Project-2 of

"Introduction to Statistical Learning and Machine Learning"

1) Logistic Regression

1-1) Bayes' rule

According to the assumptions in 1-1), we have that:

1. Class variable $y \in \{0, 1\}$ which obeys Bernoulli distribution with parameter α, which means that $P(y = i) = \alpha^i * (1 - \alpha)^{1-i}$;
2. D-dimensional data vector $\vec{x} = (x_1, x_2, x_3, \dots, x_D)^T$, and with a given associated label y, each dimension of $\vec{x}$ are all intendent and obey the Gaussian distribution with parameter σ_i and μ_i (*i.i.d.*), which means that for each dimension x_i of $\vec{x}$, $f_{x_i}(x_i = x) = \frac{e^{-\frac{(x_i - \mu_{iy})^2}{2*{\sigma_i}^2}}}{\sqrt{2*\pi}*\sigma_i}$;
3. $\mu_i = \mu_{i1}\ if\ y == 1$ else μ_{i0}.

With Bayes' rule:

$$\mathrm{P}(B_i|\mathrm{A}) = P(B_i) * \frac{\mathrm{P}(\mathrm{A}|B_i)}{\sum_{j=all} P(B_j) * P(A|B_j)}$$

The classification probability $P(y = 1|\vec{x})$ can be computed as follow:

$$P(y = 1|\vec{x}) = P(y = 1) * \frac{P(\vec{x}|y = 1)}{\sum_{i=0}^{i=1} P(y = i) * P(\vec{x}|y = i)}$$

$$= \alpha * \frac{\prod_{i=1}^{i=D} P(x_i|y = 1)}{\alpha * \prod_{i=1}^{i=D} P(x_i|y = 1) + (1 - \alpha) * \prod_{i=1}^{i=D} P(x_i|y = 0)}$$

$$= \alpha * \frac{\prod_{i=1}^{i=D} f(x_i|y = 1)}{\alpha * \prod_{i=1}^{i=D} f(x_i|y = 1) + (1 - \alpha) * \prod_{i=1}^{i=D} f(x_i|y = 0)}$$

$$= \alpha * \frac{\prod_{i=1}^{i=D} \frac{e^{-\frac{(x_i - \mu_{i1})^2}{2*{\sigma_i}^2}}}{\sqrt{2 * \pi} * \sigma_i}}{\alpha * \prod_{i=1}^{i=D} \frac{e^{-\frac{(x_i - \mu_{i1})^2}{2*{\sigma_i}^2}}}{\sqrt{2 * \pi} * \sigma_i} + (1 - \alpha) * \prod_{i=1}^{i=D} \frac{e^{-\frac{(x_i - \mu_{i0})^2}{2*{\sigma_i}^2}}}{\sqrt{2 * \pi} * \sigma_i}}$$

$$= \frac{1}{1 + (1-\alpha) * \frac{\prod_{i=1}^{i=D} \frac{e^{-\frac{(x_i-\mu_{i0})^2}{2*{\sigma_i}^2}}}{\sqrt{2*\pi}*\sigma_i}}{\alpha * \prod_{i=1}^{i=D} \frac{e^{-\frac{(x_i-\mu_{i1})^2}{2*{\sigma_i}^2}}}{\sqrt{2*\pi}*\sigma_i}}}$$

$$= \frac{1}{1 + \frac{1-\alpha}{\alpha} * e^{\sum_{i=1}^{i=D}\left(\frac{{\mu_{i1}}^2-{\mu_{i0}}^2}{2*{\sigma_i}^2}+\frac{\mu_{i0}-\mu_{i1}}{{\sigma_i}^2}*x_i\right)}}$$

$$= \frac{1}{1 + e^{-\left(\sum_{i=1}^{i=D}\left(\frac{\mu_{i1}-\mu_{i0}}{{\sigma_i}^2}*x_i\right)+\left(\sum_{i=1}^{i=D}\left(\frac{{\mu_{i0}}^2-{\mu_{i1}}^2}{2*{\sigma_i}^2}\right)+\ln\left(\frac{\alpha}{1-\alpha}\right)\right)\right)}}$$

If we define $\vec{w}$ and b as follow:

$$\vec{w} = \left(\frac{\mu_{i1}-\mu_{i0}}{{\sigma_i}^2}\right)_{i=1,2,\dots,D}^{T}$$

$$b = \left(\sum_{i=1}^{i=D}\left(\frac{{\mu_{i0}}^2-{\mu_{i1}}^2}{2*{\sigma_i}^2}\right)+\ln\left(\frac{\alpha}{1-\alpha}\right)\right)$$

We can simplify the expression of $P(y=1|\vec{x})$ as follow:

$$P(y=1|\vec{x}) = \frac{1}{1+e^{-(\vec{w}^T\cdot\vec{x}+b)}} = Sigmoid(\vec{w}^T\cdot\vec{x}+b)$$

So it proves that under the assumption that *under the given label* y, *each dimension of data vector* $\vec{x}$ *are all independent, and obey the Gaussian distribution with parameter* μ_{iy} *and* σ_{iy}, *and* $\sigma_{i|y=1} = \sigma_{i|y=0}$ *for all the* i *vary from 1 to D,* the classification probability $P(y=1|\vec{x})$ can be written as the value of a sigmoid function $Sigmoid(\vec{w}^T\cdot\vec{x}+b)$, where the variable $\vec{w}$ and b is depend on the Gaussian parameter μ_{iy} and σ_{iy}, as well as the prior probability of label y: $P(y=1)=\alpha$. That makes using sigmoid function to predict an unknown sample in a 0-1 classification problem make sense.

1-2) Maximum Likelihood Estimation

If we set P_{right} as the probability of the classifier giving a correct classification. That is:

$$P_{right} = P(y = 1|\vec{x})\ if\ y == 1\ else\ P(y = 0|\vec{x})$$
$$= y * P(y = 1|\vec{x}) + (1 - y) * P(y = 0|\vec{x})$$

So to enhance the fitting ability, the goal is optimizing the variable $\vec{w}$ and b to get the maximum P_{right} over the independent data batch $D = \{(\vec{x}^1, y^1), (\vec{x}^2, y^2), \dots, (\vec{x}^N, y^N)\}$.

In another word, under the view of maximum likelihood estimation, we should maximum the probability of "the classifier would predict the same result as we already know". In mathematic expression, we should maximum the Maximum Likelihood Function under the independent assumption:

$$L(\vec{w}, b) = \ln(P_{D\ right})$$
$$= \ln\left(\prod_{i=1}^{i=N} P_{right}\right)$$
$$= \sum_{i=1}^{i=N} \ln(P_{right})$$
$$= \sum_{i=1}^{i=N} \ln\left(y * P(y = 1|\vec{x}) + (1 - y) * P(y = 0|\vec{x})\right)$$
$$= \sum_{i=1}^{i=N} y * \ln\left(P(y = 1|\vec{x})\right) + (1 - y) * \ln\left(P(y = 0|\vec{x})\right)$$
$$= \sum_{i=1}^{i=N} y * \ln(Sigmoid(\vec{w}^T \cdot \vec{x} + b)) + (1 - y)$$
$$* \ln(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b))$$

So now we should maximum $L(\vec{w}, b)$, or minimum $-L(\vec{w}, b)$. And so the cost function can be set as the average of $-L(\vec{w}, b)$:

$$E(\vec{w}, b) = Cost$$
$$= -\frac{1}{N}$$
$$* \left(\sum_{i=1}^{i=N} y * \ln\left(Sigmoid(\vec{w}^T \cdot \vec{x} + b)\right) + (1 - y)\right.$$
$$\left. * \ln\left(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b)\right)\right)$$

We can also derive the $Cost$ in another way. According to the associated knowledge in Theory of Information, if we consider $p = \{y^i\}$ as the

absolute distribution of y, and $q = \{P(y = 1|\vec{x})\}$ as the estimate distribution of y, what we should maximum is the ability that q fitting p. And the cross entropy function $cross\_entropy(p, q)$ happens to describe that ability. So what we should do is just maximum $cross\_entro \quad (p, q)$ or minimum $-cross\_entropy(p, q)$. So we have:

$$\begin{aligned} Cost &= -cros_{\ entropy(p,q)} \\ &= -\frac{1}{N} * \sum_{i=1}^{i=N} E_p(\ln(q)) \\ &= -\frac{1}{N} * \sum_{i=1}^{i=N} \sum_{(whole\ distribution)} p_i * \ln(q_i) \end{aligned}$$

The absolute distribution p can be describe as:

$$\begin{aligned} p: P_p(y = 1) &= y \\ P_p(y = 0) &= 1 - y \end{aligned}$$

So the $Cost$ is:

$$\begin{aligned} Cost = &-\frac{1}{N} * \sum_{i=1}^{i=N} (P_p(y = 0) * \ln\left(P_q(y = 0)\right) + P_p(y = 1) \\ &* \ln(P_q(y = 1))) \\ = -\frac{1}{N} * &\sum_{i=1}^{i=N} (y * \ln\left(Sigmoid(\vec{w}^T \cdot \vec{x} + b)\right) + (1 - y) * \ln(1 \\ &- Sigmoid(\vec{w}^T \cdot \vec{x} + b))) \end{aligned}$$

So the log-likelihood cost function can be explained in the view of maximum likelihood estimation as well as the theory of information. So it make sense to select log-likelihood function as the cost of the classification.

And the partial derivative of $\vec{w}$ and b also can be computed:

$$\nabla_b Cost = \frac{1}{N} * \sum_{i=1}^{i=N} \left(Sigmoid\left(\vec{w}^T \cdot \vec{x}^i + b\right) - y^i\right)$$

$$\nabla_{\vec{w}} Cost = \frac{1}{N} * \sum_{i=1}^{i=N} \left(Sigmoid\left(\vec{w}^T \cdot \vec{x}^i + b\right) - y^i\right) * \vec{x}^i$$

1-3) L2 Regularization

With Gaussian assumptions and the Gaussian prior probability placed on

the variables, we have that:

$$P(\vec{w},b|D) = (P(\vec{w},b) * P(D|\vec{w},b))/(\sum_{all\ \vec{w},b} P(\vec{w},b) * P(D|\vec{w},b))$$

Where the term $\sum_{all\ \vec{w},b} P(\vec{w},b) * P(D|\vec{w},b)$ doesn't depend on $\vec{w}$ and b, but only depends on the data set D and the distribution $\vec{w},b$ obey, in other words, depends on the D and λ under i.i.d. assumption.

So we set:

$$C(D,\lambda) = \frac{1}{\sum_{all\ \vec{w},b} P(\vec{w},b) * P(D|\vec{w},b)}$$

And so we have:

$$P(\vec{w},b|D) = \big(P(\vec{w},b) * P(D|\vec{w},b)\big) * C(D,\lambda)$$

Under the i.i.d. assumption:

$$P(\vec{w},b) = \left(N\left(\frac{0,1}{\lambda}\right)\right)^{D+1}$$

$$= \prod_{all} \frac{\lambda}{\sqrt{2*\pi}} * e^{-\lambda^2 * \frac{w_i{}^2}{2}} * \frac{\lambda}{\sqrt{2*\pi}} * e^{-\lambda^2 * \frac{b^2}{2}}$$

$$= \left(\frac{\lambda}{\sqrt{2*\pi}}\right)^{D+1} * e^{-\lambda^2 * \frac{b^2}{2}} * \prod_{all} e^{-\lambda^2 * \frac{w_i{}^2}{2}}$$

$$P(D|\vec{w},b) = P(y|X,\vec{w},b)$$
$$= P_{right}(D)$$
$$= \prod_{all} (P(y=1|\vec{x})\ if\ y == 1\ els\ \ P(y=0|\vec{x}))$$
$$= \prod_{all} y * P(y=1|\vec{x}) + (1-y) * P(y=0|\vec{x})$$

So in conclusion, we have:

$$P(\vec{w},b|D) = \left(e^{-\lambda^2 * \frac{b^2}{2}} * \prod_{all} e^{-\lambda^2 * \frac{w_i{}^2}{2}} * \prod_{all} y * P(y=1|\vec{x}) + (1-y) * P(y=0|\vec{x})\right) * C(D,\lambda)$$

$$= \left(e^{-\lambda^2 * \frac{b^2}{2}} * \prod_{all} e^{-\lambda^2 * \frac{{w_i}^2}{2}} * \prod_{all} y * Sigmoid(\vec{w}^T \cdot \vec{x} + b) + (1 - y) * (1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b)) \right) * C(D, \lambda)$$

And if we ignore $C(D, \lambda)$, then we have:

$$P(\vec{w}, b|D) \propto \left(e^{-\lambda^2 * \frac{b^2}{2}} * \prod_{all\ i} e^{-\lambda^2 * \frac{{w_i}^2}{2}} * \prod_{all\ i} y^i * Sigmoid(\vec{w}^T \cdot \vec{x}^i + b) + \left(1 - y^i\right) * (1 - Sigmoid(\vec{w}^T \cdot \vec{x}^i + b)) \right)$$

Just as what we have discussed, we should know "the variable which is of maximum-likelihood according to the data", that is to maximum $P(\vec{w}, b|D)$ to fit the data better. So we can define the $Cost$ to be $-\ln(P(\vec{w}, b|D))$:

$$Cost = -\ln(P(\vec{w}, b|D))$$

$$= -\ln\left(\left(e^{-\lambda^2 * \frac{b^2}{2}} * \prod_{all} e^{-\lambda^2 * \frac{{w_i}^2}{2}} * \prod_{all} (y * Sigmoid(\vec{w}^T \cdot \vec{x} + b) + (1 - y) * \left(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b)\right) \right) * C(D, \lambda) \right)$$

$$= -\sum_{i=1}^{i=N} (y * \ln\left(Sigmoid(\vec{w}^T \cdot \vec{x} + b)\right) + (1 - y) * \ln(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b))) + \sum_{i=1}^{i=D} \lambda^2 * \frac{{w_i}^2}{2} + \lambda^2 * \frac{b^2}{2} + Cont(D, \lambda)$$

Where $Cont(D, \lambda) = -\ln(C(D, \lambda))$.

To make the equation more simple and more easy to compute and to validate the hyper-parameter, we just change the expression form of that equation:

$$Cost = -\frac{1}{N} * \sum_{i=1}^{i=N} (y * \ln(Sigmoid(\vec{w}^T \cdot \vec{x} + b)) + (1 - y) * \ln(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b))) + \lambda/2 * \sum_{i=1}^{i=D} {w_i}^2 + \lambda * \frac{b^2}{2} + Cont(D, \lambda)$$

And that is the cost function under the Gaussian assumptions (which is always true in most machine learning problems) and Gaussian prior probability.

The equation introduces a hyper-parameter λ into the $Cost$. λ controls the prior probability of $\vec{w}$ and b, with a larger λ, $\vec{w}$ and b tend to converge around zero, in other words, tend to have small value, and on the other hand, have big one. It also can been concluded from $Cost$, when we minimum the $Cost$, the term $-\frac{1}{N} * \sum_{i=1}^{i=N} (y * \ln(Sigmoid(\vec{w}^T \cdot \vec{x} + b)) + (1 - y) * \ln(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b)))$ forces the model to fit the data as hard as it can, and the term $\lambda/2 * \sum_{i=1}^{i=D} {w_i}^2 + \lambda * \frac{b^2}{2}$ forces the model to have small values of $\vec{w}$ and b. The latter one, in fact, introduces some error into the optimization, forces the model to fit the data as well as it can meanwhile keep small $\vec{w}$ and b. So the model would make a choice between the both. The result is that the model tends to fit only the main distribution of the data and omit the noises. And that is what we want, which can enhance the robustness and prevent model from over-fitting.

Because only $\vec{w}$ would directly multiply with the data and learn from data, and b is just a threshold. There is no need to penalize b, and also, the meaning of $Cost$ is to update $\vec{w}$ and b, so $Cont(D, \lambda)$ is useless because there is no link between it and $\vec{w}$ and b. So the cost function we use is:

$$Cost = -\frac{1}{N} * \sum_{i=1}^{i=N} (y * \ln(Sigmoid(\vec{w}^T \cdot \vec{x} + b)) + (1 - y) * \ln(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b))) + \lambda/2 * \sum_{i=1}^{i=D} {w_i}^2$$

And the partial derivative of $\vec{w}$ and b also can be computed:

$$\nabla_b Cost = \frac{1}{N} * \sum_{i=1}^{i=N} \left(Sigmoid\left(\vec{w}^T \cdot \vec{x}^i + b\right) - y^i\right)$$

$$\nabla_{\vec{w}} Cost = \frac{1}{N} * \sum_{i=1}^{i=N} \left(Sigmoid\left(\vec{w}^T \cdot \vec{x}^i + b\right) - y^i\right) * \vec{x}^i + \lambda * \vec{w}$$

2) Digit Classification

2-1) K-Nearest Neighbors

In the feature space, the data space's sample points' characteristics are utterly represented by their feature vectors. So the distance between two features vectors represents how "far" the two sample points associated are in the feature space, or in other words, how similar they are according to the features we are focusing on. Generally speaking, we would think the similar sample points may have great probability to have the same label in a smooth feature space.

So one simple idea to predict an unknown sample point is to find the nearest or the most similar point whose label has been already known, and take its label as the prediction. And that method is called "The Nearest Neighbor". It is a simple method and sometimes performs OK, but it is time-consuming and space-consuming because in fact it doesn't "learn" but just "remember" all the data point. When predicting a new sample point, it should compute the distances of all the points in memory. When the training data set is big, that would take much more space and time.

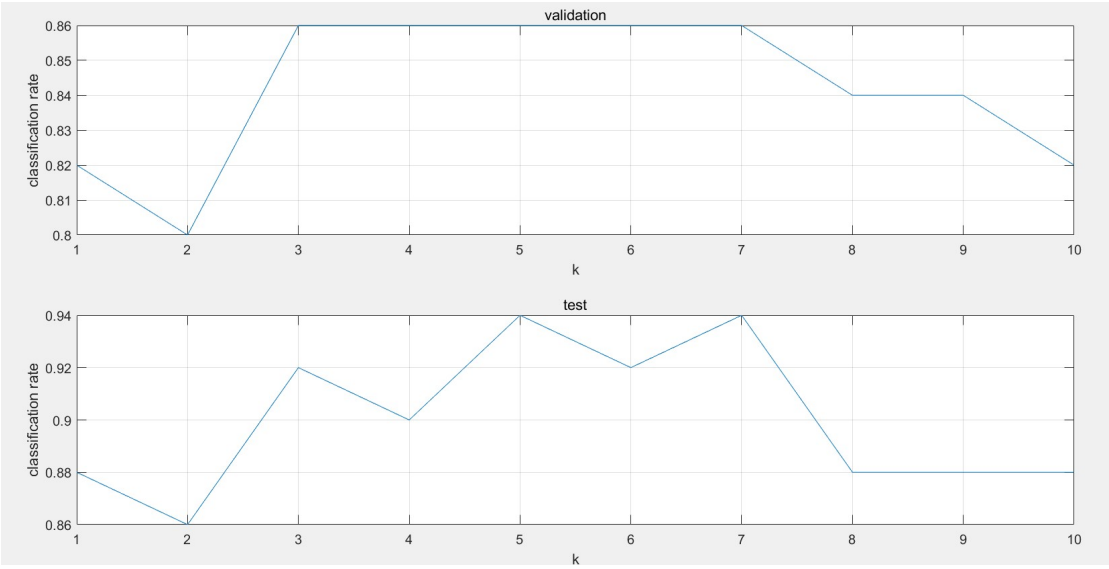
It is common that the feature space is sharp in some directions at some areas, with small distance between two different labels, and that would make "The Nearest Neighbor" make mistakes. And also, the method would perform poorly when there are noises in the training data because it would remember the noises and predict according to the noises. So to enhance the robustness to the feature space and noises, prevent the model from over-fitting, we can select K nearest points and let them vote to decide the prediction, in other words, take the "average label" as the prediction. And this method is called "K-Nearest Neighbors".

K should be regarded as a hyper-parameter, with a small K the model would remember to many noises and performs poorly when the feature space is

not so smooth, while with a large K, the model tends to make the same prediction (the average label among the whole training data), and lose the ability to fit the real model, get trouble with under-fitting. So the value of K should be validated among a validation set to pick the best one. Here we won't use k-fold cross validation as usual, because for K-NN, the model itself is the input data, and when using cross validation, changing the input data is in fact changing the model within every loops. And that makes no sense. We just use one training data set to train, and one validation data set to validate. Of course it would somewhat over fit the validation data.

Another hyper-parameter of K-NN is the distance. There are varies of distance to choose from, and in different problem, the best choice might be different. In the Digit Classification problem, we choose Euclidian Distance or L-2 distance.

(The codes (KNN_cross_validation.m, run_KNN_cross_validation.m) for K-NN and their validation has been added to attachment. Only the result would been shown here.)



The highest classification rate on validation set is 86%, and the $K = \{3,4,5,6,7\}$, and the highest rate on test set is 94%, the $K = \{5, 7\}$.

Considering its simplicity, the result is encouraging, and as speculated before, the model performs poorly when K is either too low or too high. And because K-NN highly depend on the distribution of input data, so the classification rate would also highly depend on the distribution of input and validation (or test) data, and that means the details of the result should be explained by the details of the three data sets' distribution. In general, it can be speculated that maybe the test data's distribution is more similar to the

train's, and may have more noises because the curve is rough.

According to the validation result, the best K should be chosen from $\{3,4,5,6,7\}$, and I would choose the median K=5. And the test accuracy of K=5 happened to be the best, 94%. It is presupposed that the test accuracy of K=5 would be great, but maybe not the best, because the distributions of validation and test set would be roughly similar, but not exactly the same. And that would be proved when we focus on the accuracy of $K = \{3,4,6,7\}$, with the validation accuracy to be 86% while the test: 92%, 90%, 92%, 94%: they are not corresponded.

2-2) Logistic Regression

It has already been proved in 1-1) that with Gaussian assumption, the classification probability can be computed by:

$$P(y|\vec{x}) = Sigmoid(\vec{w}^T \cdot \vec{x} + b)$$

Where the variables $\vec{w}$ and b depend on the distributions of $\vec{x}$ and the prior probability of y. So we can make predictions according to $P(y|\vec{x})$ and learn the variable to let the fitting error (cost) to be small and the model to be similar to the distribution of training data using Gradient Descent to minimum the cost, rather than compute the variable according to the distribution. According to 1-2), the cost function we should minimum is:

$$COST = -\frac{1}{N} * \sum_{i=1}^{i=N} (y * \ln(Sigmoid(\vec{w}^T \cdot \vec{x} + b)) + (1 - y) * \ln(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b)))$$

And the gradient:

$$\nabla_b Cost = \frac{1}{N} * \sum_{i=1}^{i=N} (Sigmoid(\vec{w}^T \cdot \vec{x}^i + b) - y^i)$$

$$\nabla_{\vec{w}} Cost = \frac{1}{N} * \sum_{i=1}^{i=N} (Sigmoid(\vec{w}^T \cdot \vec{x}^i + b) - y^i) * \vec{x}^i$$

To make the procedure of gradient descent faster and more efficient, I transformed the equations to the matrix form. With the assumption mentioned in codes, there are:

$$P(y|\vec{x}) = Sigmoid(X * W)$$

$$Cost = -\frac{1}{N} * sum(\vec{y} .* \ln(P(y|\vec{x})) + (1 - \vec{y}) .* \ln(1 - P(y|\vec{x})))$$

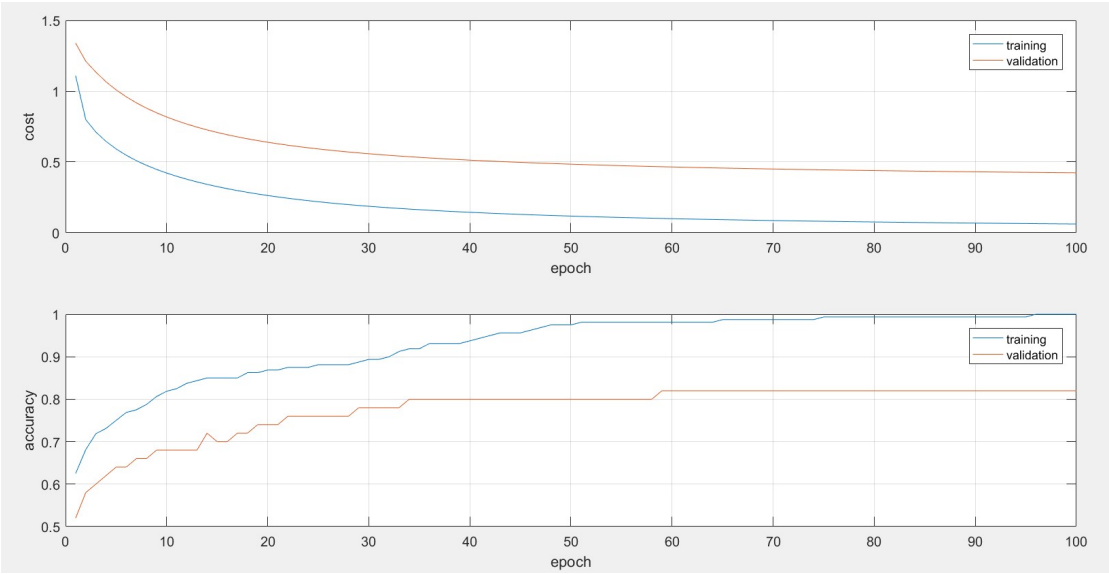
$$\nabla_W Cost = \frac{1}{N} * X^T * (P(y|\vec{x}) - \vec{y})$$

When doing gradient descent, there are three main hyper-parameters to been chosen, the learning rate, the number of iterations and the way to initial the variable W. The analysis of the three parameter are as follow:

1. The learning rate: It controls the step size the model takes towards the minimum point of the cost function. With small learning rate, it would take a long time for model to converge while if the rate is too large, the model may end vibrating around the minimum point, and even fail to converge and get a high cost and low accuracy. So the performance would be poor when rate is too high, and too low if the number of iteration is not enough.
2. The number of iteration: It would work together with the learning rate, a low number of iteration may make model stop learning too early, while a really large one may make model learn the training data again and again and lead to its fitting the noises of the data, improving the performance on training data but weakening the generalization ability, and thus, perform poorly on validation and test data. (But it turns out that when we want the model to over fit, the number of iteration should be much larger, and the affect will be easily eliminated by the regularization term lambda.)
3. The way to initial the variable W: The initialization plays a great role in the performance of a model. A great initialization should initial the model to a point near to the global minimum, and where the gradient is high so that model would learning fast, at least won't saturate and stop learning at first time. Generally, there are three ways: uniform distribution, Gaussian distribution with $\mu = 0$ and $\sigma = 1$, Gaussian distribution with $\mu = 0$ and σ is a smaller one. Theoretically, because of the Gaussian assumption we make, the W obeys a Gaussian distribution with $\mu = 0$, it is more suitable to initial W with Gaussian distribution, and that would make the model fit the distribution better and thus nearer to the minimum, compared with uniform distribution. While if the σ is large, $X * W$ would obey a Gaussian distribution with a much larger σ, which means the bandwidth of $X * W$ would be large and there is a large probability that $X * W$ is either too small or too high, leading the sigmoid function to saturate and the model to stop learning. So it would takes a long time for the model to start to converge. A better way to initialize is initial the W by a Gaussian distribution with a smaller σ, keep the σ of $X * W$ to be small (maybe 1 is suitable) and the sigmoid function to be active at first time. So I finally choose a Gaussian distribution with $\sigma = 1/(1 + D)$, D is the dimention of $\vec{x}$, to make the σ of $X * W$ equals to one.

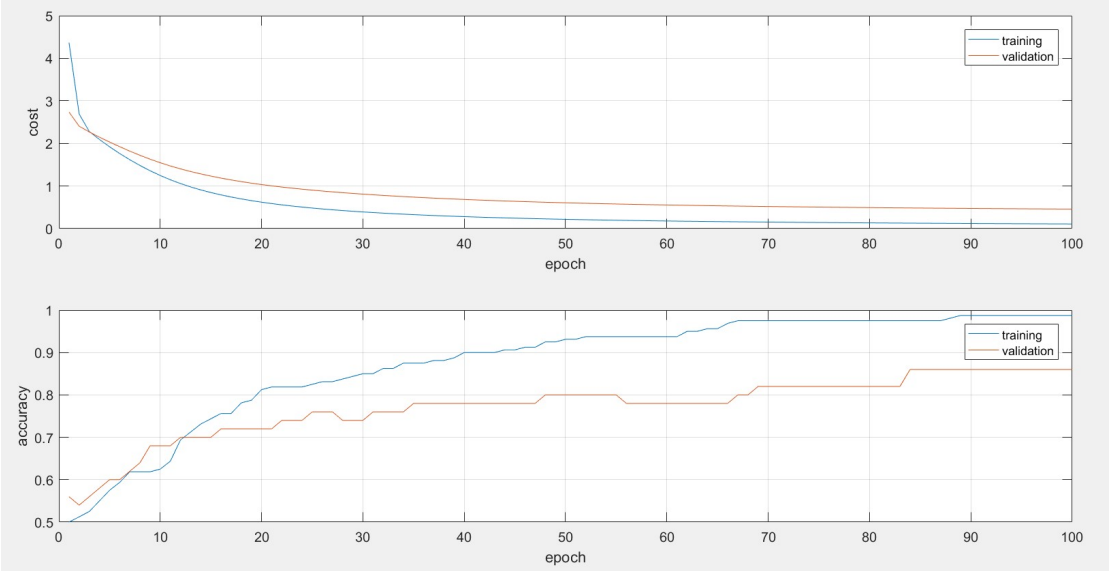
(The codes (logistic.m, logistic_predict.m, evaluate.m, logistic_regression_template.m) for logistic regression and their validation has been added to attachment. Only the result would been shown here.)

Initial with uniform distribution:



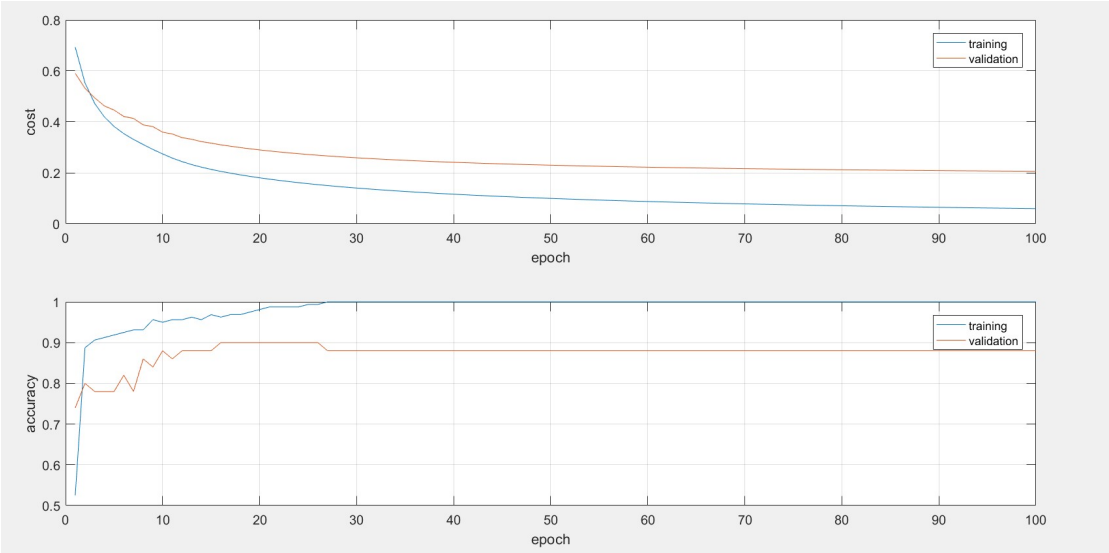
The TRAIN CE is 0.062283, TRAIN FRAC is 100.00%, VALIC_CE is 0.423857, VALID FRAC is 82.00%;

Initial with Gaussian distribution with $\mu = 0$ and $\sigma = 1$:



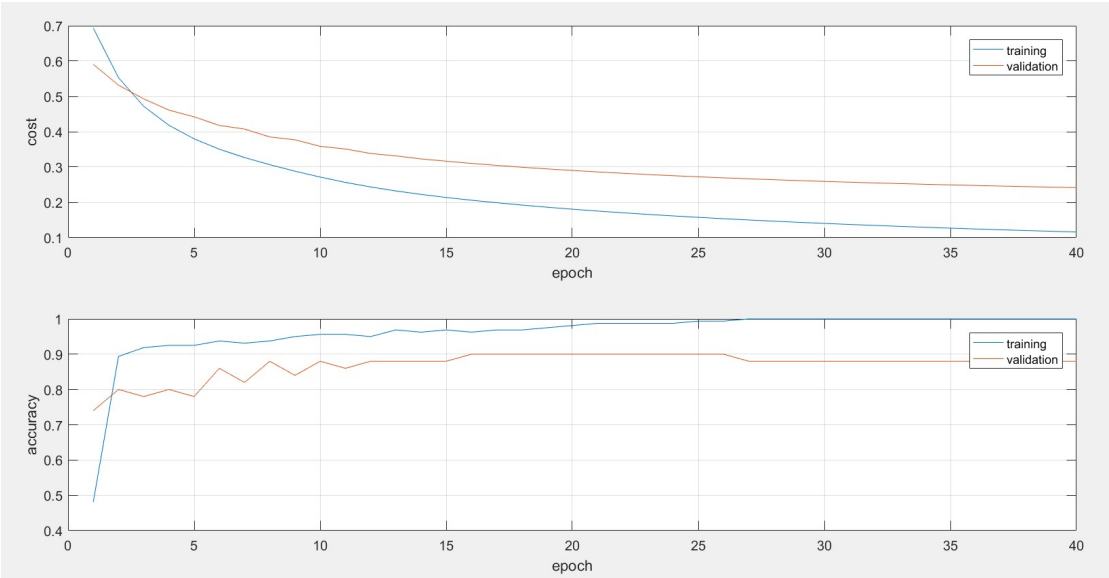
The TRAIN CE is 0.108657, TRAIN FRAC is 98.75%, VALIC_CE is 0.457515, VALID FRAC is 86.00%;

Initial with Gaussian distribution with $\mu = 0$ and $\sigma = 1/(1 + D)$:



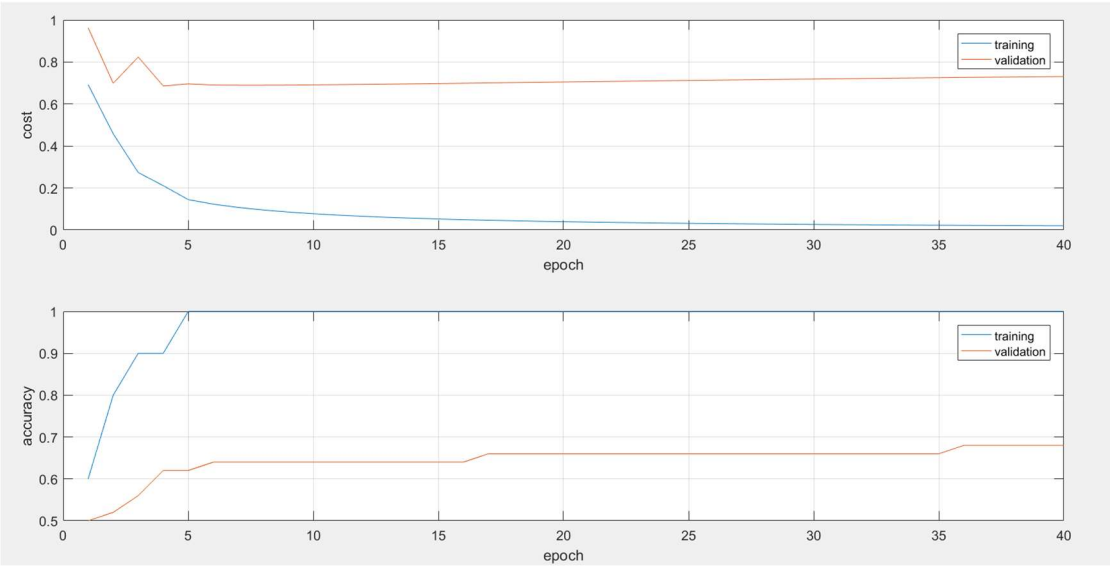
The TRAIN CE is 0.059544, TRAIN FRAC is 100.00%, VALIC_CE is 0.205965, VALID FRAC is 88.00%.

The results of the experiments confirmed my theory before. Finally I choose to initial with Gaussian distribution with $\mu = 0$ and $\sigma = 1/(1 + D)$, number of iteration is 40, learning rate is 0.3.



And the TRAIN CE is 0.116113, TRAIN FRAC is 100.00%, VALIC_CE is 0.241681, VALID FRAC is 88.00%. And on test set, the TEST_CE is 0.222316, VALID FRAC is 92.00%.

And on the small training data set:



The TRAIN CE is 0.019805, TRAIN FRAC is 100.00%, VALIC_CE is 0.731007, VALID FRAC is 68.00%.

I ran several times per model, while there were only a little differences between the results. I think it is because I didn't give too much randomness to the model when initializing, which means the range I allowed the model to initialize itself is not too large. Because that the randomness of the model mainly comes from initialization (especially when use gradient descent for the whole data set, not for a batch), it is reasonable that there is not too much randomness in the result. While if so, I think I would narrow the range down and take the average performance.

Another thing worth to be noticed is that we get very low cost on training data set and very high accuracy (100%), while on test and validation set, the performance is not so good. That means the model is very good at fitting but not at generalizing, maybe has learnt the noises in the training data. And that indicated that regularization is necessary to prevent over-fitting problem.

2-3) Penalized Logistic Regression

According to 1-3), under the Gaussian assumption, we have:

$$Cost \quad -\frac{1}{N} * \sum_{i=1}^{i=N} (y * \ln(Sigmoi \ \ (\vec{w}^T \cdot \vec{x} + b)) + (1 - y) * \ln(1 - Sigmoid(\vec{w}^T \cdot \vec{x} + b))) + \lambda/2 * \sum_{i=1}^{i=D} {w_i}^2$$

$$\nabla_b Cost = \frac{1}{N} * \sum_{i=1}^{i=N} \left(Sigmoid\left(\vec{w}^T \cdot \vec{x}^i + b\right) - y^i\right)$$

$$\nabla_{\vec{w}} Cost = \frac{1}{N} * \sum_{i=1}^{i=N} \left(Sigmoid\left(\vec{w}^T \cdot \vec{x}^i + b\right) - y^i\right) * \vec{x}^i + \lambda * \vec{w}$$

And in matrix form, we have:

$$P(y|\vec{x}) = Sigmoid(X * W)$$

$$Cost = -\frac{1}{N} * sum\left(\vec{y}.* \ln\left(P(y|\vec{x})\right) + (1 - \vec{y}).* \ln\left(1 - P(y|\vec{x})\right)\right) + \frac{\lambda}{2} * sum(W'.\hat{}2)$$

$$\nabla_W Cost = \frac{1}{N} * X^T * (P(y|\vec{x}) - \vec{y}) + \lambda * W'$$

Where W' is the vector $(w_1, w_2, w_3, \dots, w_D, 0)^T$.

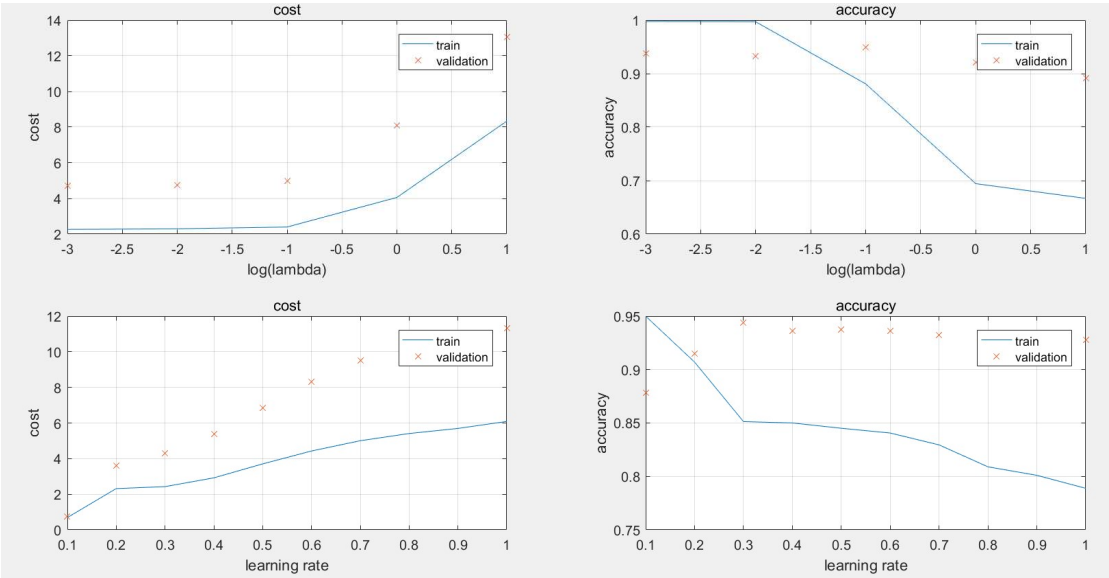
As discussed in 1-3), there is another hyper-parameter when doing gradient descent: λ. When the λ is small, the model tends to fit the noises in input data, end in a great ability to fit but a poor ability to generate. While when the λ is too large, the model would force all the w_i to be zero, that would destroy the model's ability to fit as well as the ability to generate.

So in all, there are two main hyper-parameter to be chosen: λ and learning rate η. Here I wouldn't follow the method before, because it would cause the model overfitting the validation data. To make full use of the training data and validation data, I choose to use K-fold cross validation. This method would prevent over fitting and meanwhile, for every model we run K times and average the result, that would reduce the influence randomness has on the result and makes the result more convictive. The theory and the details of K-fold cross validation have been illustrated in *"Project 1".* And thereis no need to spell out again.

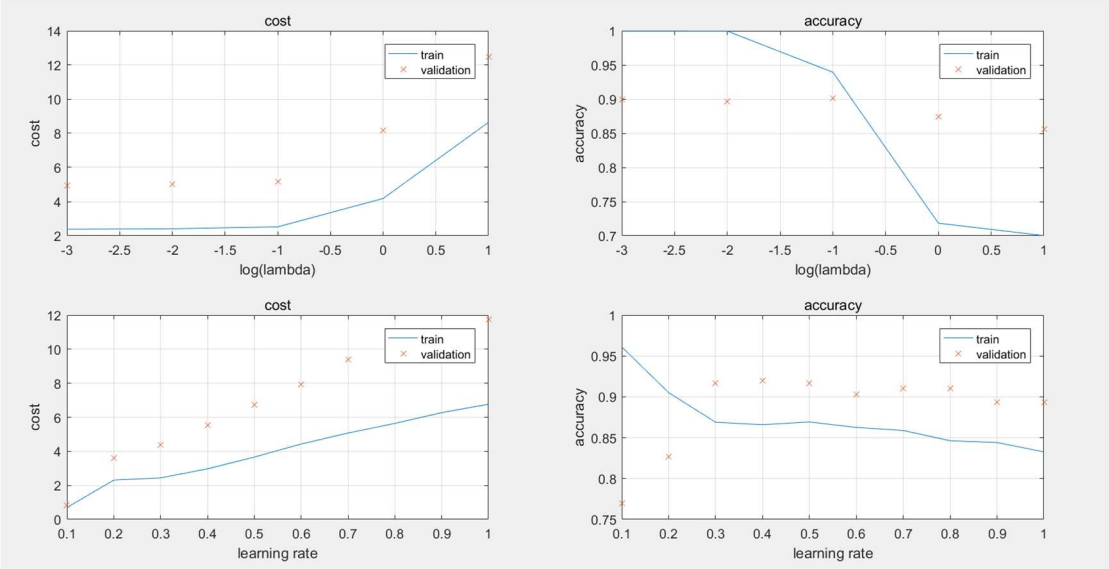
(The codes (logistic_pen.m, K_fold_cross_validation.m, Train.m, logistic_regression_template_pen.m) for logistic regression and their validation has been added to attachment. Only the result would been shown here.)

According to the demand, I chose $K = 10$.

And here is the result:



And on the small training data set:

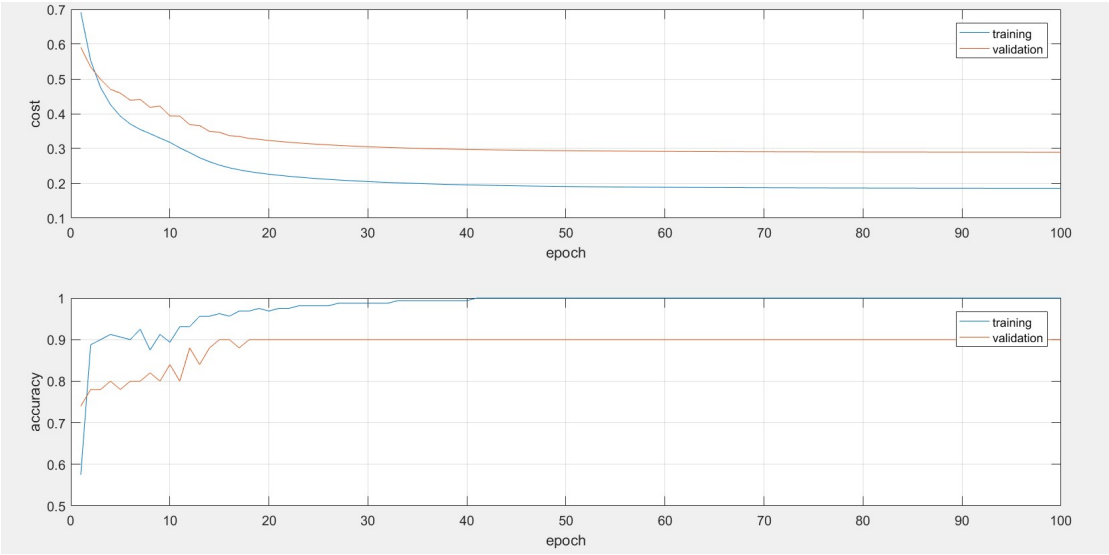


Just as what we deduct before, when learning rate is very small, the cost of training and validation data would be high and accuracy would be low because model converge slowly and there is not enough number of iteration; and when it is too high the cost would be high and accuracy would be low again because model fails to get to a lower point and even fails to converge. And the best choice of learning rate should be neither too large nor too low. 0.3 proves to be a good one.

As for the regularization term λ, we have known that it introduces some noises and prevent the model from fitting the data too hard, so with greater λ, the model would have lower ability to fit, and thus higher cost and lower accuracy on training data; while λ is small, the model may suffer from the over-fitting problem just as stated before, so the ability to fit is great but to

generalize is poor, which lead to low accuracy and high cost on validation, and when λ is too high, the model would suffer from under-fitting problem, can't even learning the distribution of the data and of course performs poorly on validation data, with high cost and low accuracy. So a middle value of λ is preferred. And 0.1 would perform well.

So the chosen pair of hyper-parameter is $(\lambda, \eta) = (0.1, 0.3)$.



And the TRAIN CE is 0.224550, TRAIN FRAC is 99.38%, VALIC_CE 0.289719, VALID FRAC is 90.00%.

And on test set, the TEST_CE is 0.269982, VALID FRAC is 92.00%.

Compared with the result of model without penalty term, the model with penalty performs better or as well as the one without penalty on validation set and test set, bet poorer on training set. Just as we expected, penalized the ability to fit the training data and enhance the ability to generate.

2-4) Naïve Bayes

The Naïve Bayes Classifier is another model to do classification. Different from the former models, Naïve Bayes classifier based firmly on the statistic equations and is one of the classic method in statistic learning. It has high interpretability.

What we want is the conditional probability $P(c|\vec{x})$, and according to the Bayes' Law, we have:

$$P(c|\vec{x}) = \frac{P(c) * P(\vec{x}|c)}{\sum_{i=1}^{i=C} P(i) * P(\vec{x}|i)}$$

So the learning procedure is just to learn the prior probability $P(c)$ for each class c, and the likelihood probability $P(\vec{x}|c)$.

We can compute or estimate $P(c)$ with the frequency of each class in training data, and for $P(\vec{x}|c)$, under the Gaussian assumption, we have that under a given class c, each x_i are all independent and obey the Gaussian distribution with parameter μ_{ci} and σ_{ci}. So we have:

$$P(\vec{x}|c) = \prod_{i=1}^{i=D} P(x_i|c)$$

$$= \prod_{i=1}^{i=D} \frac{1}{\sqrt{2*\pi}*\sigma_{ci}} * e^{-\frac{(x_i-\mu_{ci})^2}{2*{\sigma_{ci}}^2}}$$

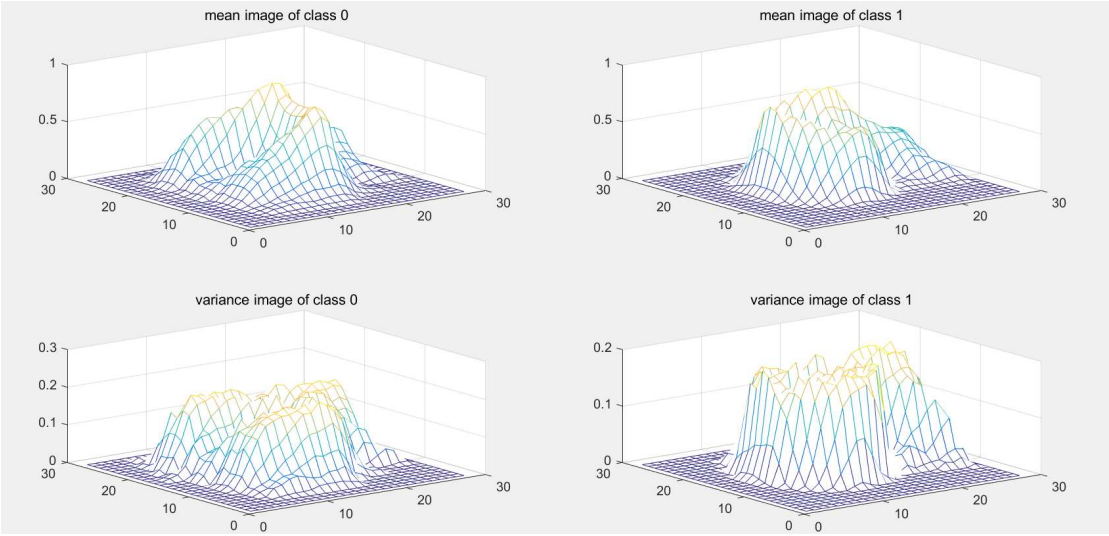
And then we have:

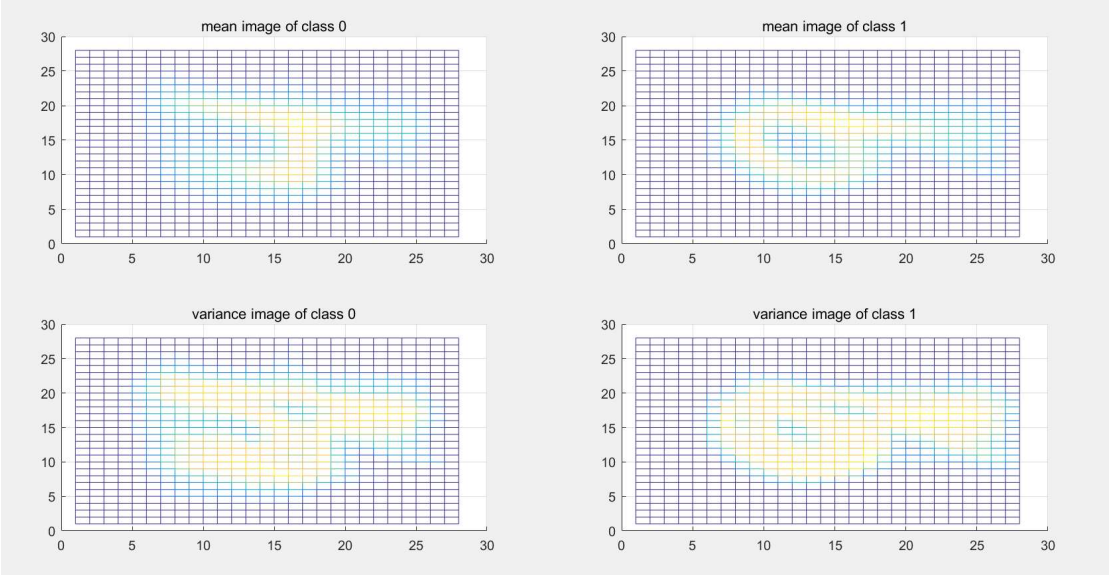
$$P(c|\vec{x}) = \frac{P(c) * \prod_{i=1}^{i=D} \frac{1}{\sqrt{2*\pi}*\sigma_{ci}} * e^{-\frac{(x_i-\mu_{ci})^2}{2*{\sigma_{ci}}^2}}}{\sum_{j=1}^{j=C} P(j) * \prod_{i=1}^{i=D} \frac{1}{\sqrt{2*\pi}*\sigma_{ji}} * e^{-\frac{(x_i-\mu_{ji})^2}{2*{\sigma_{ji}}^2}}}$$

And all μ_{ci} and σ_{ci} here can be estimated by the sample mean and sample standard deviation, according to the theory of parameter estimation. So all the training work we should do is to select the associated sample from training data, according to class c and dimension i, and compute the sample mean and sample standard deviation, as well as the frequency of each class in training data.

(The codes (run_nb.m) for logistic regression and their validation has been added to attachment. Only the result would been shown here.)

After running the Naïve Byers Classifier, I got training accuracy to be 86.25%, and test accuracy to be 80%.





It is not a very good result compared with the former models, I think it is because we set too many "hard" assumptions and rules that the model should obey, (like we set that $P(\vec{x}|c)$ obeys the Gaussian distribution) and that would restrict the model, making the model hard to learn by itself, according to the real distribution of the data. And in addition, the learning procedure depend too much on the sample and have less robustness and less ability to generate.

As for the visualization, we can clearly see a "4" and a "9" out of the mean of the images, it is natural because we just average all the images in class "4" and class "9". And the variance indicates that along the strokes the variance is high, it is also natural because all the image have the same framework ("4" and "9"), but have differences in details along the strokes.

2-5) Compare K-NN, Logistic Regression and Naïve Bayes

In conclusion, we got the test accuracy to be 94% with K-NN, 92% with logistic regression, and 80% with Naïve Bayes.

Naïve Bayes is different from the other two, it firmly based on statistic theories, and it “compute” or “estimate” rather than “learn”. It has high interpretability but also too many restrictions, so it is theoretically perfect, while performs not so well. The other two are weak in interpretability, but with high freedom, the model can learn better. It supposed that logistic regression would perform better than K-NN, because K-NN seems to be so simple to fit complex data, and with many tricks like regularization, logistic regression would win the competition, while it is not. I think one reason is that there is still a big potentiality for logistic regression, using more tricks like multi-layers, batch normalization, convolution layers and so on. And it has been proved that with these tricks, logistic regression would perform further better even facing much more difficult problems. And another reason is that the problem we are handling is simple enough for K-NN to get a rather great performance. The difference between K-NN and logistic regression is that K-NN just “remember” the data, not “learn” from the data, while logistic regression focuses on “learning” from the given data and tring to conclude the main distribution out of the data.