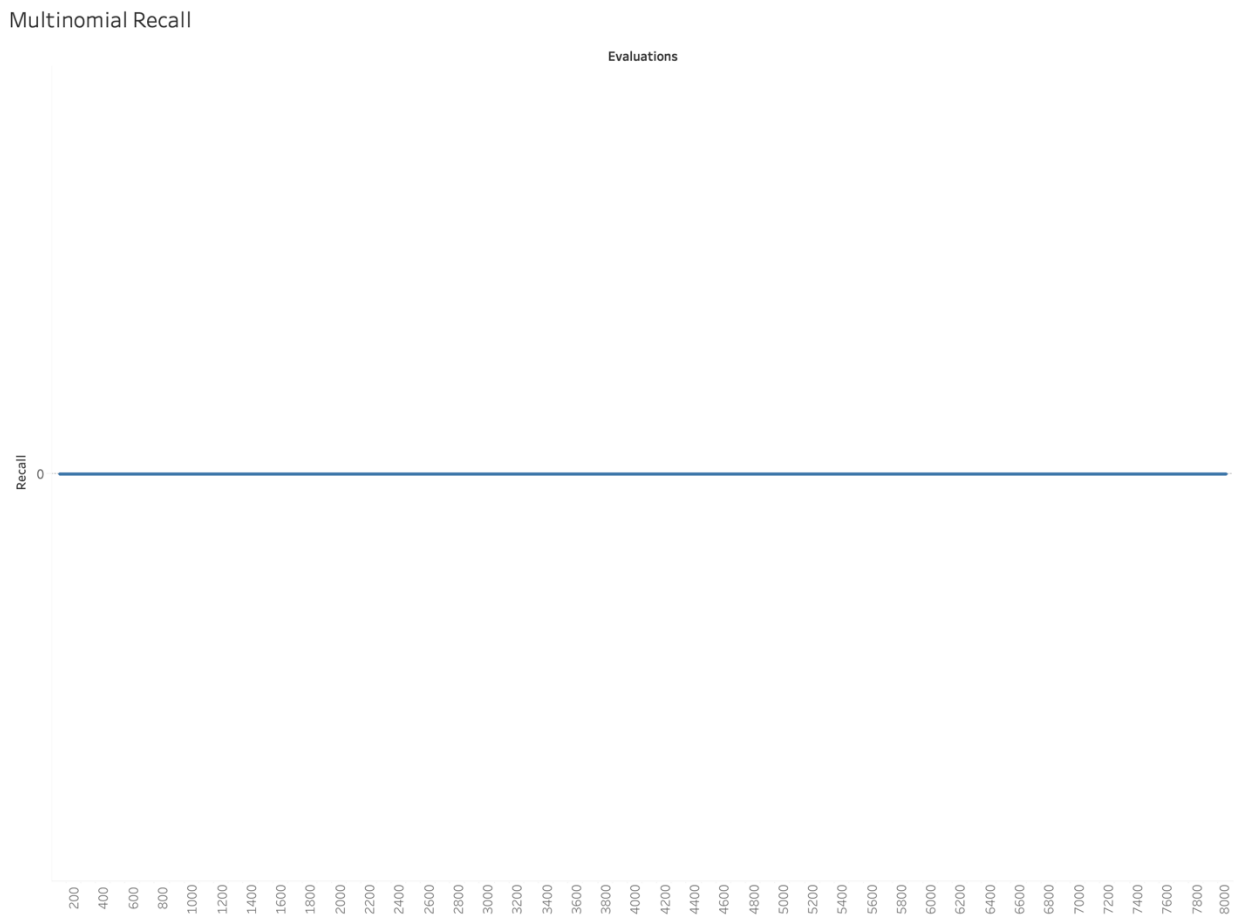

Figure 9. Recall score using multinomial naïve Bayesian

- Complement naïve Bayesian:

Here we tried to solve the problem using another classifier in naïve Bayes family: complement naïve Bayesian. According to the documentation of naïve Bayes in Python Scikit-Learn package, the complement naïve Bayesian classifier works well in the unbalanced problem. As we mentioned above, we will always feed the classifier with unbalanced data, hence, we will show the performance of the complement naïve Bayesian in this part.

  - Algorithm performance:

    From Figure 10 we can see that the performance of the CGS algorithm is slightly better than using multinomial naïve Bayesian, although the algorithm still failed to reach the optimum within 8000 function evaluations.

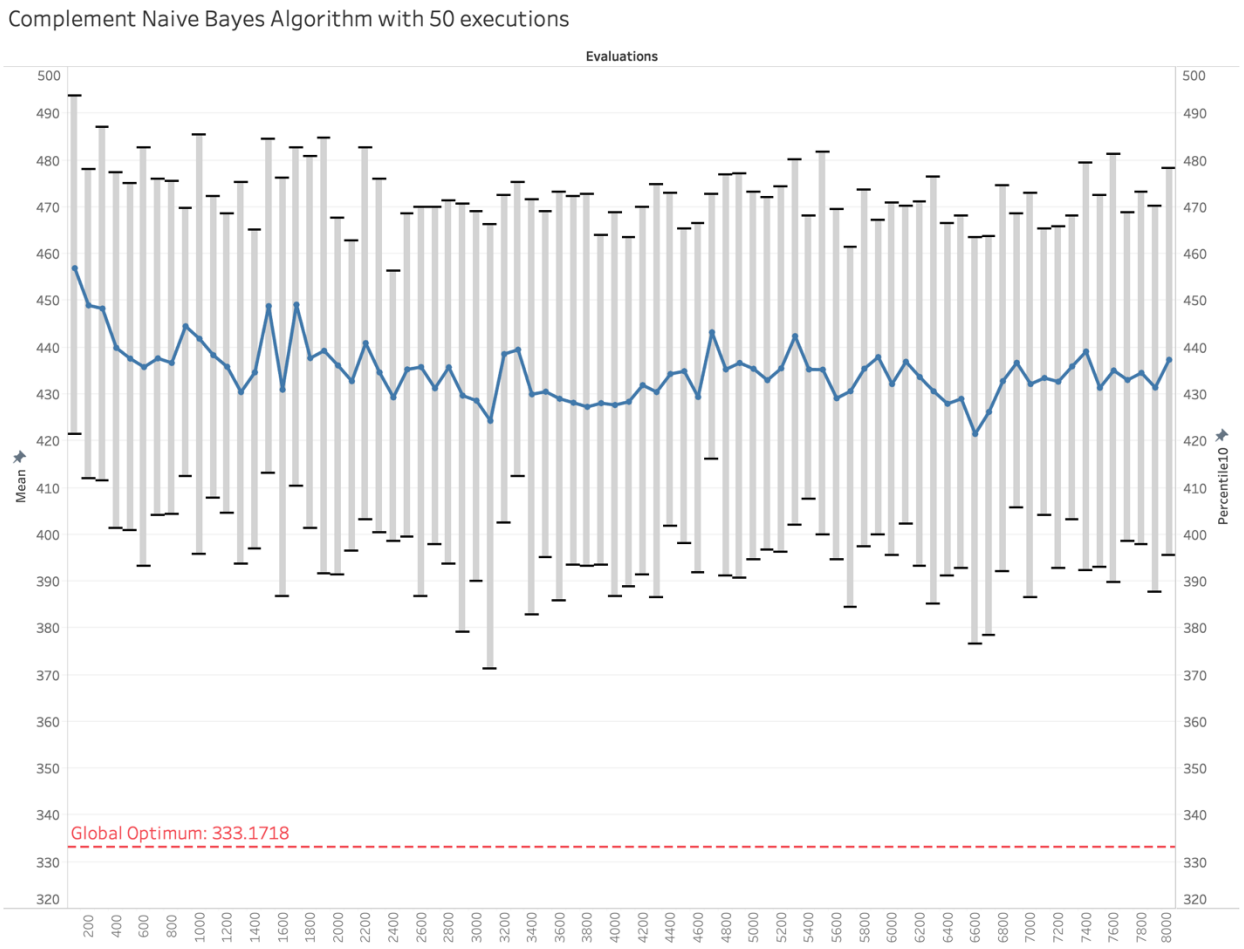

Figure 10. TSP result by complement naïve Bayesian

- Classifier performance:

In the aspect of classifier accuracy, complement naïve Bayesian is much better than multinomial naïve Bayesian. After we switched to using complement naïve Bayesian, the average recall is around 0.15. It's possible for the algorithm to find the optimum if we don't stop at the 8000 function evaluations.

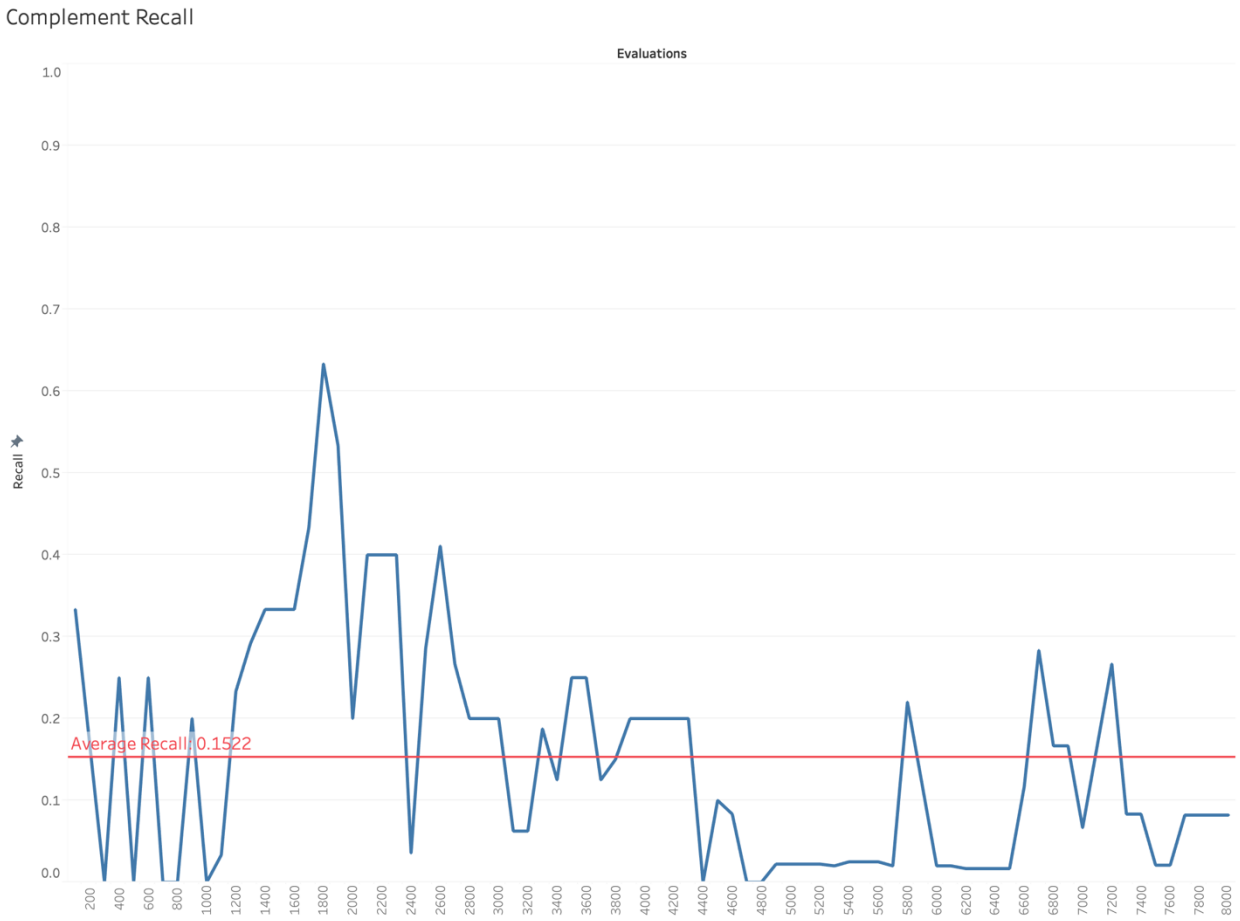

Figure 12. Recall score using complement naïve Bayesian

Discussion and future works

The CGS algorithm is an effective method for solving optimization problems with discrete and/or continuous variables and continuous and/or discontinuous responses. However, the performance of the algorithm heavily depends on the accuracy of the classifier. One challenge for implementing the CGS algorithm is that there is no a classifier that is universally good for every problem. For example, in the twenty-item optimization problem, we were able to reach the global optimum only after we switched from multinomial to Bernoulli naïve Bayesian classifier and this is because we know that our data follows a Bernoulli distribution. Imaging we have a data with an unknow distribution; if we don't have the knowledge about the distribution of our data, we would not know how to choose the classifier. In this case, we can only try out various types of classifier, which will decrease the computational efficiency, and yet there is no guarantee we will find a suitable classifier. Another limitation is that the CGS algorithm is based on probabilities, however, not every classifier will output a probability given the classes, therefore, the CGS algorithm is limited to only probabilistic classifiers, for example, Bayesian network, logistic regression, and etc. In addition, although various Bayesian classifier are very popular in the machine learning community, the majority of them are based on the independence assumption which means naïve Bayes. However, sometimes we will encounter a problem that severely violates this assumption, for example, the Traveling Salesperson Problem. Solving these types of problems, we need a more complicated Bayesian Network. Therefore, the future works of this project will be: 1. Develop an augmented naïve Bayesian classifier for the TSP problem 2. Test other probabilistic machine learning classifiers 3. Develop other method or modify the CGS algorithm to make it compatible with non-probabilistic classifiers.

References

Peter B. Backlund, David W. Shahan & Carolyn Conner Seepersad (2015) Classifier-guided sampling for discrete variable, discontinuous design space exploration: Convergence and computational performance, Engineering Optimization, 47:5, 579-600, DOI: 10.1080/0305215X.2014.908869

Documentation of Naïve Bayesian in Python Scikit-Learn Package
https://scikit-learn.org/stable/modules/naive_bayes.html