COMPSCI 671D: Homework 1 Solutions

Harsh Parikh

January 2019

*Note: Solutions for Problem 2 and Problem 3 have been taken from Shuai Yuan's Homework 1 submission. Thanking him for almost perfect write-up and solutions.

1 Perceptron I

a) It is not necessary that we end up with same hyperplane or with same time for convergence. It depends on the order we perform updates to w's.
Consider, the Problem 2 (part c and d) (Perceptron II), you see two different decision boundaries for original and shuffled data.

b) Proving XOR is not linearly separable:
We need,
$w_1 + 0w_2 \geq t$,
$0w_1 + w_2 \geq t$,
$0w_1 + 0w_2 < t$,
$w_1 + w_2 < t$,

$w_1 \geq t$,
$w_2 \geq t$,
$0 < t$,
$w1 + w2 < t$.
Contradiction. QED

Proof for non-linear separability of parity functions:
Let's assume parity function P of $n > 2$ variables is linearly separable. Group the variables into two parts, first $floor(n/2)$ in first group and rest of variables in other. So, the outcome is 1 if there are total odd number of bits and 0 if even.
There are 4 possible cases for this function $f(G1, G2)$:
1) $G1 = odd$, $G2 = odd = 0$
2) $G1 = even$, $G2 = even = 0$
3) $G1 = odd$, $G2 = even = 1$
4) $G1 = even$, $G2 = odd = 1$

Thus f behaves like an XOR function (or it is essentially an XOR function). If the P is linearly separable in X then so should be f on $G1, G2$ because they are essentially the same function on same inputs. However, we know f is not linearly separable and hence contradiction to our initially assumption on P.

Hence, P is not linearly separable.

c) $w_1 = 1$, w_2=1, w_3 = -2.
If $w \cdot x \geq 1/2$ then it is in class 1, else it is in class 0.

d) Let's denote the number of linearly separable partitions by $C(n, p)$. We will find an expression for $C(n, p)$ through induction. Imagine first having n points and then adding one more point. Now, considering the linearly separable partitions of the previous n points, there are two possibilities: (1) there is a separating hyperplane for the previous n points passing through the new point, in which case each such linearly separable partition of the previous n points gives rise to two distinct linearly separable partitions when the new point is added as the hyperplane can be shifted infinitesimally to place the new point in either class; (2) there is no separating hyperplane passing through the new point in which case each such linearly separable partition gives rise to only one linearly separable partition when the

new point is added, namely the one consistent with the linear separability of all the $n+1$ points. Therefore to find $C(n+1,p)$, we have to count the number of linearly separable partitions in (1) twice and the number of linearly separable partitions in (2) once. This is the same thing as counting the total number of partitions, i.e. $C(n,p)$, and adding the number of linearly separable partitions in (1). But the number of linearly separable partitions in (1) is precisely $C(n,p-1)$, because restricting the separating hyperplane to go through a particular point is the same as eliminating one degree of freedom and thus projecting the p points to a $(p-1)$-dimensional space. Thus, we have the recursive relation:
$C(n+1,p) = C(n,p) + C(n,p-1)$
implies $C(n+1,p) = 2\sum_{i=0}^{p-1} \binom{n}{i}$

2 Perceptron II

a) Since all data were intervals, we parsed all the end points of intervals and used their mean as the numeric data value. As is shown in Fig 1, the data points are linearly separable. We can separate the data points by the line age=50. That means only including two features may be enough to find a decision boundary that separates positive and negative samples. The codes are shown in appendix.

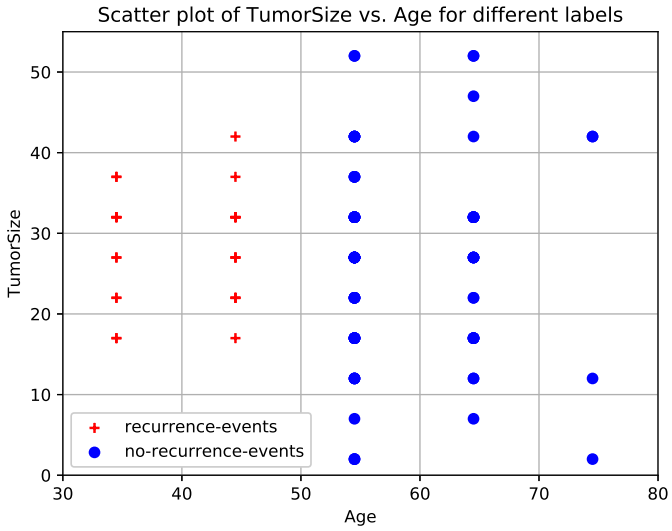

Figure 1: Scatter plot of TumorSize vs. Age (recurrence-events: red cross; no-recurrence-event: blue circle)

b)

```
# Problem 2b
def perceptron(X, y):
    n,p = X.shape
    X = np.hstack((X, np.ones((n, 1)))) # adding intercept dimension
    w = np.zeros(p+1)
    has_mistake = True
    while has_mistake:
        has_mistake = False
        for i in range(n):
            if y[i] * np.dot(X[i], w) <= 0:
                has_mistake = True
                w = w + y[i] * X[i]
    return w
```

c)

Solution. The perceptron algorithm outputs $\boldsymbol{w} = (-0.1692, 2.2843e-14, 5.3783)$, the three dimensions being age, tumor size and intercept, respectively. In order to show the direction, we can normalize $\boldsymbol{w}$ to have unit norm, which gives $(-0.0217, 0.0000, 0.9998)$. The decision boundary are shown in Fig 2. The codes are in appendix.

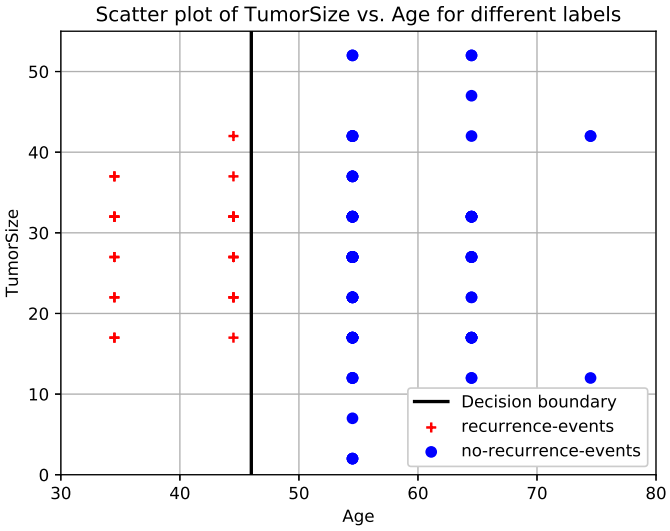

Figure 2: Scatter plot with decision boundary

d)

Solution. After shuffling the dataset, we obtain the following output: $\boldsymbol{w} = (-1.7596, 0.4326, 70.9816)$. If we normalize it to have unit norm, that gives $(-0.0248, 0.0061, 0.9997)$. The decision boundary are shown in Fig 3. The codes are in appendix. Note that the new boundary seems passing a sample point, but it is actually just very closed to the point. The two decision boundaries are different. This is because we changed the order of dataset, and thus the order that we examine samples and update the $\boldsymbol{w}$ is different.

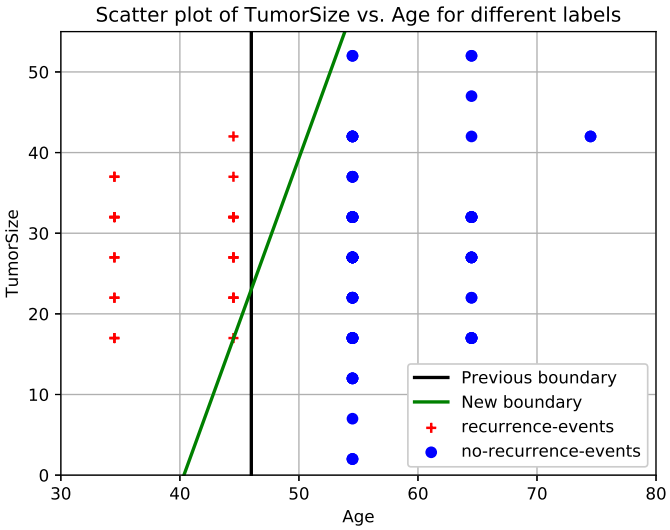

Figure 3: Scatter plot with decision boundary

e)

Solution. The plots are shown in Fig 4 and Fig 5. We can see that the two convergence plots are both consistently going up and down. The first experiment takes 2076 iterations to converge, while the shuffled one takes 33185 iterations. They are not the same. Because the scale of $\boldsymbol{w}$ does not matter (multiplying $\boldsymbol{w}$ by any constant provides us the same decision boundary), there are a lot of feasible solutions to this algorithm, so different executions of the same algorithm do not necessarily converge to the same $\boldsymbol{w}$. In the perceptron convergence theorem, we prove the convergence by showing that the normalized inner product (cosine of angle) between current coefficient $\boldsymbol{w}^{(t)}$ and the optimal $\boldsymbol{w}^*$ goes to one. This means that $\boldsymbol{w}^{(t)}$ is always getting closer to $\boldsymbol{w}^*$, but the training and test accuracy does not have to grow

monotonically since fixing one mistaken point may still incur other mistakes. Therefore, it is totally possible that changing the order of data may make a big difference.

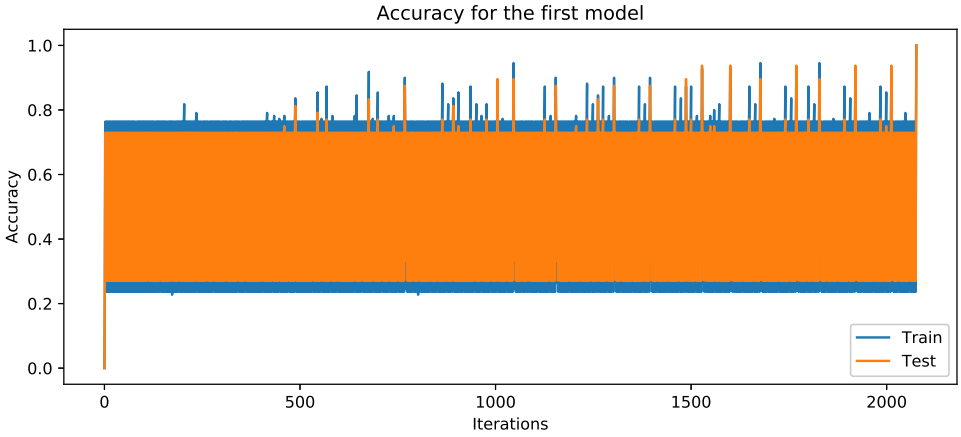

Figure 4: Accuracy for each update

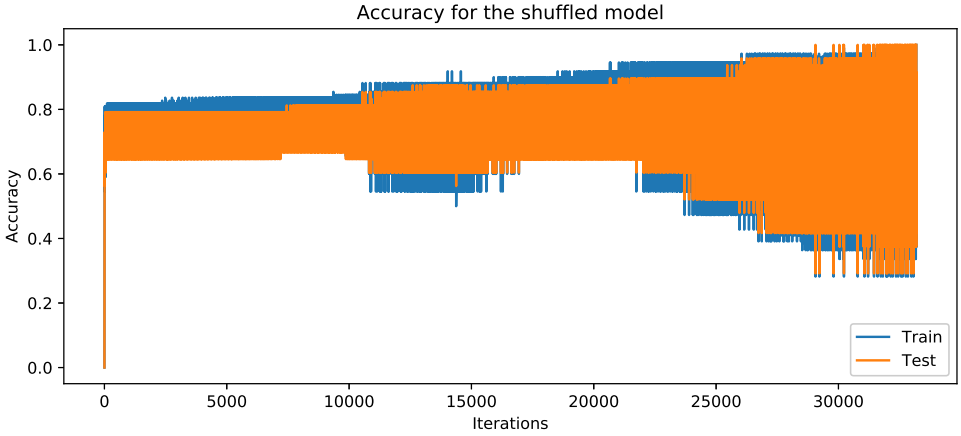

Figure 5: Accuracy for each update (shuffled)

3 ROC, AUC and Cross-Validation

For this problem, all codes can be found in Appendix A.2.

a)

Solution. We first sampled 25% from the original data as our dataset, and we used stratified sampling so that the proportion of labels could be preserved. We then used the stratified cross validation function provided by the sklearn package to do 10-fold split.

For each training case, we used "ovr" multi-class logistic regression with solver "liblinear". In the logistic regression package, the regularization parameter is implemented by C, which is actually the inverse of K. Therefore, we set C to be a large number (1e7) to enact vanilla logistic regression. Inspired by the sklearn online documentation[**?**], we computed the micro average of TPR and FPR, and used that data to draw ROC curves for each of the ten fold cases. The micro average takes the mean values of TP and FP and then compute TPR/FPR, whereas the macro average first computes TPR/FPR for all classes and then takes average. Micro average is more robust to handle class imbalance, and our data actually have some degrees of class imbalance, so we use it here. The ROC curves are shown in Fig 6, and all AUC have been shown in the legends. The code can be found in appendix.

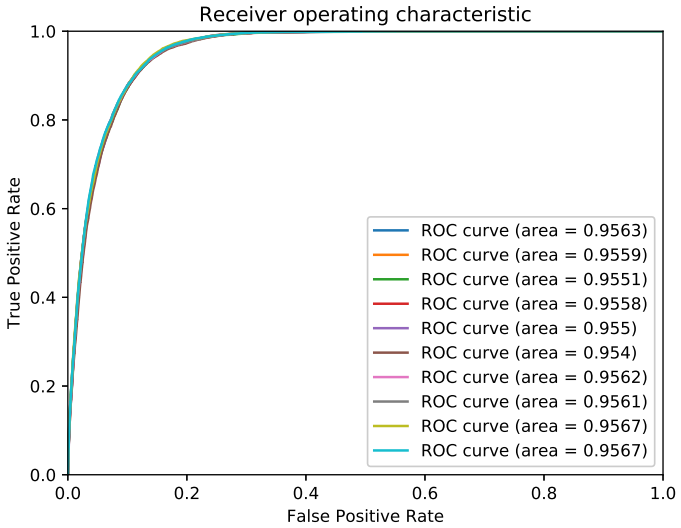

Figure 6: ROC curves for each fold

b)

Solution. In the logistic regression package, they implemented the regularization parameter C, which is the inverse of K. We split the interval $[0.001, 1000]$ log-linearly, using the code "10 ** np.linspace(-3, 3, 12)", so that we can try parameters of different scales.

We rotated for all 9 validation sets and computed their validation accuracies for all 12 possible regularization parameters. We computed the mean and standard deviation of validation accuracy across the 9 experiments. The code can be found in appendix.The results are shown in Tab 1. We can see that the best parameter is $C = 0.5337$

Table 1: Validation accuracy for different regularization parameters

C	0.001	0.0035	0.0123	0.0433	0.1520	0.5337
K	1000	284.8036	81.1131	23.1013	6.5793	1.8738
Accuracy Mean	0.6841	0.6961	0.7036	0.7081	0.7074	**0.7095**
Accuracy Std.	0.0039	0.0037	0.0036	0.0039	0.0045	0.0038
C	1.8738	6.5793	23.1013	81.1131	284.8036	1000
K	0.5337	0.1520	0.0433	0.0123	0.0035	0.001
Accuracy Mean	0.7090	0.7078	0.7093	0.7091	0.7094	0.7082
Accuracy Std.	0.0037	0.0038	0.0041	0.0042	0.0043	0.0045

($K = 1.8738$), which yields an accuracy of $0.7095(\pm 0.0038)$. However, the differences in performance across parameters are not very significant.

c)

Solution. The accuracy values for each regularization parameter are shown in Tab 2. We can see that the algorithm

Table 2: Test accuracy for different regularization parameters

C	0.001	0.0035	0.0123	0.0433	0.1520	0.5337
K	1000	284.8036	81.1131	23.1013	6.5793	1.8738
Accuracy	0.6941	0.7029	0.7164	0.7177	0.7168	**0.7208**
C	1.8738	6.5793	23.1013	81.1131	284.8036	1000
K	0.5337	0.1520	0.0433	0.0123	0.0035	0.001
Accuracy	0.7164	0.7205	0.7186	0.7168	0.7182	0.7199

reaches the largest accuracy at $C = 0.5337$ ($K = 1.8738$). The largest accuracy is 0.7208. Again, the differences among different parameters are not very significant. The parameter K we chose in the last problem still performed the best on the test set. The code can be found in appendix.

d)

Solution. The box plots are shown in Fig 7 and Fig 8. The x-axis means the index of parameter in the list that we are using. The x index corresponds to $C \in \{0.001, 0.0035, 0.0123, 0.0433, 0.1520, 0.5337, 1.8738, 6.5793, 23.1013, 81.1131, 284.8036, 1000\}$ respectively. The codes are shown in appendix.

We can see that tuning parameters does have some effect on the performance. When C is too small (too much regularization), the performance is worse than other cases. When C is large enough ($C \geq 0.5337$), the performance tends to saturate. However, although tuning parameters does make a difference, the performance improvement is still limited: Test accuracy rises from around 0.693 to 0.720, and AUC rises from 0.948 to 0.957. It is hard to say whether these improvements are significant enough, because it all depends on how hard the problem is. To summarize, just for this experiment, tuning parameter is useful to some extent, but limitations still remain.

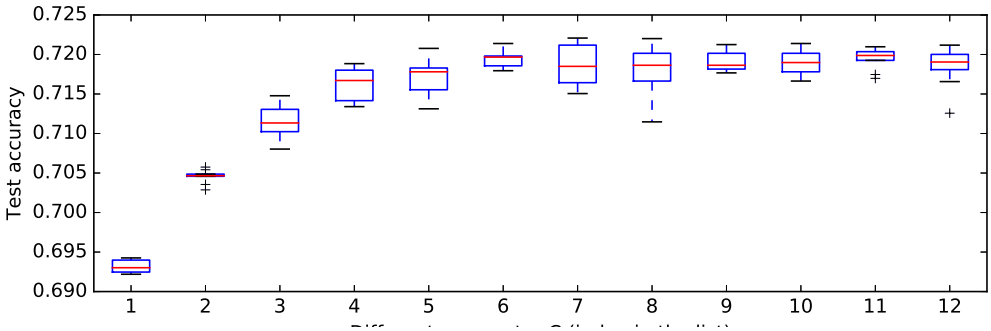

Figure 7: Test accuracy box plots

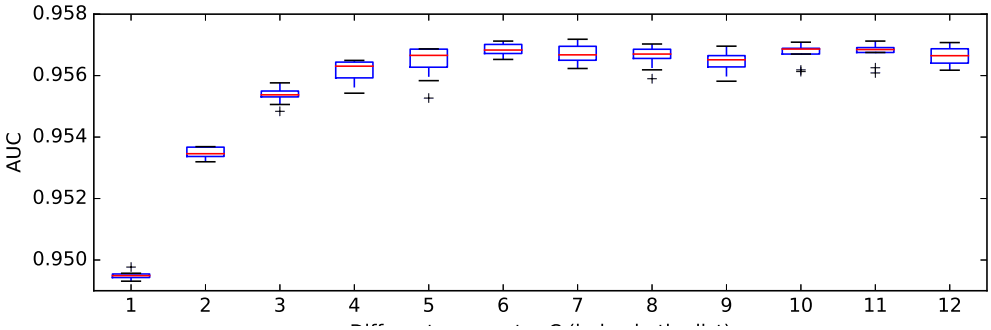

Figure 8: AUC box plots

e)

Solution. We used the previous models in part c) and generated their ROC curves on the test set. The result is shown in Fig 9. All corresponding auc and parameter $C = \frac{1}{K}$ are shown in the legends. We can see that all these curves overlap greatly, which means that the change of parameter does not significantly affect performance.

Although the parameter seems not influential in this experiments, I would still prefer to tune the parameter values. Actually, whether tuning parameter makes some difference strongly depends on the algorithm and the dataset that we are using. Just because tuning parameters do not work on this particular example, does not mean that it never works in general. Tuning parameters can sometimes help us find better models (especially in Deep Learning), and also help us understand the performance better.

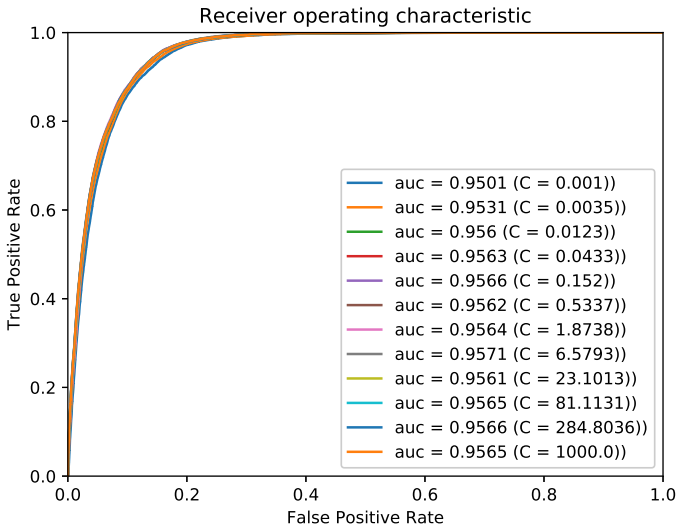

Figure 9: ROC curves for all 12 models in part c