William Tang and Jason Dinh
Professor Jean-François Brière
Intro to Programming
14 May 2019

Wine Classification with Machine Learning

Introduction

Classification is a broad and important topic in machine learning. A program that can classify things into different categories autonomously has a variety of uses (spam filters, disease diagnostics, etc.) and provides a solution for sorting through large amounts of data. As an introductory project into machine learning and classification, we look at one type of classifier called logistic regression. We wanted to know how the parameters of a logistic regression affect the performance of a classification and how we can optimize them. After finding the appropriate data, we build a model from scratch to see if it would correctly predict between two sorts of wine at a time. We then evaluate how our model performed and study some of the parameters used.

Model

To classify things in supervised learning, we will be using **logistic regression**. Logistic regression takes the data as an input (in our case, characteristics of the wine) and outputs the appropriate category for that element (type of wine). To simplify things, we deal with **binary logistic regression**; our program will categorize between two different things at a time only.

So the goal in logistic regression is to put things of one type in one category, and things of another type in the other category. It uses a **sigmoid function** to return a probability value between 0 and 1. Based on some threshold that we decide, that value will be rounded to a 0 or a 1 - these represent the categories, and the program can now make the prediction.

The sigmoid function is what allows us to do this. It maps any real value into another value in the range (0,1). It is given by:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Now, where do the characteristics of the wine come in? We'll make up some equation that takes a linear combination of the 13 characteristics, plus a constant. The equation that takes the linear sum of all the feature variables (that are multiplied by some coefficients beta) is given as follows :

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ \dots \ + \ \beta_{13} x_{13}$$

Where the *x* variables are the numerical values of the characteristics of the wine. In order to get the "prediction", we'll run the value of this sum through the sigmoid function.

$$h(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_{13} x_{13})}}$$

This is called the **hypothesis function**. Now we have something that, given a set of input parameters (characteristics of wine), will spit out a number between 0 and 1 and make a prediction on the type. **Our goal will be to find the best $\beta_0, \beta_1, \beta_2, ...$ parameters that will give the most accurate predictions.**

To achieve this, we need another function in our model: something that will calculate the error between the prediction the program gives and the actual answer. In logistic regression, the mathematical equation of the error J is given as:

$$J = cost(h(x),\ y) = -\frac{1}{m}\sum_{i=1}^{m} y_i log(h(x_i)) + (1-y_i)log(1-h(x_i))$$

Again, the x_i variables represent the numerical values of the characteristics of the wine, and the y_i variables indicate the type of wine (turned into 0 or 1). *m* is the total number of data points. We call this the **cost function, or the loss function**. In order to find the best parameters that give the most accurate predictions, we will have to minimize the error between predictions and real answers, which in turn means minimizing the the function that represents the error - the cost function.

Where does this cost function come from? We can actually think of this problem as *maximizing the likelihood of our model.* In statistics, probability and likelihood are different concepts. Suppose we have a model that will try to predict the right outcomes (class labels) given a set of parameters like we saw in the previous paragraphs. However, we do not know the specific values for this set of parameters. All we can do is observe the target classes and come out with some kind of estimation for those parameters. ***Maximizing the likelihood*** **comes down to choosing values of the parameters that would maximize the probability that we will get the right class predictions.** In logistic regression, the overall likelihood is computed by simply taking product of the y-values of the sigmoid function, given a set of parameters (in our case we denote them by $\beta_0, \beta_1, \beta_2, ...$). You can think of this as the probabilities; look at the hypothesis function for a reminder. We need to look at the likelihood that our model will correctly predict between the two classes. As such, we can now define the likelihood of a single point as $h(x)$ when the the label is 1 (probability of predicting a 1) and as $1 - h(x)$ when the label is 0 (probability of predicting a 0). All we have to do is multiply all of these likelihoods together.

Lots of people prefer to take the logarithm of the overall likelihood to make the function smoother and easier to work with. So in the end, after a couple of mathematical simplifications, we end up with this:

$$\sum_{i=1}^{m} y_i log(h(x_i)) + (1-y_i)log(1-h(x_i))$$

Depending on whether or not the actual class label is 1 or 0, one part of the equation will disappear.

In machine learning, a lot of the methods we use are based on minimizing a function. Because the likelihood function has a maximum, what we can do is flip the sign by adding a negative sign. Now we have something that looks like the cost function we described (dividing by *m* is simply taking the average).

In the end, **logistic regression will be interested in finding the $\beta_0, \beta_1, \beta_2, ...$ parameters that should give the smallest value for the cost function, which in turn maximizes the likelihood**. We describe the method used to achieve this in the following section.

Method

We would now like to know how to find the $\beta_0, \beta_1, \beta_2, ...$ parameters that minimize the cost function, given a set of characteristics of the wine. The numerical method we use here is called **gradient**

descent. Recall that the gradient of a function is just the vector whose components are the partial derivatives of the function in question, with respect to the components (x_1, x_2, ..., x_n) of the function. We may compare gradient descent method with Euler's method: it calculated the slope, and *then took a step in the direction of that slope* to estimate the actual curve. The same idea applies here. If we have the slope (the partial derivative), then we can take a step (the size of which we define) in the direction of the slope - as we saw in Euler's method, small steps ensure that we follow that actual curve. Also, in our case we are always planning on going downwards. The goal is to take steps until we reach the bottom of the curve, the bottom of our cost function. We will know that we reached the bottom (the minimum) when the gradient is zero, or really close to zero. The function we are dealing with uses the logarithm of values. This makes the curve "smooth", as there will be no local minima/maxima. This means that the minimum we find will also be the global minimum.

Now, the slope changes as we compute a new partial derivative at the updated point. The derivative at a minimum should be zero, which means the difference between the updated point and the previous one should also be zero at that point. **This means that if we are continuously updating our point (a,b), at some point it will stop decreasing.** In other words, we found the minimum. All we have to do is start at a random initialization of $\beta_0, \beta_1, \beta_2, ..$, and go down from there. For a convex function, it does not matter where you start since there will be only one minimum. For a non-convex function, it becomes more problematic…. Fortunately, in our project the function to minimize is convex: as we saw earlier, taking the log likelihood makes the function smooth and have no local minima/maxima.

The gradient descent method is as follows:

$$\beta_0 = \beta_0 - \alpha\frac{\delta J}{\delta\beta_0}$$
$$\beta_1 = ...$$
$$...$$

As seen above, each coefficient β_j is brought into a continuous loop. We update all of them at the same time. α is the **learning rate*;*** it's how big of step you want to take each time. Finally, for any $\beta_j x_j$ parameter in the hypothesis function

$$h(x) = \frac{1}{1 + e^{-(\beta_0+\beta_1x_1+\beta_2x_2+ ... + \beta_{13}x_{13})}}$$

the partial derivative of **the cost function** with respect to β_j will be given by

$$\frac{\delta J}{\delta\beta_j} = \frac{1}{m}\sum_{i=1}^{m}(h(x_i) - y_i)x_{j\ i}$$

Afterwords, the difference between the $\beta's$ are calculated. If this is below the tolerance level, the loop is terminated and the results are displayed. Another method is just to iterate for a set number of times; if we know that the cost function does not change much after a certain point, this way could be faster.

One thing that was also included in our model was a normalization of the data. Normalization is really useful when your results would be affected by the feature with the largest variance or scale: you could have one numerical feature that has a range from 0-10, while another could have a range from 1000 to 10000. Since one feature could be very large while the other is small, this obviously is not too ideal. On a graph, you can think of the **cost function** resembling really steep canyons, instead of nicely shaped circular pits. What happens is that to reach the optimal point, you may need to a very large step in one direction, while only needing to take a small step in the other direction. If the step size is too large, you will easily overshoot the optimal point; but if the step size is small, it will take really long!

This is why it is preferable to have all our data on the same scale. **Normalizing your inputs can help the gradient descent converge much faster.** There are a couple of ways of doing this, one of which is a standardization to z-scores. This consists of taking all the data points of *one feature/characteristic* and scaling the whole set to a **normal distribution with mean 0, standard deviation 1.** For example, consider the numerical values of the "alcohol" characteristic in our wine dataset. Each wine has a value for the alcohol level. We will need to apply the following mathematical transformation on each of the alcohol values:

$$z = \frac{x - \mu}{\sigma}$$

Where z is the scaled feature, x is the initial value for the feature, and μ and σ respectively are the average and standard deviation of the all the values of the feature in question.

Results

Creating the model

The first step is to create a working logistic regression. We use the wine dataset from the UCI Machine Learning repository to classify between two types of wine, specifically the wine labelled as "1" and the one labelled as "2" (this was changed to "0" for our logistic regression). Before starting, however, it is nice to get a sense of what our data looks like. Using a Jupyter notebook (adapted from class labs), we plotted out a few feature characteristics between the two wines.

Figure 1 shows four of the plots we did. We see some good separation between the two wines. This shows that feature 1 (alcohol) and feature 13 (proline) may be good indicators of the difference between the type of wine. This also tells us that a linear separation may be possible, so a simple logistic regression would work.

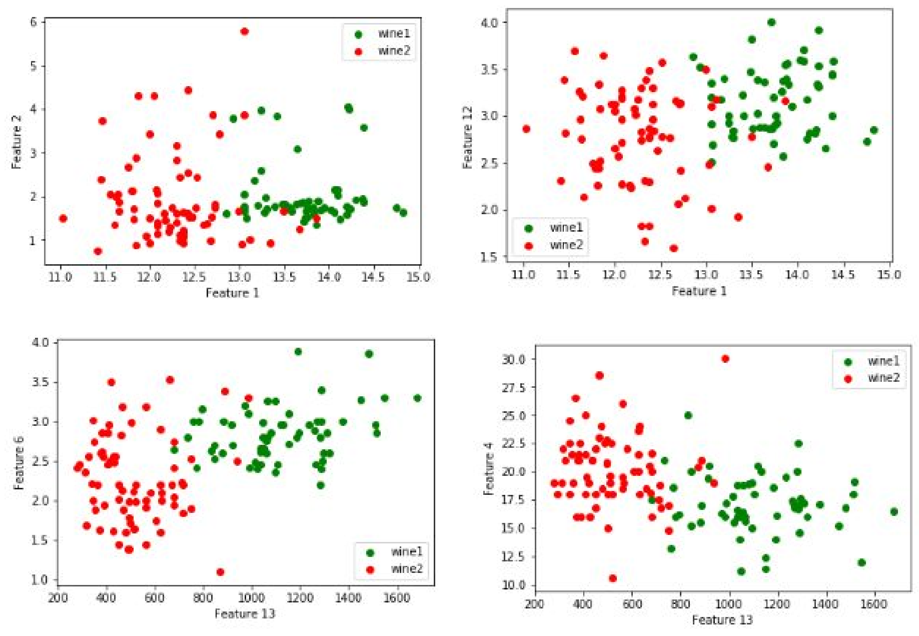


Figure 1: Exploratory Data

After this exploratory data phase, we can start the logistic regression. The wine dataset had 130 data points, containing classes for wine 1 (1) and wine 2 (0). Each point contains 13 different characteristics. Initially, the data is shuffled around and split into training and test sets at a **70:30 ratio.** A gradient descent is performed with the training set, and initial weights are set to zero. For this experiment, we used **a step size of 0.1 with 5000 iterations** of gradient descent. Also note that we always **standardize all feature data to z-scores beforehand**. After everything is done, we can directly get the predictions of our trained model and validate the results **with our test set, with a prediction threshold of 0.5 (this is what we use for all other trials unless indicated otherwise).** Checking the accuracy of our model is as simple as comparing the prediction of the model with the actual class label. The total number of correct answers is divided by the total number of predictions to get the accuracy. We ran the logistic regression 1000 times and took the average accuracy.

Now we have the experimental model of logistic regression to study. Before starting, however, we check to see if the gradient descent is working as it should by plotting the cost function of this model over the amount of iterations. We expect the cost to decrease over time, and the rate of change to slow down considerably after some iterations. Figure 2 confirms our logistic regression is working properly.

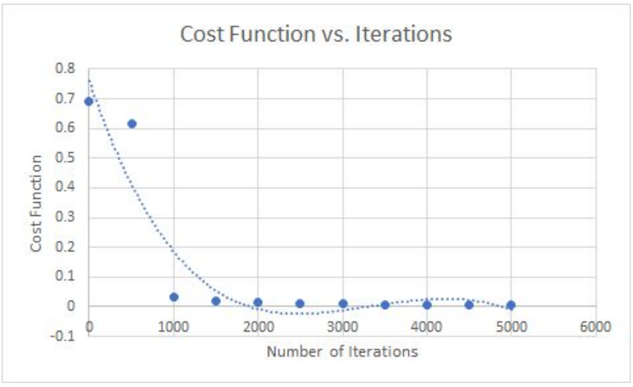


Figure 2: Cost vs. Iterations

We can also check the accuracy of the model with a confusion matrix.

	Predicted : 0	**Predicted: 1**
Actual: 0	21235 (true negative)	486 (false positive)
Actual: 1	48 (false negative)	17231 (true positive)

Figure 3: Confusion matrix

From this table in figure 3, we can see the predictions the logistic regression made, as well as the actual class it made the prediction on. This allows us to obtain other measures of error other than the accuracy:

Error Type	**Value**	**Error Type**	**Value**
Accuracy	0.9863076923076923	**Specificity**	0.9776253395331707
Error Rate	0.013692307692307693	**Precision**	0.9725687193091381
Sensitivity/Recall	0.9972220614618902	**Prevalence (class 1)**	0.443051282051282
False Positive Rate	0.022374660466829337	**Specificity**	0.9776253395331707

Figure 4: Error Types

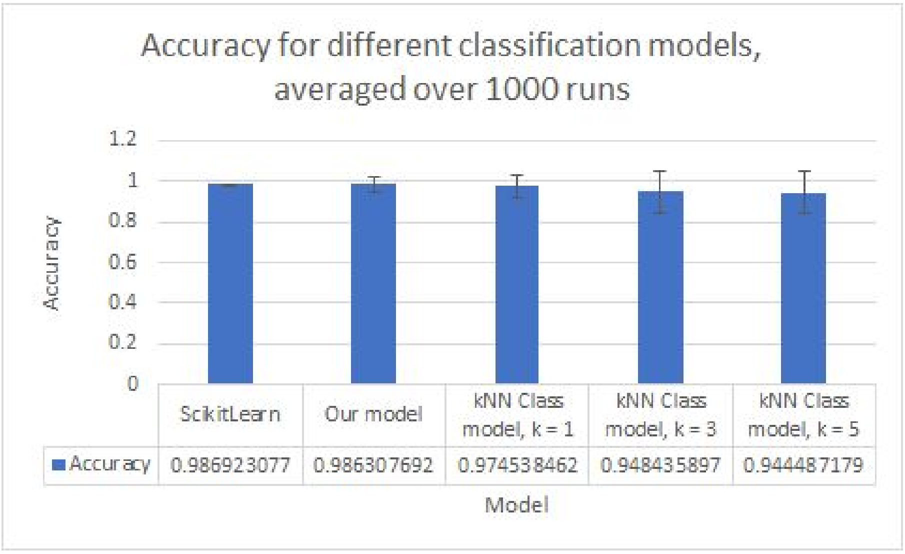
Accuracy and error rate go hand in hand; this is obtained following what we described in the previous paragraph. The **recall** is measure of the true positives divided by the number of actual positive labels. The **false positive rate** is the number of false positives divided by total number of actual negative labels. The **specificity** is essentially 1.0 minus the false positive rate: it is the number of true negatives divided by the total number of actual negative labels. The **precision** is the number of true positives divided by the total number of predicted positive classes. Finally, the **prevalence** of the wine of class "1" indicates the proportion of this class in the dataset. One good baseline accuracy to compare our model with is just the highest number between 1.0 minus the prevalence, and the prevalence itself. The "baseline" model in this case would just look at the most frequent class, and classify everything as that class. We see that the baseline accuracy is about **55.7%** here. The model performs much better than this. In summary, the model has a good performance on all metrics.

Now we would like to see how our trained model fits the wine dataset. Unlike linear regression, there are many ways to calculate a measure of how well the model fits your data. We chose to use McFadden's R^2 value and use it to evaluate our logistic regression. It is calculated by taking the difference between **the log likelihood of the trained model minus the log likelihood of a model without any predictor variables.** This difference is then divided by the latter. A model without any predictor variables is simply a logistic regression trained without the characteristics of the wine! From this R^2 value, results closer to 0 are indicators of a bad fit, while results closer to 1 indicate a good fit. Figure 5 shows that the logistic regression model and the $\beta_0, \beta_1, \beta_2, \ldots$ parameters it found fits the data pretty well, and should have a high likelihood of giving the right predictions.

Set	**McFadden's R^2**
Training Set	0.9951609615489888
Test Set	0.9624937103640429

Figure 5: McFadden's R-Squared

At this point, we can assume that the logistic regression model we made is working as intended: the cost function is going down, accuracy and related measures of error are where they should be, and the model fits the data quite well. The last thing to do before studying the model and its parameters is to compare what we made with other classifiers, just to make sure our results are not abnormal for this dataset. Figure 6 is a comparison of the accuracy of our trained model against other classification algorithms.



	ScikitLearn	Our model	kNN Class model, k = 1	kNN Class model, k = 3	kNN Class model, k = 5
■ Accuracy	0.986923077	0.986307692	0.974538462	0.948435897	0.944487179

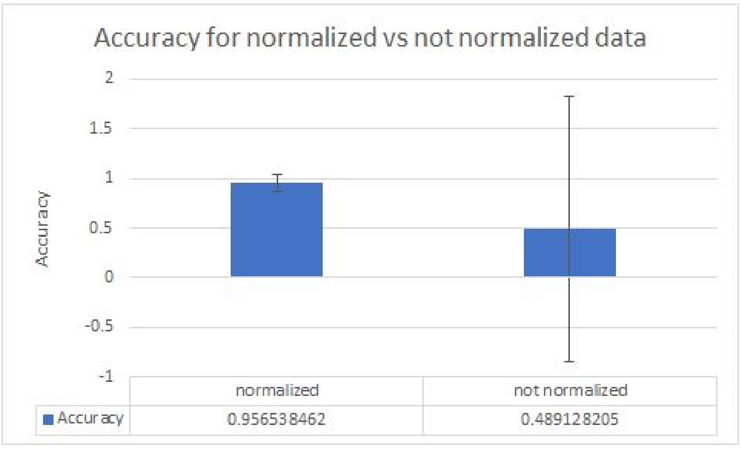
Figure 6: Accuracy Comparison

ScikitLearn is a machine learning library in python. We used their logistic regression model and tested against the same wine dataset. Unlike our model where we can modify the step size and number of iterations, no parameters need to be specified when running this program. Furthermore, we used the k Nearest Neighbours model done in a previous class lab to classify the wine dataset as well. Apart from changing the hyperparameter **k**, no other modifications were made to the original code. All classification models were used on the same dataset. Accuracy is an an average over 1000 runs. Figure 6 tells us that the results we are getting are normal (other, similar classifiers are getting around the same thing).

Studying the model

We have created the logistic regression and have validated its classification performance. This next section deals with results related to the study of certain parameters and settings we can tweak in the logistic regression to make it even better.

Normalizing the data, as we said earlier, allows for the logistic regression to converge much faster. To show this, we experimented with logistic regression *without* normalization, and one *with* normalization, keeping all other parameters constant:



	normalized	not normalized
🟦 Accuracy	0.956538462	0.489128205

Figure 7: Normalized Accuracy

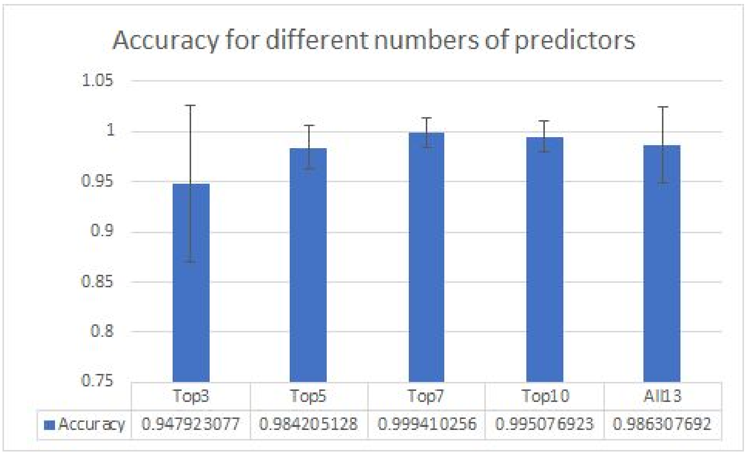
Figure 7 was created with the same model made in the first section, but the gradient descent runs for **10 iterations. The accuracy displayed is an average accuracy over 1000 runs.** The method of normalization was a **standardization to z-scores**. As seen from the results, normalized data converges quite quickly and reaches around 95% accuracy after *only 10 iterations of gradient descent*, while the data set that did not have any normalization had horrible results (worse than the baseline of around 55%!). The graph shows that normalization has a significant influence on the speed of convergence of the logistic regression.

Another question we can ask ourselves is which characteristics of wine give a good indication of whether it belongs to one class or another. In other words, we are looking to see which characteristics of wine influence the prediction model the most. To do this, we may **rank the trained weights of our model based on their absolute value (the parameters** $\beta_0, \beta_1, \beta_2, ...$), but it is important that we do this on normalized data, since it is useless to compare values that are on different scales. Figure 8 shows the top 10 predictors of wine 1 vs wine 2. It was created with the original logistic regression model we made in the first section. From this information, we are able to compare accuracies of models for different numbers of predictors. Figure 9 shows the accuracies of different logistic regressions that have a certain number of top predictors. Again, we used the same settings as our original logistic regression model in the first section.

Each accuracy represents the average over 1000 runs. We observe that the highest accuracy is obtained with the top 7 predictors.

Feature Num	Coefficient value
13	3.726950223446625
1	2.971185746740978
4	-1.9918437015638841
3	1.7652098282721669
2	1.4146953732198817
12	1.3846765321707586
10	1.2813960452926596
9	-0.6775803535431819
8	-0.6343208469819344
5	0.5990987837092538

Figure 8: Weight Ranking



	Top3	Top5	Top7	Top10	All13
■ Accuracy	0.947923077	0.984205128	0.999410256	0.995076923	0.986307692

Figure 9: Accuracy for different number of predictors

Next, we can also study the tolerance level. Instead of running the logistic regression for a set amount of iterations like we did, we can indicate a stopping condition for convergence. If the difference between one coefficient and its updated version from the gradient descent is near zero, we may stop the loop. This seems like a rather obvious feature to include in our model, but is it really that useful? Figure 10 was generated from one run of a the logistic regression, with **step size 0.001, a maximum number of iterations allowed of 30000, with varying tolerance levels.** The tolerance starts at **0.00000001** and goes **all the way up to 0.0001**. **The blue curve** indicates the accuracy. **The red bar plot** indicates the log value of the number of iterations it took (a log value was used in order to scale it properly to the graph). As expected, as the tolerance level increases, the accuracy tends to decrease as well. However, the number of iterations decreases, so it takes less time to run. Also, if the tolerance level gets too large, the loop will terminate after 1 or 2 iterations, which is not really what we want (see the normalization of data section to understand why it stays at 95% accuracy after only 1 or 2 iterations!).

Thus, figure 10 indicates to us that there will be trade-off between the accuracy and speed of the logistic regression if you consider a tolerance level for the model. One cannot arbitrarily define a tolerance level close to zero and assume it will go well.

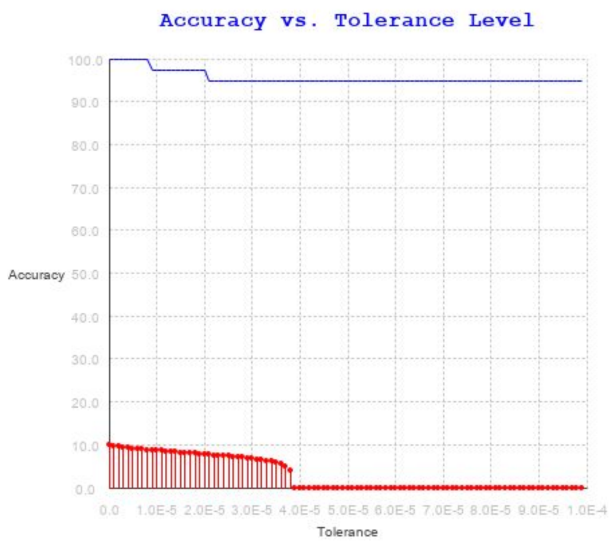


Figure 10: Accuracy vs. Tolerance

On a related note, throughout this whole process, we've kept the prediction threshold at 0.5. Although this may seem like a nice number, it is quite arbitrary as well. How do we know that this is the appropriate threshold to round up or round down the probabilities? One of the ways to do this is to look at the ROC curve, or the Receiver-Operator-Characteristic curve of the logistic regression.

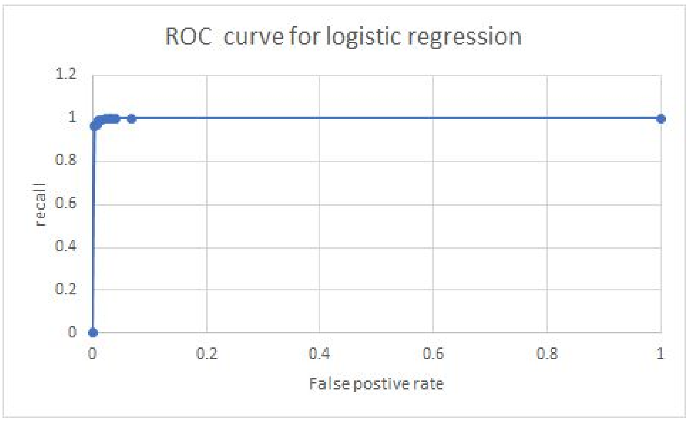


Figure 11: ROC curve

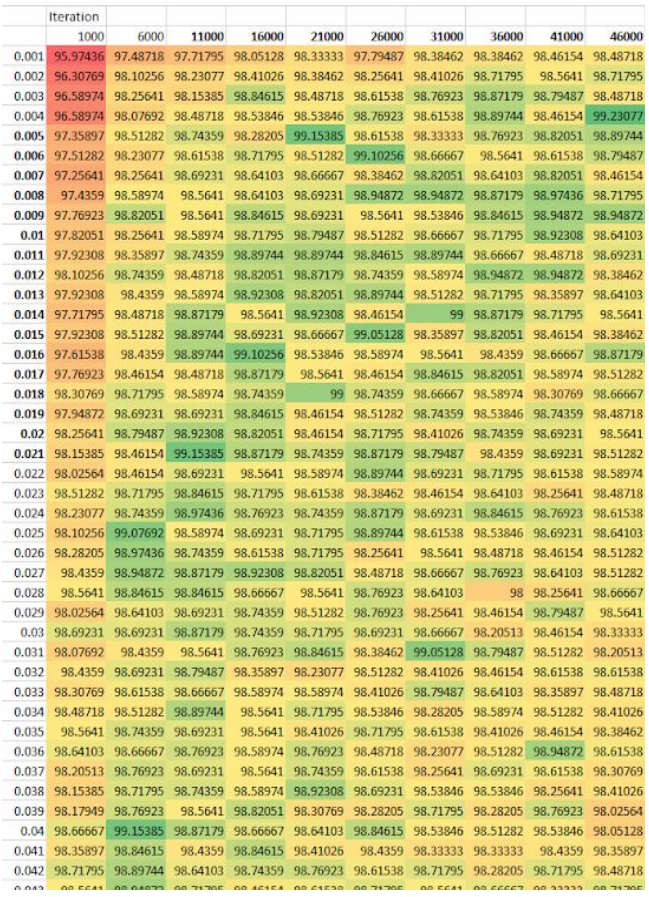
Threshold	Recall	FPR	Threshold	Recall	FPR
0	1	1	0.5	0.997674419	0.02293578
0.1	1	0.06765233	0.6	0.993067591	0.015214385
0.2	1	0.041189931	0.7	0.991831972	0.010978957
0.3	0.999419617	0.032154341	0.8	0.975044563	0.00766802
0.4	0.997683845	0.03083295	0.9	0.962449451	0.003688336
			1	0	0

The ROC curve plots the **recall** of the model against the **false positive rate**. Points are obtained by computing the recall and false positive rate for different prediction thresholds. Figure 11 was created by computing the average recall and false positive rate of logistic regression over 100 trials for each threshold. Other than varying the prediction threshold, all other parameters were identical to the original model made in the first section. We observe that when choosing the prediction threshold, there is always a trade-off between recall and the false positive rate, and that going for 0.5 is not necessarily the best option, depending on what the context is.

Finally, we may study the relationship between the step size and the number of iterations. We decided to approach this aspect by studying their relationship experimentally; that is, to see how the step size (α) and number iterations together can affect the accuracy of our program, we make a heat map of accuracy. We looped through the program, and have it calculate and print out the accuracy for different step sizes **(0.001 up to 0.1)** and different iterations **(1000 to 49,000 iterations)**. Normalized data was used, and all weights started at zero. The prediction threshold was set to 0.5.

Figure 12 shows the first half of the data.[1] Because the accuracy can vary for different run, taking the accuracy from only one run would not be accurate, that's why for each alpha and iteration, we take the **average of 100 independent trials**. We then put the all of the accuracies into Excel, colouring **lower accuracy with red and high accuracy with green.** In the heat map, the columns are number of iterations and the rows are different step sizes. Although there is a lot of noise, notice that the green coloured zone starts off on the right side, then tends to exponentially decrease in iterations as the alpha increases. The result is exactly what we would expect: in general, smaller step sizes have a larger optimal number of iterations. Larger step sizes, on the other hand, reach their optimal accuracy at a much smaller number of iterations, as we can see from the graph.

As a result, figure 12 is useful in two ways: it shows the general trend of the evolution of the optimal pair of step size and iterations; and it allows us (and others!) to directly get the optimal solution for a given step size/number of iterations.



	Iteration									
	1000	6000	11000	16000	21000	26000	31000	36000	41000	46000
0.001	95.97436	97.48718	97.71795	98.05128	98.33333	97.79487	98.38462	98.38462	98.46154	98.48718
0.002	96.30769	98.10256	98.23077	98.41026	98.38462	98.25641	98.41026	98.71795	98.5641	98.71795
0.003	96.58974	98.25641	98.15385	98.84615	98.48718	98.61538	98.76923	98.87179	98.79487	98.48718
0.004	96.58974	98.07692	98.48718	98.53846	98.53846	98.76923	98.61538	98.89744	98.46154	99.23077
0.005	97.35897	98.51282	98.74359	98.28205	99.15385	98.61538	98.33333	98.76923	98.82051	98.89744
0.006	97.51282	98.23077	98.61538	98.71795	98.51282	99.10256	98.66667	98.5641	98.61538	98.79487
0.007	97.25641	98.25641	98.69231	98.64103	98.66667	98.38462	98.82051	98.64103	98.82051	98.46154
0.008	97.4359	98.58974	98.5641	98.64103	98.69231	98.94872	98.94872	98.87179	98.97436	98.71795
0.009	97.76923	98.82051	98.5641	98.84615	98.69231	98.5641	98.53846	98.84615	98.94872	98.94872
0.01	97.82051	98.25641	98.58974	98.71795	98.79487	98.51282	98.66667	98.71795	98.92308	98.64103
0.011	97.92308	98.35897	98.74359	98.89744	98.89744	98.84615	98.89744	98.66667	98.48718	98.69231
0.012	98.10256	98.74359	98.48718	98.82051	98.87179	98.74359	98.58974	98.94872	98.94872	98.38462
0.013	97.92308	98.4359	98.58974	98.92308	98.82051	98.89744	98.51282	98.71795	98.35897	98.64103
0.014	97.71795	98.48718	98.87179	98.5641	98.92308	98.46154	99	98.87179	98.71795	98.5641
0.015	97.92308	98.51282	98.89744	98.69231	98.66667	99.05128	98.35897	98.82051	98.46154	98.38462
0.016	97.61538	98.4359	98.89744	99.10256	98.53846	98.58974	98.5641	98.4359	98.66667	98.87179
0.017	97.76923	98.46154	98.48718	98.87179	98.5641	98.46154	98.84615	98.82051	98.58974	98.51282
0.018	98.30769	98.71795	98.58974	98.74359	99	98.74359	98.66667	98.58974	98.30769	98.66667
0.019	97.94872	98.69231	98.69231	98.84615	98.46154	98.51282	98.74359	98.53846	98.74359	98.48718
0.02	98.25641	98.79487	98.92308	98.82051	98.46154	98.71795	98.41026	98.74359	98.69231	98.5641
0.021	98.15385	98.46154	99.15385	98.87179	98.74359	98.87179	98.79487	98.4359	98.69231	98.51282
0.022	98.02564	98.46154	98.69231	98.5641	98.58974	98.89744	98.69231	98.71795	98.61538	98.58974
0.023	98.51282	98.71795	98.84615	98.71795	98.61538	98.38462	98.46154	98.64103	98.25641	98.48718
0.024	98.23077	98.74359	98.97436	98.76923	98.74359	98.87179	98.69231	98.84615	98.76923	98.61538
0.025	98.10256	99.07692	98.58974	98.69231	98.71795	98.89744	98.61538	98.53846	98.69231	98.64103
0.026	98.28205	98.97436	98.74359	98.61538	98.71795	98.25641	98.5641	98.48718	98.46154	98.51282
0.027	98.4359	98.94872	98.87179	98.92308	98.82051	98.48718	98.66667	98.76923	98.64103	98.51282
0.028	98.5641	98.84615	98.84615	98.66667	98.5641	98.76923	98.64103	98	98.25641	98.66667
0.029	98.02564	98.64103	98.69231	98.74359	98.51282	98.76923	98.25641	98.46154	98.79487	98.5641
0.03	98.69231	98.69231	98.87179	98.74359	98.71795	98.69231	98.66667	98.20513	98.46154	98.33333
0.031	98.07692	98.4359	98.5641	98.76923	98.84615	98.38462	99.05128	98.79487	98.51282	98.20513
0.032	98.4359	98.69231	98.79487	98.35897	98.23077	98.51282	98.41026	98.46154	98.61538	98.61538
0.033	98.30769	98.61538	98.66667	98.58974	98.58974	98.41026	98.79487	98.64103	98.35897	98.48718
0.034	98.48718	98.51282	98.89744	98.5641	98.71795	98.53846	98.28205	98.58974	98.51282	98.41026
0.035	98.5641	98.74359	98.69231	98.5641	98.41026	98.71795	98.61538	98.41026	98.46154	98.38462
0.036	98.64103	98.66667	98.76923	98.58974	98.76923	98.48718	98.23077	98.51282	98.94872	98.61538
0.037	98.20513	98.76923	98.69231	98.5641	98.74359	98.61538	98.25641	98.69231	98.61538	98.30769
0.038	98.15385	98.71795	98.74359	98.58974	98.92308	98.69231	98.53846	98.53846	98.25641	98.41026
0.039	98.17949	98.76923	98.5641	98.82051	98.30769	98.28205	98.71795	98.28205	98.76923	98.02564
0.04	98.66667	99.15385	98.87179	98.66667	98.64103	98.84615	98.53846	98.51282	98.53846	98.05128
0.041	98.35897	98.84615	98.4359	98.84615	98.41026	98.4359	98.33333	98.33333	98.4359	98.35897
0.042	98.71795	98.89744	98.64103	98.74359	98.76923	98.61538	98.71795	98.28205	98.71795	98.48718
0.043	98.5641	98.94872	98.71795	98.46154	98.61538	98.71795	98.5641	98.66667	98.33333	98.71795

Figure 12: Heatmap of accuracy

[1] Full heatmap of accuracy is found in Annex I

Discussion

In this project, we successfully made a logistic regression classifier that is able to predict between two types of wine. We observed that it was capable of a fairly high accuracy of classification. We also observe that there is no statistically significant difference in accuracy between our logistic regression model and other classification models.

With this in hand, we were able to study and answer our research question. There are many factors that can influence the performance of a logistic regression. Normalization, number of class characteristics, the stopping condition for convergence, the prediction threshold, the step size, and the number of iterations in gradient descent are all factors that play into the performance of logistic regression.

Normalization of data allows our model to find the optimal solution much faster. Results indicate that a good logistic regression should always include it in their model.

In the wine dataset, features 13 ("proline") and 1 ("alcohol") were the strongest predictors of wine type, as we hypothesized in the exploratory data phase. When we tested a logistic regression with only the top 7 predictors of wine type, we managed to get far better averaged results than the model with all 13 characteristics. This shows that the procedure by which we **rank the absolute value of the** $\beta_0, \beta_1, \beta_2, \ldots$ **coefficients in descending order** is an extremely useful tool to single out the useless predictors versus the strong ones.

For the **stopping condition**, results show that there is always a trade-off of one aspect in return of the other. If we did use a tolerance level in our experiments, we would have to think about how much performance we get at the cost of prediction accuracy. As a result, using a stopping condition does not necessarily improve the model. Choosing a tolerance level would need your team to consult the requirements or the context of the situation that the logistic regression is being applied to. Afterwards, you may need to experimentally determine the best tolerance level.

As for the **prediction threshold**, 0.5 is not always the best! Although 0.5 is a good default most of the time, it really depends on whether the recall or false positive rate is more important. This is similar to the stopping condition; there is always a trade-off, and it depends on the context. For example, when diagnosing rare and life-threatening diseases, you would want a high recall to avoid false negatives (saying the patient is healthy when he's actually dying is bad!).

Finally, it is also possible to study the **step size and the number of iterations of gradient descent.** It is understood that by choosing random step sizes and iteration numbers, we could easily overshoot the minimum, or not do enough iterations to reach the minimum. The problem, then, is how we can find the correct amount of iterations for an appropriate step size. We can't just have the program run with an extremely small alpha for a high number of iterations; it would take too long for the program to run. This is where the heat map of the accuracy comes in handy. We can experimentally determine the optimal set of hyperparameters for our logistic regression in this way. The optimal solution is found at a high number of iterations for small alpha, while for large alpha the optimal solution is found for a smaller amount of iterations. The full heatmap shows that our choice of taking a step size **of 0.1 with 5000 iterations was not that bad of a choice, although 1000 iterations seems to be slightly better.**

In conclusion, although some parameters need to be studied with an appropriate context (tolerance level and prediction threshold), our project has shown that there are many other parameters to take into account if you want a well-performing logistic regression.

References

D., Dua and C., Graff. “Wine Data Set.” *UCI Machine Learning Repository*, Irvine, University of California, School of Information and Computer Science, http://archive.ics.uci.edu/ml

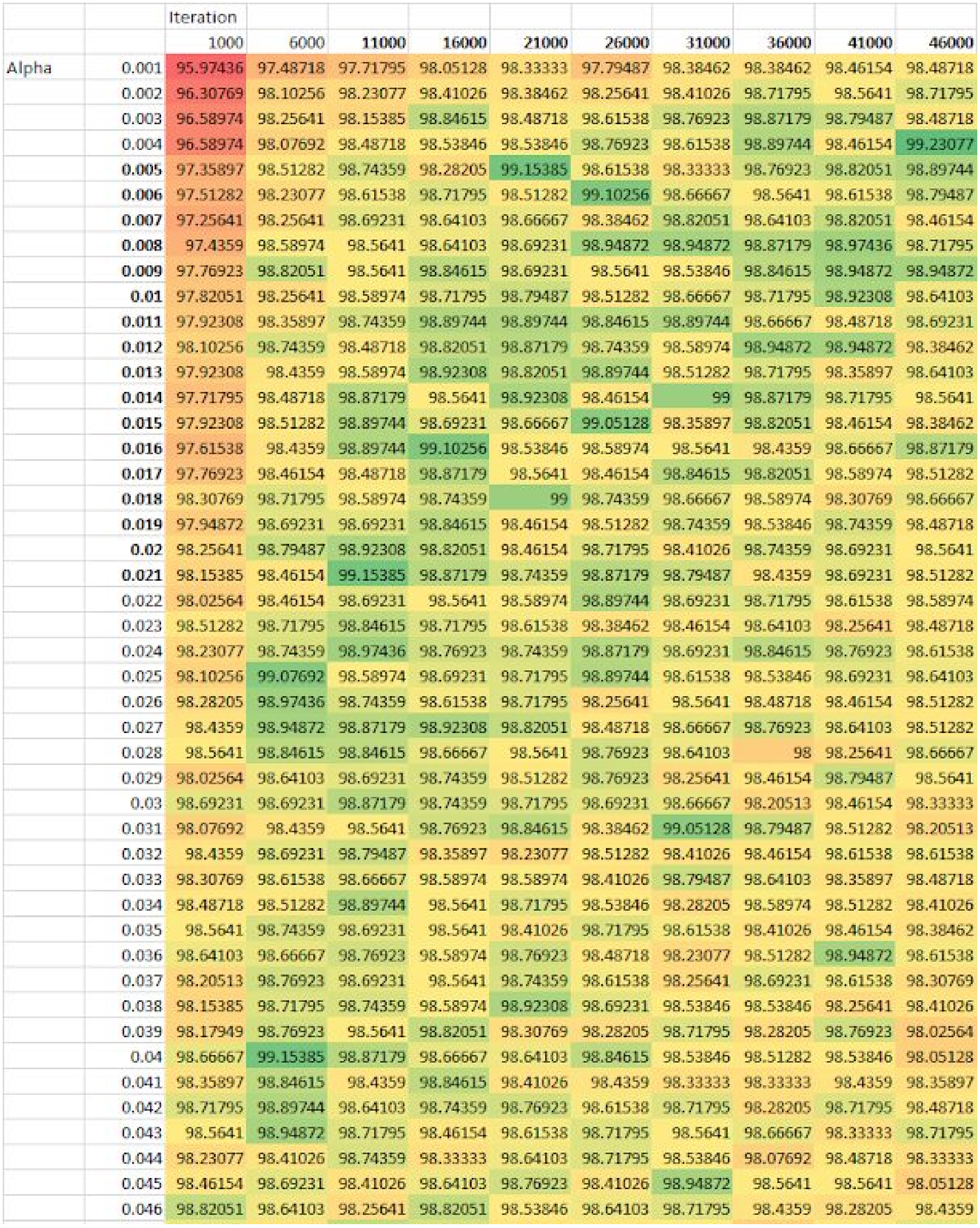
McFadden, D. 1974. Conditional logit analysis of qualitative choice behavior. In: Frontiers in Economics, P. Zarembka, eds. New York: Academic Press.

“ROC Curve Analysis.” *MedCalc*, https://www.medcalc.org/manual/roc-curves.php.

Molnar, Christoph. “4.2 Logistic Regression.” 12 April 2019, https://christophm.github.io/interpretable-ml-book/logistic.html

“The Cost Function in Logistic Regression.” *Internal Pointers*, 31 March 2018, https://www.internalpointers.com/post/cost-function-logistic-regression

Annex I - Full Heatmap of Accuracy based on Alpha and Iterations



		Iteration									
		1000	6000	11000	16000	21000	26000	31000	36000	41000	46000
Alpha	0.001	95.97436	97.48718	97.71795	98.05128	98.33333	97.79487	98.38462	98.38462	98.46154	98.48718
	0.002	96.30769	98.10256	98.23077	98.41026	98.38462	98.25641	98.41026	98.71795	98.5641	98.71795
	0.003	96.58974	98.25641	98.15385	98.84615	98.48718	98.61538	98.76923	98.87179	98.79487	98.48718
	0.004	96.58974	98.07692	98.48718	98.53846	98.53846	98.76923	98.61538	98.89744	98.46154	99.23077
	0.005	97.35897	98.51282	98.74359	98.28205	99.15385	98.61538	98.33333	98.76923	98.82051	98.89744
	0.006	97.51282	98.23077	98.61538	98.71795	98.51282	99.10256	98.66667	98.5641	98.61538	98.79487
	0.007	97.25641	98.25641	98.69231	98.64103	98.66667	98.38462	98.82051	98.64103	98.82051	98.46154
	0.008	97.4359	98.58974	98.5641	98.64103	98.69231	98.94872	98.94872	98.87179	98.97436	98.71795
	0.009	97.76923	98.82051	98.5641	98.84615	98.69231	98.5641	98.53846	98.84615	98.94872	98.94872
	0.01	97.82051	98.25641	98.58974	98.71795	98.79487	98.51282	98.66667	98.71795	98.92308	98.64103
	0.011	97.92308	98.35897	98.74359	98.89744	98.89744	98.84615	98.89744	98.66667	98.48718	98.69231
	0.012	98.10256	98.74359	98.48718	98.82051	98.87179	98.74359	98.58974	98.94872	98.94872	98.38462
	0.013	97.92308	98.4359	98.58974	98.92308	98.82051	98.89744	98.51282	98.71795	98.35897	98.64103
	0.014	97.71795	98.48718	98.87179	98.5641	98.92308	98.46154	99	98.87179	98.71795	98.5641
	0.015	97.92308	98.51282	98.89744	98.69231	98.66667	99.05128	98.35897	98.82051	98.46154	98.38462
	0.016	97.61538	98.4359	98.89744	99.10256	98.53846	98.58974	98.5641	98.4359	98.66667	98.87179
	0.017	97.76923	98.46154	98.48718	98.87179	98.5641	98.46154	98.84615	98.82051	98.58974	98.51282
	0.018	98.30769	98.71795	98.58974	98.74359	99	98.74359	98.66667	98.58974	98.30769	98.66667
	0.019	97.94872	98.69231	98.69231	98.84615	98.46154	98.51282	98.74359	98.53846	98.74359	98.48718
	0.02	98.25641	98.79487	98.92308	98.82051	98.46154	98.71795	98.41026	98.74359	98.69231	98.5641
	0.021	98.15385	98.46154	99.15385	98.87179	98.74359	98.87179	98.79487	98.4359	98.69231	98.51282
	0.022	98.02564	98.46154	98.69231	98.5641	98.58974	98.89744	98.69231	98.71795	98.61538	98.58974
	0.023	98.51282	98.71795	98.84615	98.71795	98.61538	98.38462	98.46154	98.64103	98.25641	98.48718
	0.024	98.23077	98.74359	98.97436	98.76923	98.74359	98.87179	98.69231	98.84615	98.76923	98.61538
	0.025	98.10256	99.07692	98.58974	98.69231	98.71795	98.89744	98.61538	98.53846	98.69231	98.64103
	0.026	98.28205	98.97436	98.74359	98.61538	98.71795	98.25641	98.5641	98.48718	98.46154	98.51282
	0.027	98.4359	98.94872	98.87179	98.92308	98.82051	98.48718	98.66667	98.76923	98.64103	98.51282
	0.028	98.5641	98.84615	98.84615	98.66667	98.5641	98.76923	98.64103	98	98.25641	98.66667
	0.029	98.02564	98.64103	98.69231	98.74359	98.51282	98.76923	98.25641	98.46154	98.79487	98.5641
	0.03	98.69231	98.69231	98.87179	98.74359	98.71795	98.69231	98.66667	98.20513	98.46154	98.33333
	0.031	98.07692	98.4359	98.5641	98.76923	98.84615	98.38462	99.05128	98.79487	98.51282	98.20513
	0.032	98.4359	98.69231	98.79487	98.35897	98.23077	98.51282	98.41026	98.46154	98.61538	98.61538
	0.033	98.30769	98.61538	98.66667	98.58974	98.58974	98.41026	98.79487	98.64103	98.35897	98.48718
	0.034	98.48718	98.51282	98.89744	98.5641	98.71795	98.53846	98.28205	98.58974	98.51282	98.41026
	0.035	98.5641	98.74359	98.69231	98.5641	98.41026	98.71795	98.61538	98.41026	98.46154	98.38462
	0.036	98.64103	98.66667	98.76923	98.58974	98.76923	98.48718	98.23077	98.51282	98.94872	98.61538
	0.037	98.20513	98.76923	98.69231	98.5641	98.74359	98.61538	98.25641	98.69231	98.61538	98.30769
	0.038	98.15385	98.71795	98.74359	98.58974	98.92308	98.69231	98.53846	98.53846	98.25641	98.41026
	0.039	98.17949	98.76923	98.5641	98.82051	98.30769	98.28205	98.71795	98.28205	98.76923	98.02564
	0.04	98.66667	99.15385	98.87179	98.66667	98.64103	98.84615	98.53846	98.51282	98.53846	98.05128
	0.041	98.35897	98.84615	98.4359	98.84615	98.41026	98.4359	98.33333	98.33333	98.4359	98.35897
	0.042	98.71795	98.89744	98.64103	98.74359	98.76923	98.61538	98.71795	98.28205	98.71795	98.48718
	0.043	98.5641	98.94872	98.71795	98.46154	98.61538	98.71795	98.5641	98.66667	98.33333	98.71795
	0.044	98.23077	98.41026	98.74359	98.33333	98.64103	98.71795	98.53846	98.07692	98.48718	98.33333
	0.045	98.46154	98.69231	98.41026	98.64103	98.76923	98.41026	98.94872	98.5641	98.5641	98.05128
	0.046	98.82051	98.64103	98.25641	98.82051	98.53846	98.64103	98.71795	98.4359	98.28205	98.4359

	0.047	98.66667	98.4359	98.82051	98.74359	98.46154	98.48718	98.53846	98.07692	98.48718	98.69231
	0.048	98.64103	98.82051	98.71795	98.64103	98.28205	98.61538	98.5641	99.07692	98.23077	98.38462
	0.049	98.5641	98.76923	98.71795	98.41026	98.38462	98.20513	98.51282	98.51282	98.82051	98.48718
	0.05	98.35897	98.84615	98.66667	98.53846	98.64103	98.82051	98.28205	98.58974	98	98.69231
	0.051	98.38462	98.84615	98.4359	98.46154	98.38462	99.12821	98.28205	98.41026	98.53846	98.23077
	0.052	98.15385	98.76923	98.76923	98.82051	98.61538	98.5641	98.38462	98.5641	98.12821	98.12821
	0.053	98.23077	98.5641	98.82051	98.51282	98.61538	98.30769	98.4359	98.69231	98.38462	98.28205
	0.054	98.64103	98.82051	98.4359	98.87179	98.61538	98.74359	98.64103	98.4359	98.4359	98.35897
	0.055	98.61538	98.48718	98.84615	98.35897	98.66667	98.35897	98.66667	98.64103	98.69231	98.51282
	0.056	98.10256	98.71795	98.76923	98.92308	98.74359	98.20513	98.33333	98.58974	98.25641	98.5641
	0.057	98.79487	98.74359	98.71795	98.76923	98.79487	98.97436	98.33333	98.35897	98.38462	98.41026
	0.058	98.07692	98.48718	98.74359	98.76923	98.48718	98.46154	98.51282	98.92308	98.41026	98.41026
	0.059	98.46154	98.79487	98.89744	98.76923	98.61538	98.87179	98.58974	98.79487	98.61538	98.69231
	0.06	98.5641	98.84615	98.79487	98.79487	98.76923	98.51282	98.58974	98.33333	98.79487	98.69231
	0.061	98.71795	98.66667	98.69231	98.66667	98.5641	98.58974	98.33333	98.23077	98.35897	98.17949
	0.062	99	98.53846	98.66667	98.53846	98.92308	98.71795	98.84615	98.41026	98.33333	98.74359
	0.063	98.5641	98.87179	98.89744	98.38462	98.17949	98.74359	98.38462	98.58974	98.48718	98.58974
	0.064	98.28205	98.5641	98.33333	98.53846	98.76923	98.48718	98.66667	98.58974	98.5641	98.48718
	0.065	98.82051	98.66667	98.5641	98.64103	98.66667	98.69231	98.33333	98.35897	98.69231	98.61538
	0.066	98.66667	98.61538	98.76923	98.4359	98.23077	98.5641	98.12821	98.64103	98.66667	98.15385
	0.067	98.35897	98.79487	98.51282	98.35897	98.4359	98.5641	98.71795	98.4359	98.41026	98.17949
	0.068	98.69231	98.89744	98.35897	98.38462	98.58974	98.12821	98.28205	98.5641	98.71795	98.48718
	0.069	98.76923	98.74359	98.58974	98.46154	98.76923	98.25641	98.38462	98.5641	98.79487	98.5641
	0.07	98.79487	98.58974	98.66667	98.51282	98.4359	98.38462	98.58974	98.5641	98.51282	98.10256
	0.071	98.41026	98.71795	98.69231	98.20513	98.12821	98.41026	98.35897	98.58974	98.23077	98.58974
	0.072	98.66667	98.71795	98.46154	98.46154	98.76923	98.58974	98.61538	98.23077	98.07692	98.20513
	0.073	98.58974	98.84615	98.61538	98.82051	98.66667	98.61538	98.61538	98.35897	98.33333	98.74359
	0.074	98.66667	98.79487	99.02564	98.41026	98.84615	98.38462	98.38462	98.46154	98.5641	98.51282
	0.075	98.51282	98.76923	98.84615	98.66667	98.35897	98.71795	98.33333	98.51282	98.46154	98.33333
	0.076	98.92308	98.5641	98.48718	98.71795	98.38462	98.23077	98.48718	98.5641	98.35897	98.46154
	0.077	98.71795	98.58974	98.61538	98.74359	98.69231	98.76923	98.64103	98.05128	98.30769	98.41026
	0.078	98.66667	99.07692	98.69231	98.74359	98.66667	98.61538	98.61538	98.23077	98.35897	98.66667
	0.079	99.07692	98.89744	98.79487	98.79487	98.71795	98.48718	98.4359	98.41026	98.4359	98.51282
	0.08	98.82051	98.79487	98.38462	98.64103	98.64103	98.4359	98.48718	98.17949	98.41026	98.53846
	0.081	98.41026	98.69231	98.5641	98.89744	98.61538	98.58974	98.53846	98.69231	98.38462	98.17949
	0.082	98.66667	98.74359	98.48718	98.38462	98.25641	98.66667	98.64103	98.5641	97.87179	98.71795
	0.083	98.46154	98.71795	98.76923	98.35897	98.51282	98.38462	98.48718	98.4359	98.5641	98.23077
	0.084	98.61538	98.79487	98.64103	98.41026	98.74359	98.41026	98.58974	98.41026	98.35897	98.33333
	0.085	98.79487	98.64103	98.66667	98.61538	98.69231	98.71795	98.48718	98.28205	98.4359	98.41026
	0.086	98.74359	98.66667	98.33333	98.41026	98.76923	98.41026	98.33333	98.46154	98.71795	98.76923
	0.087	98.61538	98.46154	98.69231	98.64103	98.4359	98.82051	98.46154	98.66667	98.23077	98.84615
	0.088	98.38462	98.61538	98.74359	98.28205	98.58974	98.4359	98.48718	98.15385	98.51282	98.17949
	0.089	98.58974	98.82051	98.74359	98.61538	98.58974	98.48718	98.69231	98.69231	98.17949	98.5641
	0.09	98.5641	98.46154	98.30769	98.33333	98.69231	98.41026	98.5641	98.38462	98.71795	98.5641
	0.091	98.79487	98.82051	98.53846	98.48718	98.66667	98.61538	98.89744	98.38462	98.58974	98.15385
	0.092	98.82051	98.35897	98.51282	98.76923	98.76923	98.64103	98.53846	98.5641	98.17949	98.46154
	0.093	98.84615	98.5641	98.76923	98.82051	98.15385	98.23077	98.51282	98.35897	98.41026	98.53846
	0.094	98.76923	98.35897	98.41026	98.82051	98.38462	98.17949	97.82051	98.17949	98.51282	98.46154
	0.095	98.25641	98.61538	98.38462	98.10256	98.38462	98.76923	98.5641	98.25641	98.46154	98.4359
	0.096	98.46154	98.79487	98.69231	98.76923	98.82051	98.48718	98.20513	98.33333	98.38462	98.28205
	0.097	98.82051	98.61538	98.84615	98.79487	98.4359	98.61538	98.61538	98.12821	98.17949	98.64103
	0.098	98.69231	98.51282	98.84615	98.46154	98.41026	98.10256	98.33333	98.41026	98.48718	98.20513
	0.099	98.94872	98.48718	98.64103	98.41026	98.20513	98.28205	98.53846	98.30769	98.58974	98.53846