CSE 446: Machine Learning
Winter 2018

Assignment 2

from
Lukas Nies
University of Washington

02/01/18

Contents

0 Policies

0.1 List of Collaborators

My collaborator was Edith Heiter (discussed Problem 2 and 4). The development of the answers though was completely independent and individually.

0.2 List of Acknowledgments

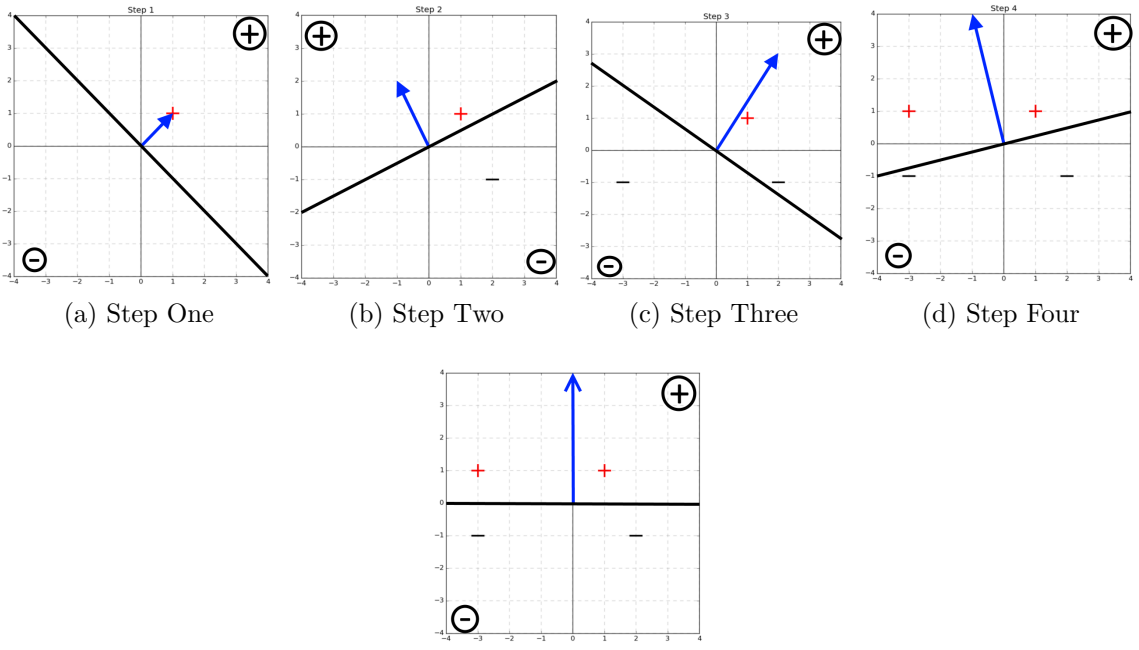
None.

0.3 Policies

I have read and understood these policies.

1 Perceptron Exercise

The evolution of the weight vector w are shown in figure 1 after the data was observed so the weights were already updated. In the same figure the maximum margin is shown since the perceptron didn't maximize the margin after the training set.
To solve Problem 1.3 I will follow the proof according to [1].



(a) Step One (b) Step Two (c) Step Three (d) Step Four

(e) Maximum margin

Figure 1: Visualization of the perceptron's decision boundary. The plots show the weight vector w in blue and the corresponding decision boundary (black) after they were updated. Plot (e) shows the maximum decision margin which is has the value 1 in this case.

Theorem 1.1. *If data D is linear separable with the geometric margin $\gamma^* = 0.5$ and has the maximum norm $\|x\| \leq 5 \forall x \in D$ then the algorithm will convert after at most $\frac{25}{\gamma^2}$ updates.*

Proof. Suppose x^* realizes $\gamma^* > 0$ and let $w^{(k)}$ be the k^{th} weight vector after k updates. We show in the following that the angle between w and w^* decreases such that the algorithm converges. That is when $w^* \cdot w$ increases faster than $\|w\|^2$.
After the k^{th} update we find

$$w^* \cdot w^{(k)} = w^* \left(w^{(k-1)} + yx\right) = w * \cdot w^{(k-1)} + yw^* \cdot x \geq w * \cdot w^{(k-1)} + k\gamma$$

and

$$\|w^{(k)}\|^2 = \|w^{(k-1)} + yx\|^2 = \|w^{(k-1)}\|^2 + 2yw^{(k-1)} \cdot x + y^2 \cdot \|x\|^2 \geq \|w^{(k-1)}\|^2 + 5^2 + 0$$

The last line shows that $\|w^{(k)}\|$ increases by at least 25 every update. Therefore $\|w^{(k)}\|^2 \leq 25k$. So

$$\sqrt{25k} \geq \|w^{(k)}\| \geq w^* \cdot w^{(k)} \geq 5\gamma \Leftrightarrow k \leq \frac{25}{\gamma^2}$$

■

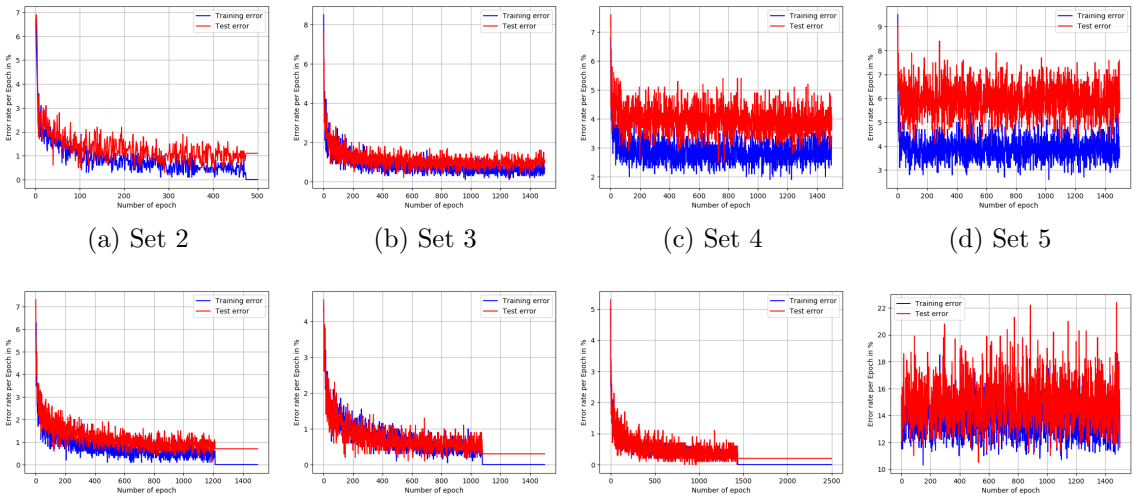
The maximum number of mistakes (updates) the perceptron has to make until it converges is given by $\frac{25}{\gamma^2} = \frac{25}{0.25} = 100$.

2 Implementation: Perceptron

See tables 1 and 2 for answers to questions 2 to 5. Some interesting plots concerning question 1 to 5 are shown in figure 3. Some remarks to the implementation: the implemented algorithm takes the number of iterations and the filenames from the command line and then loops for a certain amount of iterations over the perceptron and trains the weight vector. If the algorithm converges, a lower bound is calculated as the smallest normalized weight vector encountered in the training process. At 10% and 90% of the epochs the weight vector is printed to check whether some features might be noise or not. In order to create a development set, the training set was split into 4:1, generating a tuning set of 200 examples to tune the maximal number of iterations. This hyper parameter was achieved by training the perceptron and then testing with the development set for the lowest development error. The corresponding number of epochs was chosen to test calculate the training, development and test error. Plots will be saved for a general overview of the results.

To tune the performance on set 6 I merged training set 6, 7, and 8 to create a larger training set and deleted some features I expected to be noise (see figure 4).

When looking at data 6 to 8 we can see that the data is from the same source and not overlapping. In order to increase the precision on the test set (which is the same for 6 to 8) we merged the three training data sets combining now a total of 7000 observations to train the perceptron and do nt consider the features which we expect to me noise (in this case, feature 10, 12, and 18). The result can be seen in table 3 and figure 4 and 3.



(a) Set 2 (b) Set 3 (c) Set 4 (d) Set 5

(e) Set 6 (f) Set 7 (g) Set 8 (h) Set 9

Figure 2: Visualization of the error rates of the different sets. In blue the training error rate and in red the test error rate.

dataset	question number 2	3	4	5
2	Is linear separable because error rates don't change after $\approx$ 380 epochs, the smallest margin found is $\approx$ 0.005, see figure 2(a)	Does overfit slightly since test set error rises slowly for longer training times, see 2(a) and doesn't match the trainings error	Weight of feature 9 is one to two orders of magnitude smaller than the others, might be noise	Tuned maximum number of iterations, set it to 94, got 8% training error, 19% test error, 4% development error
3	Is not linear separable because it does not converge and error rate rises a little after $\approx$ 1500 epochs	Does overfit a little, see figure 2(b) since test error rises with growing numbers of epochs	Weight of feature 6 and 19 are one to two orders of magnitude smaller than the others, might be noise	Tuned maximum number of iterations, set it to 176, got 10% training error, 10% test error, 3% development error
4	Is not linear separable because it does not converge and error rate rises a little after $\approx$ 1500 epochs, doesn't match the training error, see figure 2(c)	Does overfit strongly training and test error don't fit at all	No obvious noise found	Tuned maximum number of iterations, set it to 815, got 20% training error, 46% test error, 7% development error (overall bad performance)
5	Is not linear separable because it does not converge and test error is steadily above training error rate	Does overfit strongly, errors don't match at all, see plot 2(d)	No obvious noise found	Tuned maximum number of iterations, set it to 632, got 17% training error, 31% test error, 8% development error (overall bad performance)

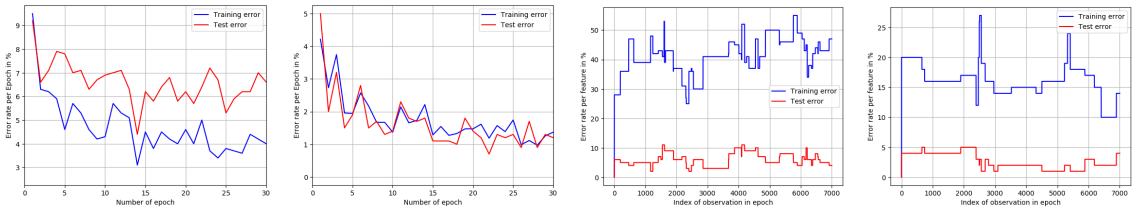
Table 1: Tabel with answers to questions of problem 2.

dataset	question number 2	3	4	5
6	The data is linear separable since the algorithm converges after roughly ≈ 1210 epochs, the smallest margin is ≈ 0.004	Does overfit slightly for larger epochs (see figure 2(e)) since test error is larger than training error	Feature 12 generates a weight which is 3 orders of magnitude smaller then the others, this might be noise	Tuned maximum number of iterations, set it to 68, got 13% training error, 28% test error, 4% development error
7	The data is linear separable since the algorithm converges after roughly ≈ 1300 epochs, the smallest margin is ≈ 0.005	Does not overfit for larger epochs (see plot 2(f)) since error rates are not separable	Feature 10, 12 and 18 generate weights which are 3 to 4 orders of magnitude smaller then the others, this might be noise	Tuned maximum number of iterations, set it to 224, got 6% training error, 6% test error, 3% development error
8	The data is linear separable since the algorithm converges after roughly ≈ 1300 epochs, the smallest margin is ≈ 0.005	Does not overfit, test error is nearly identical with training error, see plot 2(g)	Feature 10, 12 and 18 still generate weights which are 3 to 4 orders of magnitude smaller then the others, this might be noise	Tuned maximum number of iterations, set it to 806, got 8% training error, 2% test error, 4% development error
9	Is not linear separable because it does not converge and error is nearly steady	Strongly overfit since test error is steadily roughly two percent worse than training error rate (see plot 2(h))	Weight of feature 1 and 20 are one to two orders of magnitude smaller than the others, might be noise	Tuned maximum number of iterations, set it to 409, got 80% training error, 154% test error, 27% development error (very bad performance, might not be a problem solvable by the perceptron.)

Table 2: Continuation of the answers for the first table.

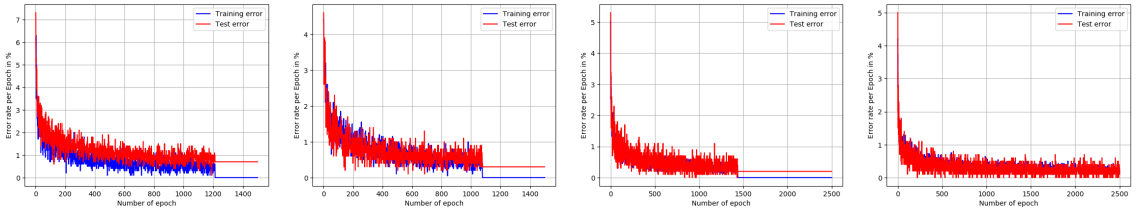
dataset	question number 2	3	4	5
Surpr.	Should be separable since it's from the same type of data as data 6, 7, and 8, so lower limit for the margin should be similar	Does not overfit, test error seems to perform equally good, maybe even better than training error	Since the "noisy" features were not considered while training and testing, no small weights could be found	Tuned maximum number of iterations, set it to 631, got 13% training error, 2% test error, 7% development error

Table 3: Surprising table with answers to the surprise problem.



(a) Set 5 zoomed (b) Set 10 zoomed (c) Set 10 early epoch (d) Set 10 late epoch

Figure 3: In (a) and (b) zoomed plots for early epochs for set 5 and 10 are shown (set 10 is set 7+8+9 - noise). We can clearly see that set 5 starts with large discrepancy between the two error rates where set 10 has good performance on both training and test rate. Plot (c) and (d) show the error rate changing after encountering new observations for an early epoch (c) and a late epoch (d) in training set 10. We can see that in late epochs less mistakes are made as in earlier epochs.



(a) 1000 datapoints (b) 2000 datapoints (c) 4000 datapoints (d) 7000 datapoints and noise cancellation

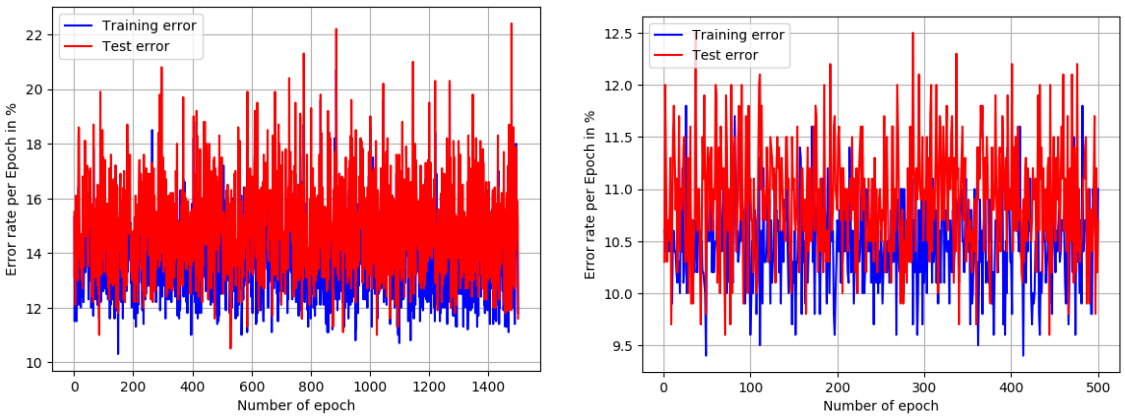
Figure 4: Visualization of the error rates of the sets from the same source with different sizes of training data . In blue the training error rate and in red the test error rate.

3 Beat the Perceptron

To beat the perceptron I implemented the voted perceptron approach in order to yield better results by paying with longer processing time. I focused on data set number 9 since the performance was quite bad in the normal approach and I wanted to test how strong the voted perceptron is even if the data is not linear separable. In figure 5 the results for the voted perceptron and the normal perceptron are compared.

After tuning the hyperparameter of maximum interation steps for the voted perceptron I found an trainingset error of 80%, test error of 120% and a dev error of 40%. By comparing both sub plots in figure 5 one can see that the overall performance is increased by nearly 5% compared to the normal perceptron implementation.

Even though I didn't expect life-changing by trying to solve a not perceptron solvable problem with another perceptron implementation (not linear separable!) I achieve a quite good improvement in the performance.



(a) Perceptron on set 9

(b) Voted perceptron on set 9

Figure 5: Visualization of the error rates of set 9 when treated by perceptron (left) and voted perceptron (right).

4 PCA on Digits

4.1 Convariance matrix construction

Let $X \in \mathbb{R}^{n \times d}$ be the data matrix, $\vec{x}_i \in \mathbb{R}^{d \times 1}$ be the i^{th} element of X, $\vec{\mu} \in \mathbb{R}^{d \times 1}$ be the mean vector of all $\vec{x}_i$, $\vec{1}$ be a vector in $\mathbb{R}^{n \times 1}$ consisting of 1's, and $X_c = X - \vec{1}\vec{\mu}^T$ be the centered data matrix. Note that all vectors are considered as column vectors.

In figure 8 ten digits from the MNIST database are plotted, in figure 9 the mean image ($\vec{\mu}$) is plotted.

There are two methods to write down the covariance matrix:

- Vector based method:

$$\Sigma = \frac{1}{n} \sum_{i=1}^{n} (\vec{x}_i - \vec{\mu}) \cdot (\vec{x}_i - \vec{\mu})^T \tag{1}$$

- Matrix based method:

$$\Sigma = \frac{1}{n} (X_c^T \cdot X_c) \tag{2}$$

By considering the dimensions of the centered data matrix $X_c \in \mathbb{R}^{n \times d}$ we find for sigma $\Sigma \in \mathbb{R}^{d \times d}$.

By coding both methods (code attached to .tar.gz, as usual) we found for the vector method a runtime of roughly 191s and < 1s for the matrix based method. This is in increase of over 19100% in runtime. By testing the average absolute difference between the two methods we found a very small value (9.78×10^{-16}), in the range of machine precision so the two results are equal.

The advantage of using Numpy's dot product instead of a for loop over all the components in order to calculate the covariance matrix lies within it's implementation as a "vector" language, realized in `C`. Those languages try to apply the same operation to a whole chunk of data, not only between two single objects and translate it to a low level and processor near, fast operation. This is an example of implicit parallelization [2].

4.2 Eigen-Analysis

By using the SVD on the covariance matrix we find following eigenvalues:

- $\lambda_1 = 5.10819064381$
- $\lambda_2 = 3.70090586283$
- $\lambda_{10} = 1.24026412182$
- $\lambda_{30} = 0.362081056419$
- $\lambda_{50} = 0.168298737356$
- $\sum_{i=1}^{d} \lambda_i = 52.4218857752$

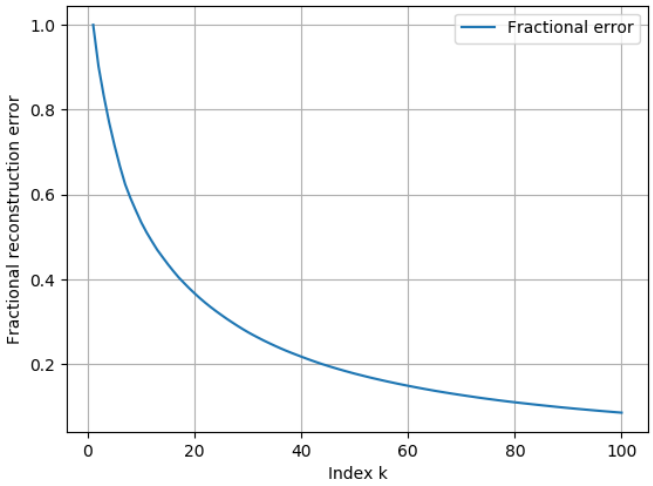


Figure 6: Fractional Reconstruction error for the first $k = 100$ eigenvalues

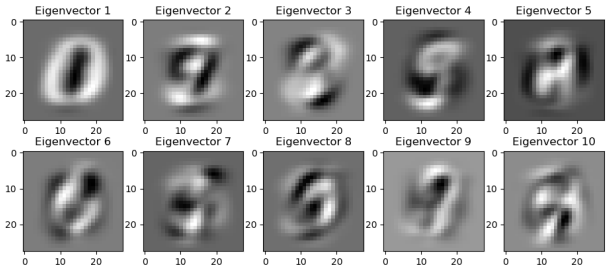


Figure 7: Plot of the first ten eigenvectors

The fractional reconstruction error for the first $k = 100$ eigenvalues is plotted in figure 6.

The first 11 eigenvalues make up 50% of the total variance and the first 44 eigenvalues make up 80% of the eigenvalues.

The first ten eigenvectors are shown in figure 7. They should represent the dimensions with the highest variance (biggest information) and therefore should carry essential information about the ten possible digits, zero to nine.

4.3 Pseudocode for Reconstruction

Goal: best reconstruction (squared error should be small) for dimensionality reduction from d to k dimensions.

1. Use the top k eigenvectors since they carry the larges variance (information) for the reconstruction. The actual number k is a hyperparameter which has to be optimized.

2. By using SVD on the covariance matrix Σ we get the k eigenvalues. The top k eigenvalues make up the matrix $\hat{U} \in \mathbb{R}^{d \times k}$. The reduced data matrix is then

given by

$$\hat{X} = \left(X - \vec{1} \cdot \vec{\mu}^T\right) \cdot \hat{U} \tag{3}$$

3. The d-dimensional reconstruction is then given by

$$X_{\text{rec}} = \hat{X} \cdot \hat{U}^T + \vec{1} \cdot \vec{\mu} \tag{4}$$

4.4 Visualization and Reconstruction

For $k = 30$ it takes roughly 2s to reconstruct the data. The plots are shown for different k in figures 10 and 11. For low values of k ($k = 1, 3, 5$) we can see that the reconstruction is composed of one, three and five eigenvalues. The reconstruction itself is not useful for determining the original digit. For medium k ($k = 10, 25$) the pictures get more and more blurry until for higher k ($k = 50, k = 200, 500+$) one finally can estimate the digit for sure. It seems that the top 10 dimensions seem to capture enough information to guess the digit more or less correctly, more dimensions are better.

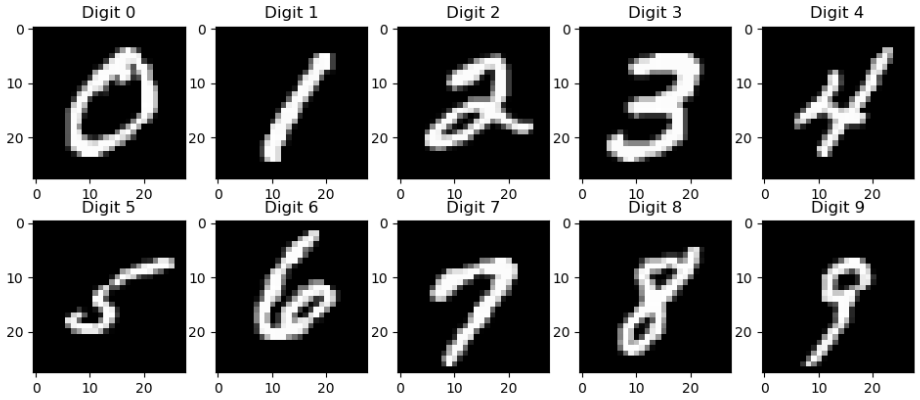


Figure 8: Grey scale plot of ten digits of the MNIST database

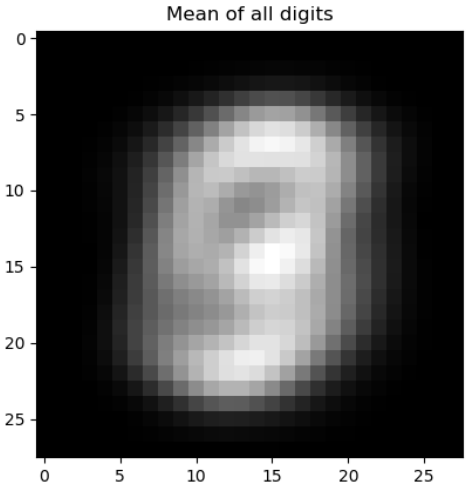
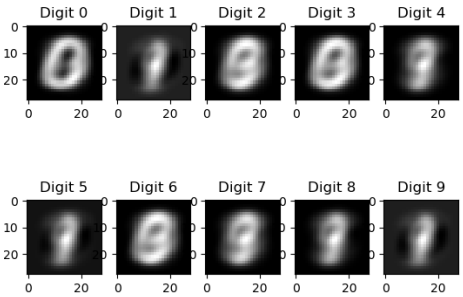
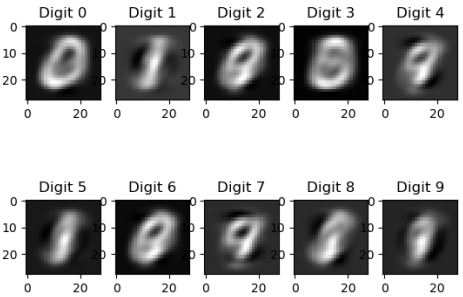


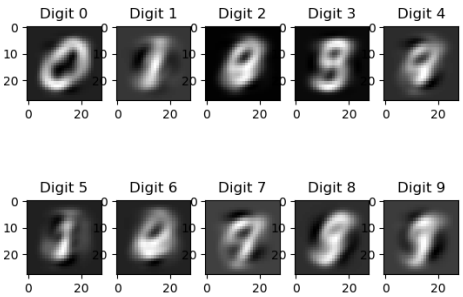
Figure 9: Grey scale plot of the mean image of the MNIST database



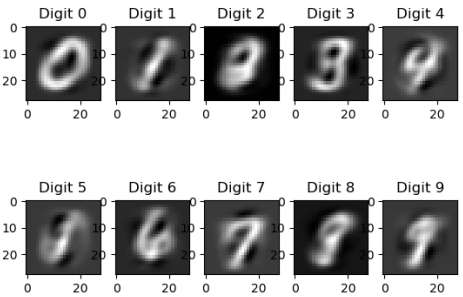
(a) Reconstruction for $k = 1$



(b) Reconstruction for $k = 3$

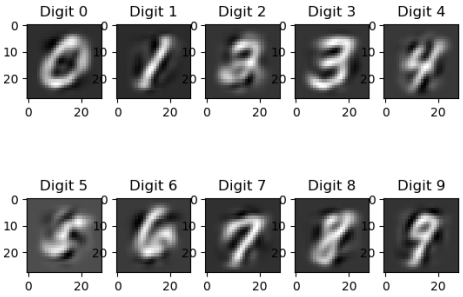


(c) Reconstruction for $k = 5$

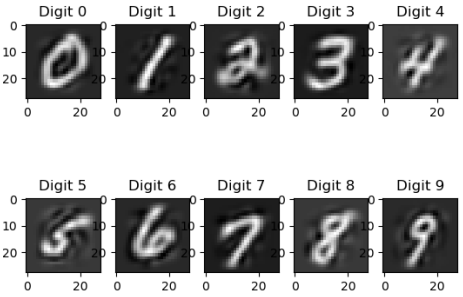


(d) Reconstruction for $k = 10$

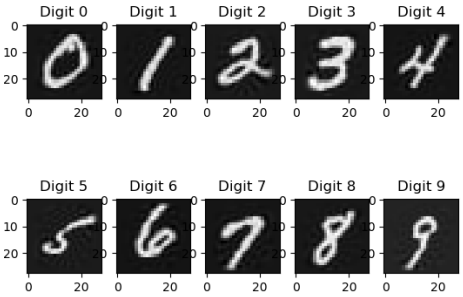
Figure 10: Reconstruction of the ten digits used in Problem part 4.1 for different k.



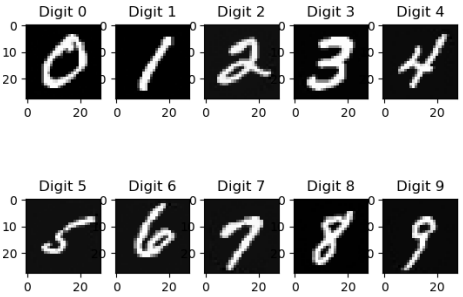
(a) Reconstruction for $k = 25$



(b) Reconstruction for $k = 50$



(c) Reconstruction for $k = 200$



(d) Reconstruction for $k = 500$

Figure 11: Reconstruction of the ten digits used in Problem part 4.1 for different k. Continuation from previous page.

5 Bayes Optimal Classifier

Theorem 5.1. *If D is a distribution with samples (x,y) where x and y are independent and the theorem for the Bayesian optimal classifier holds, then the we can write*

$$f^{BO}(x) = \arg\max_y D(x, y) = \arg\max_y D(y|x). \tag{5}$$

Proof. When we assume that x and y are independent and the Bayesian theorem holds then we can write

$$D(x, y) = D(x)D(y) = D(y)D(x|y) = D(y)\frac{D(x)}{D(y)}D(y|x) = D(x)D(y|x). \tag{6}$$

Because we want to optimize the classifier in terms of y we can drop the $D(x)$ since it is a constant and independent of y. Therefore:

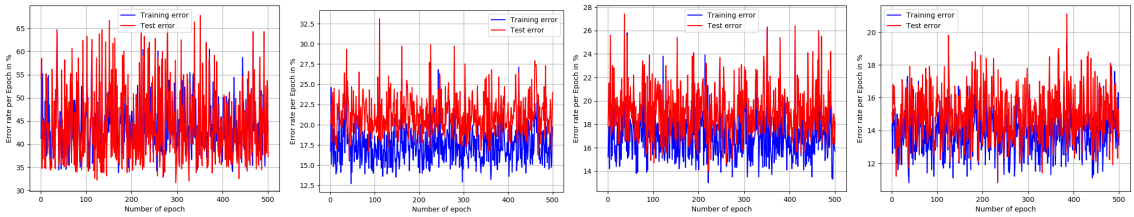
$$f^{BO}(x) = \arg\max_y D(x, y) = \arg\max_y D(x)D(y|x) = \arg\max_y D(y|x) \tag{7}$$

■

This slightly different definition tries to maximize the probability to find y conditioned on x. If we had a classifier which makes fewer errors than the Bayesian classifier then the probability to find an y conditioned on x would still be higher by definition for the Bayesian classifier. Therefore the overall error of BO would be smaller and the other classifier would perform worse.

6 Bonus: Dimensionality Reduction

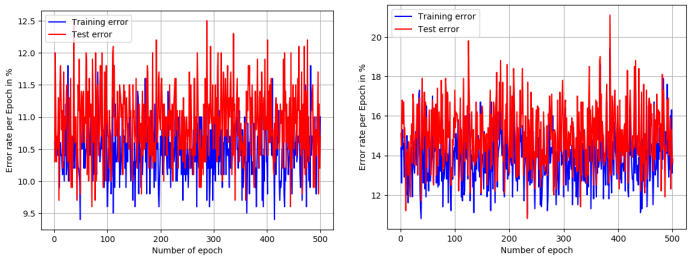
The dimensionality algorithm from previous problem was utilized to project dataset 9 down to different dimensions k to test the performance. Since I expected at least two features to be noise in set 9, I though I could filter noise in form of dimensionality reduction. Plot 12 compares the performance of different k for the normal perceptron implementation. One can clearly see that the performance against my expectations deteriorates. For $k = 20$ we completely reconstruct the data, therefore no changes.



(a) Performance for $k = 5$ (b) Performance for $k = 15$ (c) Performance for $k = 17$ (d) Performance for $k = 20$

Figure 12: Performance for different k for the normal perceptron implementation on set 9.

Since the voted perceptron has quite a long runtime I just performed only one test with $k = 18$ since I expected two features to be noise, but as plot 13 shows, no improvement was yielded.



(a) Performance for $k = 5$ (b) Performance for $k = 15$

Figure 13: Performance for different $k = 18$ for the voted perceptron implementation on set 9 (left) and the normal implementation withou dimensionality reduction (right).

References

[1] Hal Daume III. *A Course in Machine Learning*. Secondary printing, January 2017.

[2] Wikipedia.org. Array programming. Retrieved online: `https://en.wikipedia.org/wiki/Array_programming`, February, 2018.