CSC311 A3

Wilson Hsu

March 2020

Question 1

Part (a)

$\boldsymbol{W^{(1)}} \in \mathbb{R}^{d\times d}, \boldsymbol{W^{(2)}} \in \mathbb{R}^{1\times d}, \boldsymbol{z_1} \in \mathbb{R}^d, \boldsymbol{z_2} \in \mathbb{R}^d$

Part (b)

Parameters $= \boldsymbol{W^{(1)}}, \boldsymbol{W^{(2)}}$, so total number will be $d^2 + d$, because $\boldsymbol{W^{(1)}} = d \times d, \boldsymbol{W^{(2)}} = d$

Part (c)

$$
\begin{aligned}
\bar{y} &= \frac{\partial \mathcal{L}}{\partial y} = (y - t) \\
\bar{W}^{(2)} &= \bar{y}\frac{\partial \dagger}{\partial W^{(2)}} = \bar{y}z_2^\top \\
\bar{z_2} &= \bar{y}\frac{\partial \dagger}{\partial z_2} = \bar{y}W^{(2)} \\
\bar{h} &= \bar{z_2}\frac{\partial z^2}{\partial h} = \bar{z_2} \\
\bar{z_1} &= \bar{h}\frac{\partial h}{\partial z_1} = \bar{h}\sigma'(z_1) \\
\bar{W}^{(1)} &= \bar{z_1}\frac{\partial z_1}{\partial W^{(1)}} = \bar{z_1}x^\top \\
\bar{x} &= \bar{z_1}\frac{\partial z_1}{\partial x} + \bar{z_2}\frac{\partial z_2}{\partial x} = \bar{z_1}W^{(1)} + \bar{z_2}
\end{aligned}
$$

Question 2

Part(a)

Compute $\frac{\partial y_k}{\partial z_{k'}}$:
Case $k' = k$:

$$
\begin{aligned}
\frac{\partial y_k}{\partial z_{k'}} &= \frac{e^{z_{k'}}(\sum_{i=1}^{K} e^{z_i}) - e^{2z_{k'}}}{(\sum_{i=1}^{K} e^{z_i})^2} \\
&= \frac{e^{z_{k'}}}{\sum_{i=1}^{K} e^{z_i}} \frac{(\sum_{i=1}^{K} e^{z_i}) - e^{z_{k'}}}{\sum_{i=1}^{K} e^{z_i}} \\
&= y_{k'}(1 - y_{k'})
\end{aligned}
$$

Case $k' \neq k$:

$$
\begin{aligned}
\frac{\partial y_k}{\partial z_{k'}} &= -\frac{e^{z_{k'}} e^{z_k}}{(\sum_{i=1}^{K} e^{z_i})^2} \\
&= -\frac{e^{z_{k'}}}{\sum_{i=1}^{K} e^{z_i}} \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} \\
&= -y_{k'} y_k
\end{aligned}
$$

Therefore:

$$
\frac{\partial y_k}{\partial z_{k'}} = \begin{cases} y_{k'}(1 - y_{k'}), k = k' \\ -y_{k'} y_k, k \neq k' \end{cases}
$$

Part(b)

Compute $\frac{\partial L_{CE}(\boldsymbol{t}, \boldsymbol{y}(\boldsymbol{x}; W))}{\partial \boldsymbol{W_k}}$:

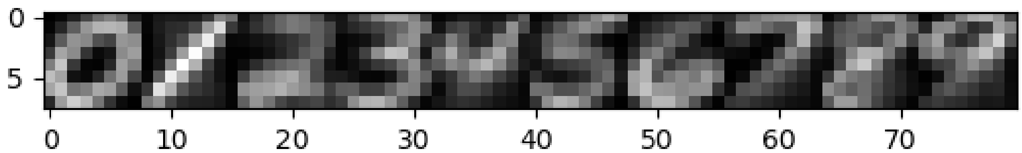
$$
\begin{aligned}
\frac{\partial L_{CE}(\boldsymbol{t}, \boldsymbol{y}(\boldsymbol{x}; W))}{\partial \boldsymbol{w}_k} &= \frac{\partial L_{CE}(\boldsymbol{t}, \boldsymbol{y}(\boldsymbol{x}; W))}{\partial \boldsymbol{y}_i} \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{z}_k} \frac{\partial \boldsymbol{z}_k}{\partial \boldsymbol{w}_k} \\
&= \frac{\partial - \sum_{i=1}^{K} \boldsymbol{t}_i log(\boldsymbol{y}_i)}{\partial \boldsymbol{y}_i} \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{z}_k} \frac{\partial \boldsymbol{z}_k}{\partial \boldsymbol{w}_k} \\
&= (-\sum_{i=1}^{K} \frac{\boldsymbol{t}_i}{\boldsymbol{y}_i}) \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{z}_k} \frac{\partial \boldsymbol{z}_k}{\partial \boldsymbol{w}_k} \\
&= (-\sum_{i=1}^{K} \frac{\boldsymbol{t}_i}{\boldsymbol{y}_i}) \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{z}_k} \boldsymbol{x} && \text{(Since } \boldsymbol{z}_k = \boldsymbol{w}_k \boldsymbol{x},\ \tfrac{\partial \boldsymbol{z}_k}{\partial \boldsymbol{w}_k} = \boldsymbol{x}) \\
&= \boldsymbol{x}(-\sum_{i=1}^{K} \frac{\boldsymbol{t}_i}{\boldsymbol{y}_i}) \frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{z}_k}
\end{aligned}
$$

We know that $\boldsymbol{y}$ and $\boldsymbol{z}$ should have same dimension, since softmax function does not change dimensions. Therefore there will be two cases, whether $i = k$ or $i \neq k$ and we have computed both cases in the previous part.

$$
\begin{aligned}
\frac{\partial \boldsymbol{y}_i}{\partial \boldsymbol{z}_k} &= \begin{cases} \boldsymbol{y}_k(1-\boldsymbol{y}_k), i = k \\ -\boldsymbol{y}_k\boldsymbol{y}_i, i \neq k \end{cases} \\
\frac{\partial L_{CE}(\boldsymbol{t}, \boldsymbol{y}(\boldsymbol{x}; W))}{\partial \boldsymbol{w}_k} &= \boldsymbol{x}(-\frac{\boldsymbol{t}_k}{\boldsymbol{y}_k}\boldsymbol{y}_k(1-\boldsymbol{y}_k) + \sum_{i \neq k} \frac{\boldsymbol{t}_i}{\boldsymbol{y}_i} y_k y_i) \\
&= \boldsymbol{x}(-\boldsymbol{t}_k(1-\boldsymbol{y}_k) + \sum_{i \neq k} \boldsymbol{t}_i \boldsymbol{y}_k) \\
&= \boldsymbol{x}(-\boldsymbol{t}_k + \boldsymbol{t}_k\boldsymbol{y}_k + \sum_{i \neq k} \boldsymbol{t}_i \boldsymbol{y}_k) \\
&= \boldsymbol{x}(-\boldsymbol{t}_k + \sum_{k=1}^{K} \boldsymbol{t}_k \boldsymbol{y}_k) \\
&= \boldsymbol{x}(-\boldsymbol{t}_k + \boldsymbol{y}_k) && \text{(Since } \boldsymbol{t}_k \text{ is just a one-hot)} \\
&= \boldsymbol{x}(\boldsymbol{y}_k - \boldsymbol{t}_k)
\end{aligned}
$$

Question 3

0.



3.1 K-NN Classifier

1.

```
knn with k = 1 on Train Set has accurary: 1.0
knn with k = 1 on Test Set has accurary: 0.96875
knn with k = 15 on Train Set has accurary: 0.9594285714285714
knn with k = 15 on Test Set has accurary: 0.9585
```

2.

I decided to break tie by choosing the first digit class that has the highest number of occurrence within k nearest neighbors to our test point. I decided to do it this way, so that I do not have to implement a random selector function which could possibly slow down the run-time and because that Numpy's argmax function uses this method.
My existing implementation for tie-breaking:

```python
digits, counts = np.unique(nearest_neighbors_labels, return_counts=True)
# Return the max number (Tie breaking: First occurence)
digit = digits[np.argmax(counts)]
```

3.

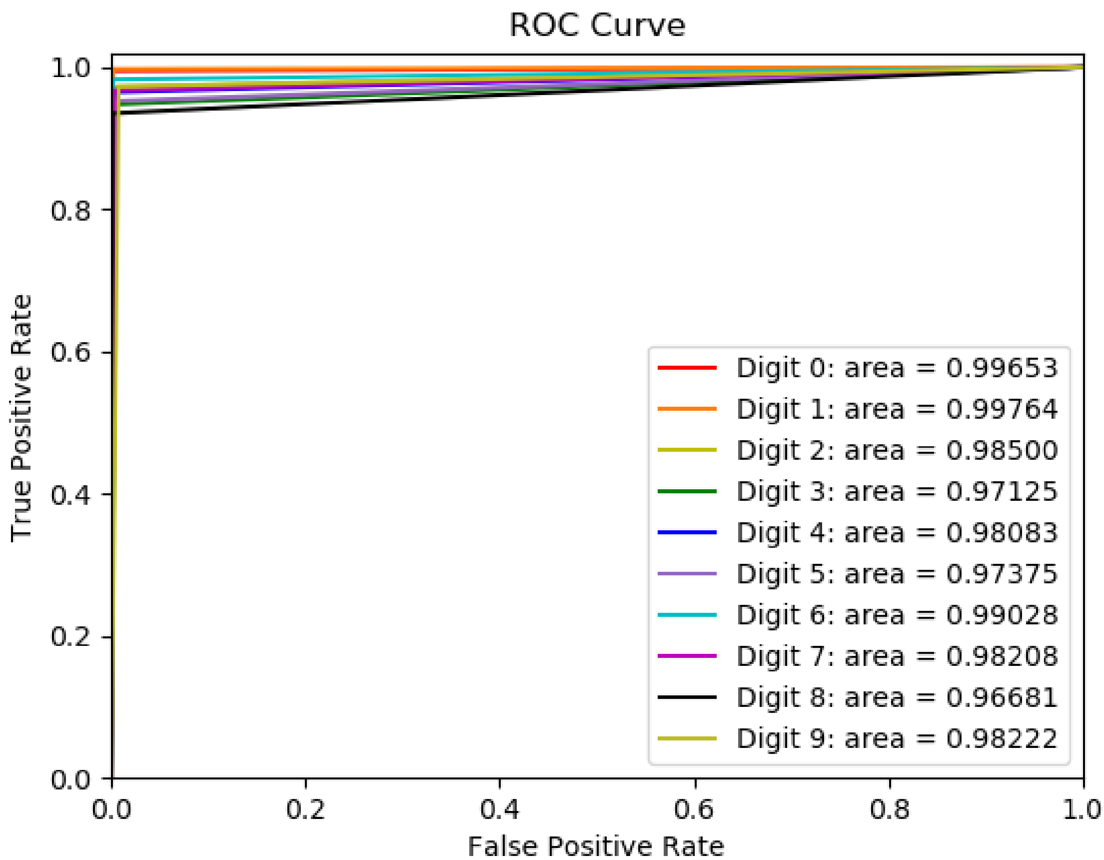
```
Optimal k: 1
Optimal k average accuracy across folds: 0.9647142857142856
Optimal knn on Train Set has accurary: 1.0
Optimal knn on Test Set has accurary: 0.96875
```

3.2 Classifiers Comparison

Code submitted in python files.
Side note, for MLP NN classifier, we take over fitting into account by adding a dropout layer.

3.3 Model Comparison

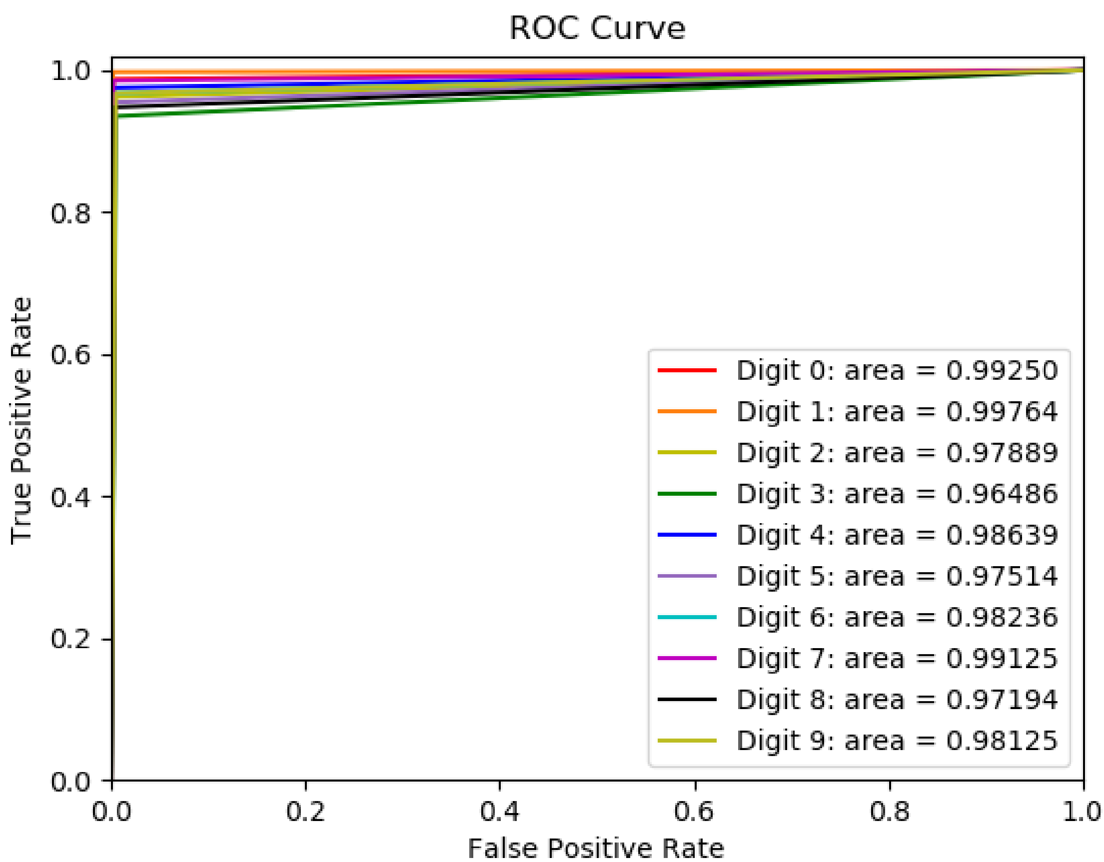
K-NN Classifier



Other metrics:

```
Mean Squared Error: 0.407
Confusion Matrix:
[[398   0   0   0   0   0   1   1   0   0]
 [  0 399   1   0   0   0   0   0   0   0]
 [  4   0 389   3   1   0   0   1   1   1]
 [  0   1   4 379   0  11   1   2   1   1]
 [  0   0   0   0 386   0   2   2   0  10]
 [  1   0   0  12   0 381   3   1   2   0]
 [  0   4   2   0   0   0 393   0   1   0]
 [  0   1   1   0   3   0   0 387   0   8]
 [  2   2   1   2   1   7   0   2 374   9]
 [  0   0   0   1   7   0   0   3   0 389]]
Accuracy: 0.96875
Precision: [0.98271605 0.98034398 0.97738693 0.95465995 0.96984925 0.95488722
 0.9825     0.96992481 0.98680739 0.93062201]
Recall: [0.995  0.9975 0.9725 0.9475 0.965  0.9525 0.9825 0.9675 0.935  0.9725]
```

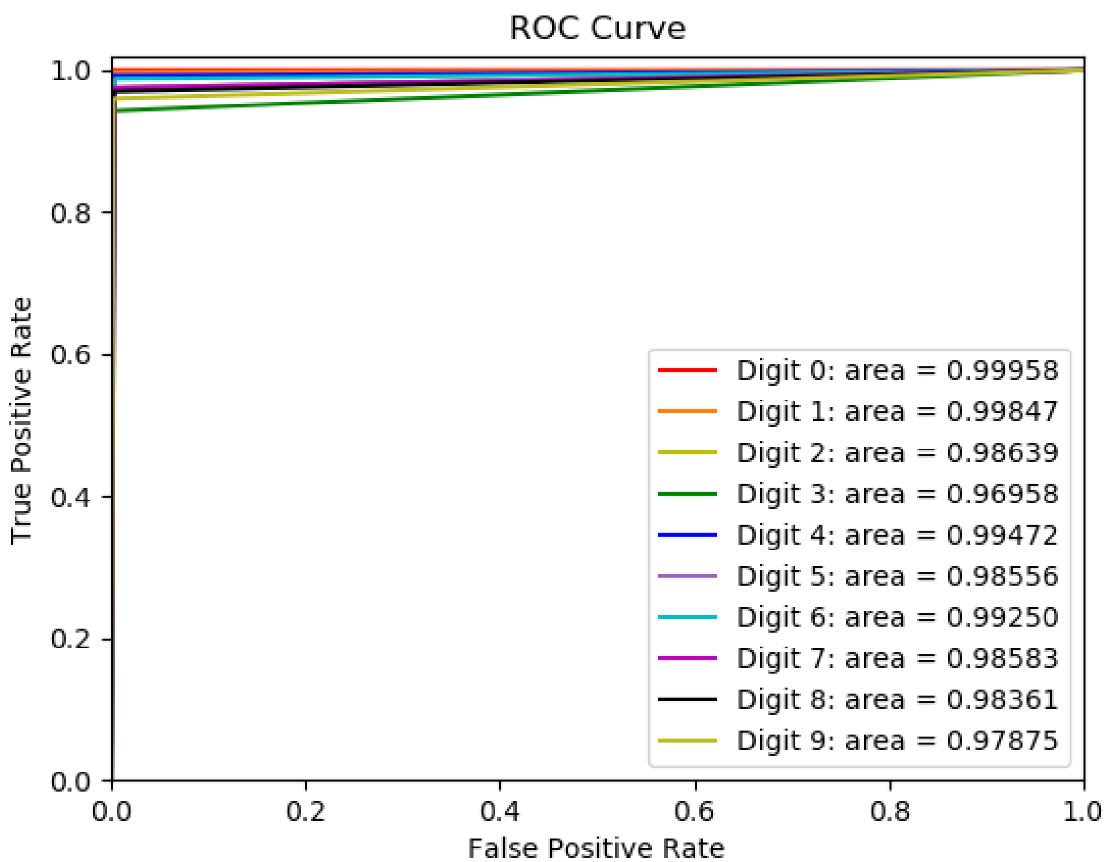
MLP - Neural Network Classifier



Other metrics:

```
Mean Squared Error: 0.50425
Confusion Matrix:
[[395   1   1   0   1   0   1   0   1   0]
 [  0 399   0   0   0   0   0   0   1   0]
 [  1   1 385   4   0   1   6   0   2   0]
 [  1   0   7 374   0   9   0   1   6   2]
 [  0   0   0   0 390   0   1   0   0   9]
 [  3   2   1   8   0 382   1   1   2   0]
 [  3   3   2   0   3   2 387   0   0   0]
 [  0   0   0   0   0   0   0 394   0   6]
 [  1   1   4   6   0   4   1   3 379   1]
 [  0   0   2   1   4   1   0   4   1 387]]
Accuracy: 0.968
Precision: [0.97772277 0.98034398 0.95771144 0.95165394 0.9798995  0.95739348
 0.97481108 0.97766749 0.96683673 0.95555556]
Recall: [0.9875 0.9975 0.9625 0.935  0.975  0.955  0.9675 0.985  0.9475 0.9675]
```

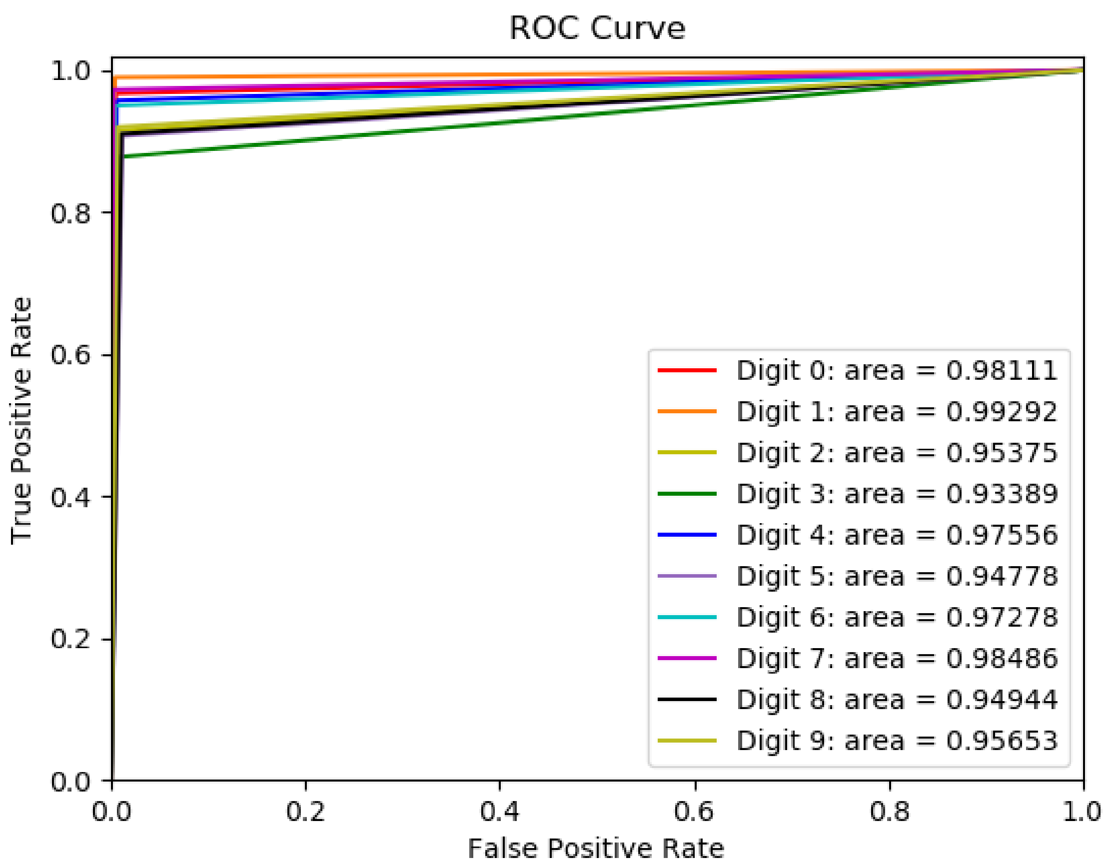
SVM Classifier



Other metrics:

```
Mean Squared Error: 0.28725
Confusion Matrix:
[[400   0   0   0   0   0   0   0   0   0]
 [  0 399   0   0   0   0   1   0   0   0]
 [  0   0 390   2   0   2   4   1   1   0]
 [  0   1   6 377   0   7   0   2   6   1]
 [  0   0   0   0 397   0   2   0   0   1]
 [  1   0   0   4   0 390   2   1   2   0]
 [  1   0   1   0   3   0 395   0   0   0]
 [  0   0   1   0   4   0   0 390   0   5]
 [  1   0   0   4   0   5   0   0 388   2]
 [  0   1   0   2   4   0   0   8   1 384]]
Accuracy: 0.9775
Precision: [0.99255583 0.99501247 0.9798995  0.96915167 0.97303922 0.96534653
 0.97772277 0.97014925 0.97487437 0.97709924]
Recall: [1.     0.9975 0.975  0.9425 0.9925 0.975  0.9875 0.975  0.97   0.96  ]
```

Adaboost Classifier



Other metrics:

```
Mean Squared Error: 0.90875
Confusion Matrix:
[[387   1   2   2   2   2   2   0   2   0]
 [  0 396   1   0   1   0   1   0   1   0]
 [  4   1 366   7   1   6   5   1   9   0]
 [  2   1  12 351   0  18   0   1  12   3]
 [  1   3   0   0 383   0   5   0   1   7]
 [  2   3   2  21   1 363   3   1   3   1]
 [  2   4   4   0   5   4 380   0   1   0]
 [  1   1   0   0   2   1   0 389   0   6]
 [  7   0   6   3   2  10   0   0 364   8]
 [  0   1   0   2   9   2   0   7  11 368]]
Accuracy: 0.93675
Precision: [0.95320197 0.96350365 0.93129771 0.90932642 0.94334975 0.89408867
 0.95959596 0.97493734 0.9009901  0.93638677]
Recall: [0.9675 0.99   0.915  0.8775 0.9575 0.9075 0.95   0.9725 0.91   0.92  ]
```

Based on accuracy, we see that SVM has the highest and Adaboost has the lowest. Looking at confusion matrix, (Above diagonal and below diagonal) SVM has significantly less false negative and false positive predictions, while Adaboost has significantly more.

It did strike me at first to find out that SVM has the best performance, even beating MLP (but it might be because of how I built my Neural Network model). But it matched my expectation that SVM (with rbf kernel) performed better than Adaboost on this dataset. Since each hand written digit's image should have a clear margin (Seen from Part (0), average image of each class are still fairly recognizable), it would bring more advantage to SVM classifier at maximizing the separation hyper-plane between each class.

While Adaboost was expected to not perform well, since it is a multi-classification problem, we had to make 10 different Adaboost classifier and take the maximum probability out of all. If one of those classifier does not perform well, making a lot of wrong decisions, it would affect the overall performance, just like what I observe with my Adaboost classifier, it perform especially bad at digit 3 making many false negative predictions.