CSC311 A2 Writeup

Richard Yan

October 2022

1 Expected Loss and Bayes Optimality

(a) The policy that keeps every email will give: $\mathbb{E}[\mathcal{J}(y,t)] = 0.2 * 1 + 0.8 * 0 = 0.2$,
where the policy that remove every email will give: $\mathbb{E}[\mathcal{J}(y,t)] = 0.2 * 0 + 0.8 * 500 = 400$.

(b) We can then combine the conditional probability with the loss function, so we can calculate the expected loss using the conditional probability, and as we minimize the expected loss, we can get a threshold to apply to predict y_*.

So if we keep email when we observe feature $\mathbf{x}$, the expected loss is $P(t = \text{spam}|\mathbf{x}) * 1 + 0 * (1 - P(t = \text{spam}|\mathbf{x})) = P(t = \text{spam}|\mathbf{x})$, while if we remove, the expected loss is $P(t = \text{spam}|\mathbf{x}) * 0 + 500 * (1 - P(t = \text{spam}|\mathbf{x})) = 500 - 500P(t = \text{spam}|\mathbf{x})$.
So we should remove if the second expected loss is smaller than the first one, so:

$$\begin{aligned} 500 - 500P(t = \text{spam}|\mathbf{x}) &< P(t = \text{spam}|\mathbf{x}) \\ 501P(t = \text{spam}|\mathbf{x}) &> 500 \\ P(t = \text{spam}|\mathbf{x}) &> \frac{500}{501} \end{aligned}$$

Hence we should remove when $P(t = \text{spam}|\mathbf{x}) > \frac{500}{501}$, and keep if less.

(c) We can then compute the conditional probability $P(t|x_1, x_2)$, which involve using the two given joint distribution $P(x_1, x_2|t)$ and marginal distribution $P(x_1, x_2)$ which we can get by using the same given joint distribution and $P(t)$ given, then marginalize.

Finally, we combine this conditional probability $P(t|x_1, x_2)$ with loss function again, and calculate the expected loss with each possible decision (i.e. keep or remove when $x_1 = 0, x_2 = 1$, etc.).

The calculated conditional probability $P(t|x_1, x_2)$ is:

	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0.1	0.97
$x_2 = 1$	0.96	1

With the inequality given in part (b), we should only remove when both feature values are 1.

(d) The expected loss will be:

$$\begin{aligned}\mathbb{E}[\mathcal{J}(y,t)] &= P(\text{spam, keep}) * 1 + P(\text{non-spam, remove}) * 500 \\ &= P(\text{keep} \mid \text{spam})P(\text{spam}) * 1 + P(\text{remove} \mid \text{non-spam})P(\text{non-spam}) * 500 \\ &= (1 - 0.12)(0.2) * 1 + (0)(0.8) * 500 \\ &= 0.176\end{aligned}$$

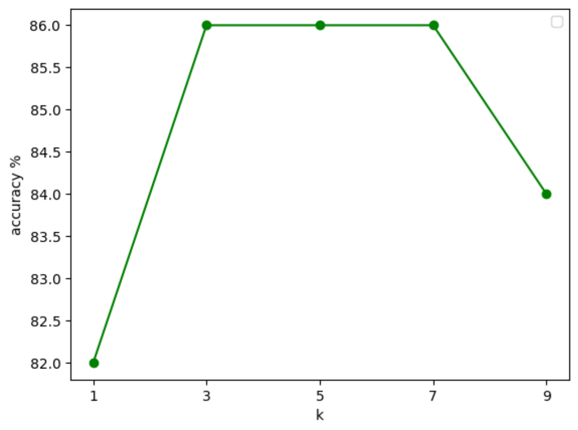
2 Feature Maps

(a) We can use the idea of half-plane and convexity. So we can observe that the three data points all lies on a straight line, but we can argue that suppose the data-set is linearly separable, then the two data points that classifies as 1 will lie on the same half-plane, so that half-plane must include the line that pass through this two points. However, since the other data point that classifies as 0 also lie on the same line, which, will lie on the same half-plane with other two data points, which give contradiction.

3 kNN vs. Logistic Regression

1. **k-Nearest Neighbors**

(a)



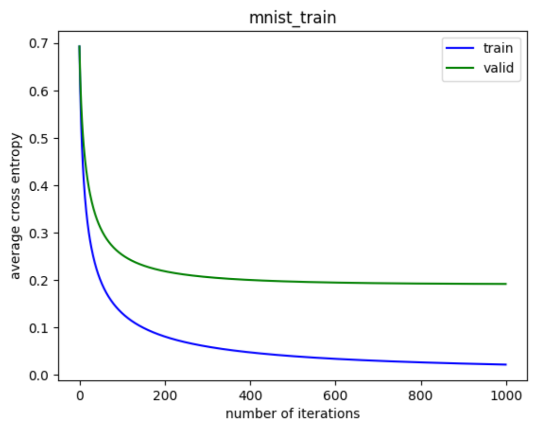
(b) Let $k^* = 5$. On validation it gets 86% accuracy, and on test it gets 94% accuracy.
For $k = 3$, validation: 86%, test: 92%.
For $k = 7$, validation:86%, test: 94%.
In this case, the validation and test accuracy didn't change that much, while $k = 3$ perform a little bit worse than $k = 5$ when testing. However, as I also peek at $k = 1$ and $k = 9$, both of them seems to be worse than $k = 5$, where small k gives 82%/88%, and large k gives 84%/88%.

2. **Logistic Regression**

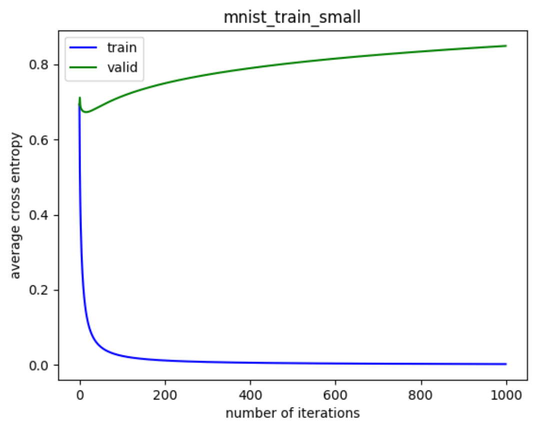
(b) The best hyperparameter settings I found was with learning rate 0.1, weight regularization 0, and num iteration 1000.
For using mnist_train as training set, the cross entropy and accuracies are as follows:
(Train)
cross entropy: 0.021540517909554153
accuracy: 1.0
(Valid)
cross entropy: 0.19180184027644923
accuracy: 0.88
(Test)
cross entropy: 0.21421419249319443
accuracy: 0.92

For using mnist_train_small as training set, the cross entropy and accuracies are as follows:
(Train)
cross entropy: 0.0023562130028028346
accuracy: 1.0
(Valid)
cross entropy: 0.8491928367694916
accuracy: 0.72
(Test)
cross entropy: 0.8202537803945671
accuracy: 0.78

(c) For mnist_train:



For mnist_train_small:



We can see that in the beginning iterations of mnist_train, the gradient descent perform large improvements on both training set and validation set, while later iterations only tune to capture some small details of the training sets (which probably not affecting the accuracy, but decrease just a little bit of the cross entropy).
However, when using a smaller training set, it seems that the training set is pretty much biased, hence doing gradient descent on this small training set doesn't decrease, but increase the cross entropy of evaluating on the validation set.

Overall, I would choose the hyperparameter that kind of balanced in performance and runtime, so choosing a not-so-small learning rate (rather than 10e-3 for example) will require less iterations but not performing bad (such as oscillating or divergent), and hence we could reduce the number of iterations to improve runtime.

4 Locally Weighted Regression

(a) Converting the loss function to vertex form will give:

$$\frac{1}{2}(\mathbf{y}-\mathbf{Xw})^T\mathbf{A}(\mathbf{y}-\mathbf{Xw})+\frac{\lambda}{2}\mathbf{w}^T\mathbf{Iw}$$

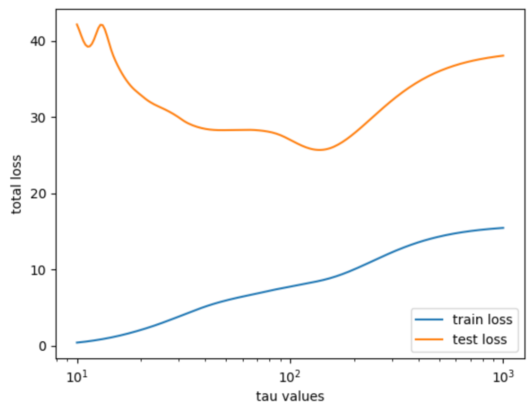
Hence computing gradient with respect to $\mathbf{w}$ will give:

$$\begin{aligned}\frac{\partial\mathcal{J}}{\partial\mathbf{w}}&=\frac{1}{2}(-2\mathbf{X}^T\mathbf{A}(\mathbf{y}-\mathbf{Xw}))+\frac{\lambda}{2}2\mathbf{Iw}\\&=-\mathbf{X}^T\mathbf{A}(\mathbf{y}-\mathbf{Xw})+\lambda\mathbf{Iw}\\&=\mathbf{X}^T\mathbf{A}(\mathbf{Xw}-\mathbf{y})+\lambda\mathbf{Iw}\end{aligned}$$

Setting the gradient to 0 and solve for $\mathbf{w}$ will give:

$$\begin{aligned}\mathbf{X}^T\mathbf{A}(\mathbf{Xw}-\mathbf{y})+\lambda\mathbf{Iw}&=0\\\mathbf{X}^T\mathbf{AXw}-\mathbf{X}^T\mathbf{Ay}+\lambda\mathbf{Iw}&=0\\(\mathbf{X}^T\mathbf{AX}+\lambda\mathbf{I})\mathbf{w}&=\mathbf{X}^T\mathbf{Ay}\\\mathbf{w}&=(\mathbf{X}^T\mathbf{AX}+\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Ay}\end{aligned}$$

(c)



(d) The algorithm perform similar when τ is too big or too small on the test set, and I believe that's because of underfitting and overfitting (just like kNN). However it's only worse when τ is too big for the training set, and I believe that's because we are predicting the training set using the training set, and increasing τ will make the local re-weighting more significant (kind of like overfitting in kNN), making the loss become generally bigger.

(e) The advantage is that the prediction is more accurate comparing to linear regression, because we calculate a new weight vector for every single input data (similar to kNN where every input data is compared to the whole training set), but still using linear regression idea to predict, so the advantage is due to weights being specific to every data input rather than a general weight for the training set only.

The disadvantage is that the runtime of this algorithm is much slower than linear regression (even slower than kNN I would say), because we have to calculate both the distance of a single input data to every data in the training set (kNN part) and minimizing the loss (linear regression part), and it's well-known that kNN already take much longer than linear regression (when training set is considerably large). Hence locally weighted regression perform very slow comparing to linear regression.