Homework #2

CSE 546: Machine Learning

1 A Taste of Learning Theory

1. Consider the definition of $R(\delta)$.

$$\begin{aligned}
\mathbb{E}_{XY,\delta}[\mathbf{1}\{\delta(X) \neq Y\}] &= \mathbb{E}_{XY}[\mathbb{E}_{\delta|XY}[\mathbf{1}\{\delta(x) \neq y\}|X=x, Y=y]] \\
&= \mathbb{E}_{XY}[\mathbf{1}\{\delta(x) \neq y\}|X=x, Y=y] \\
&= \mathbb{E}_{XY}[P(\delta(x) \neq y)|X=x, Y=y] \\
&= \mathbb{E}_{XY}[1-\alpha_\delta(x,y)|X=x, Y=y] \\
&= \mathbb{E}_{X}[\mathbb{E}_{Y|X}[1-\alpha_\delta(x,y)|X=x]] \\
&= \mathbb{E}_{X}[\sum_{i=1}^{K}(1-\alpha_\delta(x,y))P(Y=y|X=x)] \quad \text{By the expectation definition} \\
&= \mathbb{E}_{X}[\sum_{i=1}^{K}P(Y=y|X=x) - \sum_{i=1}^{K}\alpha_\delta(x,y)P(Y=y|X=x)] \quad \sum_{i=1}^{K}P(Y=y|X=x)=1 \\
&= \mathbb{E}_{X}[1-\sum_{i=1}^{K}\alpha_\delta(x,y)P(Y=y|X=x)]
\end{aligned}$$

To find the permissible Bayes classifiers, we have to minimize the last expression $\mathbb{E}_X[1-\sum_{i=1}^{K}\alpha_\delta(x,y)P(Y=y|X=x)]$, which is the same as maximize $\sum_{i=1}^{K}\alpha_\delta(x,y)P(Y=y|X=x)$. Thus, we can write the maximization problem as follows.

$$\begin{gathered}
\max_\alpha \sum_{i=1}^{K}\alpha_\delta(x,y)P(Y=y|X=x) \\
\sum_{y=1}^{K}\alpha_\delta(x,y)=1 \\
0 \leq \alpha_\delta(x,y) \leq 1
\end{gathered}$$

Note that $P(Y=y|X=x)$ is always positive. This implies that our objective function will maximized when we set $\alpha_\delta(x,y)=1$ for the largest value of $P(Y=y|X=x)$. Thus, the set of all Bayes classifiers is as follows.

$$\delta_* = \{\delta|\sum_{y\in Y'}\alpha(x,y)=1 \quad \text{and} \quad \alpha(x,y)=0 \quad \forall y \notin Y'\}$$

where $Y' = \{y = \text{argmax}_y P(Y=y|X=x)\}$.
For a deterministic decision rule, the maximization problem will written as follows.

$$\begin{gathered}
\max_\alpha \sum_{i=1}^{K}\alpha_\delta(x,y)P(Y=y|X=x) \\
\sum_{y=1}^{K}\alpha_\delta(x,y)=1 \\
\alpha_\delta(x,y) \in \{0,1\}
\end{gathered}$$

Thus, the objective function will be maximized when we pick the largest value of $P(Y = y|X = x)$. Thus, the decision rule is to sort $P(Y = y|X = x)$ for all $y \in \{1, \dots, K\}$ and set $\alpha_\delta(x, y*) = 1$ where $y* = \text{argmax}_y P(Y = y|X = x)$ and set $\alpha_\delta(x, y \neq y*) = 0$. If there are more than one $y* \in \{1, \dots, K\}$ that leads to $\text{argmax}_y P(Y = y|X = x)$, we are indifferent between these $y*$'s. Note that we can pick only one due to the restriction on $\alpha_\delta(x, y)$.

2.

a. First, note that $\mathbf{1}(\widetilde{f}(x_i) \neq y_i)$ is random variable that follows Bernoulli distribution by its definition. Suppose $\mathbf{1}(\widetilde{f}(x_i) \neq y_i)$ follows Bernoulli distribution with parameter p. Then, we have $p = \mathbb{E}_{XY}[\mathbf{1}(\widetilde{f}(X) \neq Y)] = R(\widetilde{f})$.

Let $Z_i = \mathbf{1}(\widetilde{f}(x_i) \neq y_i)$. Then, we have following expression.

$$|\widehat{R}_n(\widetilde{f}) - R(\widetilde{f})| = |\frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(\widetilde{f}(x_i) \neq y_i) - p| = |\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}_{XY}[Z_i]|$$

Based on the last expression above, we can use the Hoeffding's inequality.

$$\mathbb{P}\left(|\widehat{R}_n(\widetilde{f}) - R(\widetilde{f})| \geq \epsilon\right) \leq 2\exp\left(-\frac{2n\epsilon^2}{(1-0)^2}\right) = 2\exp\left(-2n\epsilon^2\right)$$

Let $2\exp\left(-2n\epsilon^2\right) = \delta$. Then, we can calculate ϵ.

$$\begin{aligned} 2\exp\left(-2n\epsilon^2\right) &= \delta \\ \exp\left(-2n\epsilon^2\right) &= \frac{\delta}{2} \\ \epsilon^2 &= \frac{1}{2n}\log(\frac{2}{\delta}) \\ \epsilon &= \sqrt{\frac{1}{2n}\log(\frac{2}{\delta})} \end{aligned}$$

Thus, we can rewrite $\mathbb{P}\left(|\widehat{R}_n(\widetilde{f}) - R(\widetilde{f})| \geq \epsilon\right)$ as follows.

$$\begin{aligned} \mathbb{P}\left(|\widehat{R}_n(\widetilde{f}) - R(\widetilde{f})| \geq \sqrt{\frac{1}{2n}\log(\frac{2}{\delta})}\right) &\leq \delta \\ 1 - \mathbb{P}\left(|\widehat{R}_n(\widetilde{f}) - R(\widetilde{f})| \geq \sqrt{\frac{1}{2n}\log(\frac{2}{\delta})}\right) &\leq 1 - \delta \\ \mathbb{P}\left(|\widehat{R}_n(\widetilde{f}) - R(\widetilde{f})| \leq \sqrt{\frac{1}{2n}\log(\frac{2}{\delta})}\right) &\leq 1 - \delta \end{aligned}$$

Thus, $A = \sqrt{\frac{1}{2n}\log(\frac{2}{\delta})}$.

b. If we replace $\widetilde{f}$ with f^* in part a., the same confidence interval still holds. As $f^* = \arg\min_{f \in \mathcal{F}} R(f)$ and $\mathcal{F} = \{f_1, \dots, f_k\}$, this means that f^* does not depend on $\{(x_i, y_i)\}_{i=1}^n$. Then, the i.i.d. condition still holds. Replacing f^*, we have $\widehat{R}_n(f^*) = \frac{1}{n}\sum_{i=1}^n Z_i$ and $R(f^*) = \mathbb{E}[Z_i]$. We can still use Hoeffdings inequality to construct the confidence interval. Thus, we will get the same confidence interval as in part a.

c. If we replace $\widetilde{f}$ with $\widehat{f}$ in part a., the same confidence interval no longer holds as the i.i.d. condition is violated due to the dependency of $\widetilde{f}$ on $\{(x_i, y_i)\}_{i=1}^n$. When the i.i.d. condition is violated, we cannot use Hoeffdings inequality to construct the confidence interval.

d. From question 1, we have $R(\delta) = \mathbb{E}_X[\sum_{i=1}^{K}(1-\alpha_\delta(x,y))P(Y=y|X=x)]$.
Now, consider deterministic classifier $f : \mathcal{X} \to \{-1,1\}$ that classifies only two classes where $f \in \mathcal{F}$ and $\mathcal{F}$ is infinite set, we can write $R(f)$ as follows.

$$\begin{aligned}
R(f) &= \mathbb{E}_X[\sum_y (1 - P(f(X)=Y))P(Y=y|X=x)] \\
&= \mathbb{E}_X[\sum_y (P(f(X)\neq Y))P(Y=y|X=x)] \\
&= \mathbb{E}_X[\sum_y \mathbf{1}(f(X)\neq Y)P(Y=y|X=x)] \\
&= \mathbb{E}_X[\mathbf{1}(f(X)=-1)P(Y=1|X=x) + \mathbf{1}(f(X)=1)P(Y=-1|X=x)]
\end{aligned}$$

If $P(Y=1|X=x) = P(Y=-1|X=x) = 0.5$ for any underlying distribution for random variable X, we will have $R(f) = 0.5$.
Since $\mathcal{F}$ is infinite such as $\mathcal{F}$ be all deterministic classifiers, there will always exist $f \in \mathcal{F}$ that could classify all data $\{x_i, y_i\}_{i=1}^n$ correctly. Thus, the empirical risk $\widehat{R}_n(\widehat{f}) = 0$.

e. For a confidence interval to be simultaneously hold for the losses of all $f \in \mathcal{F}$, we have to consider $\mathbb{P}(\sup_{f\in\mathcal{F}} |\widehat{R}_n(f) - R(f)| \geq \epsilon)$.

$$\begin{aligned}
\mathbb{P}(\sup_{f\in\mathcal{F}} |\widehat{R}_n(f) - R(f)| \geq \epsilon) &= \mathbb{P}(\bigcup_{f\in\mathcal{F}} |\widehat{R}_n(f) - R(f)| \geq \epsilon) \\
&\leq \sum_{f\in\mathcal{F}} \mathbb{P}(|\widehat{R}_n(f) - R(f)| \geq \epsilon) \\
&= \sum_{f\in\mathcal{F}} 2\exp\left(-2n\epsilon^2\right) \\
&= 2|\mathcal{F}|\exp\left(-2n\epsilon^2\right)
\end{aligned}$$

Then, we have $\mathbb{P}(\sup_{f\in\mathcal{F}} |\widehat{R}_n(f) - R(f)| \geq \epsilon) \leq 2|\mathcal{F}|\exp\left(-2n\epsilon^2\right)$, which be expressed as follows.

$$\begin{aligned}
1 - \mathbb{P}(\sup_{f\in\mathcal{F}} |\widehat{R}_n(f) - R(f)| \geq \epsilon) &\geq 1 - 2|\mathcal{F}|\exp\left(-2n\epsilon^2\right) \\
\mathbb{P}(\sup_{f\in\mathcal{F}} |\widehat{R}_n(f) - R(f)| \leq \epsilon) &\geq 1 - 2|\mathcal{F}|\exp\left(-2n\epsilon^2\right)
\end{aligned}$$

Let $2|\mathcal{F}|\exp\left(-2n\epsilon^2\right) = \delta$. Then, we can solve for ϵ.

$$\begin{aligned}
\exp\left(-2n\epsilon^2\right) &\leq \frac{\delta}{2|\mathcal{F}|} \\
\frac{1}{\exp\left(2n\epsilon^2\right)} &\leq \frac{\delta}{2|\mathcal{F}|} \\
\frac{2|\mathcal{F}|}{\delta} &\leq \exp\left(2n\epsilon^2\right) \\
\log\left(\frac{2|\mathcal{F}|}{\delta}\right) &\leq 2n\epsilon^2 \\
\frac{1}{2n}\log\left(\frac{2|\mathcal{F}|}{\delta}\right) &\leq \epsilon^2 \\
\sqrt{\frac{1}{2n}\log\left(\frac{2|\mathcal{F}|}{\delta}\right)} &\leq \epsilon
\end{aligned}$$

Thus, $B = \sqrt{\frac{1}{2n}\log\left(\frac{2|\mathcal{F}|}{\delta}\right)}$.

f. Note that $|R(\widehat{f}) - \widehat{R}_n(\widehat{f})| = |\widehat{R}_n(\widehat{f}) - R(\widehat{f})|$. Since $\widehat{f} \in \mathcal{F}$ or $\widehat{f}$ is the best classifier in $\mathcal{F}$, then we have the following expression.

$$|\widehat{R}_n(\widehat{f}) - R(\widehat{f})| \leq |\sup_{f\in\mathcal{F}} \widehat{R}_n(f) - R(\widehat{f})|$$

where $|\sup_{f\in\mathcal{F}} \widehat{R}_n(f) - R(\widehat{f})|$ holds with a probability greater than $1-\delta$ according to part e.

g.

$$\begin{aligned}
R(\widehat{f}) - R(f^*) &= R(\widehat{f}) - \widehat{R}_n(\widehat{f}) + \widehat{R}_n(\widehat{f}) - R(f^*) \\
&= (R(\widehat{f}) - \widehat{R}_n(\widehat{f})) + (\widehat{R}_n(\widehat{f}) - R(f^*)) \\
&\leq |\sup_{f\in\mathcal{F}} \widehat{R}_n(f) - R(\widehat{f})| + (\widehat{R}_n(\widehat{f}) - R(f^*)) \quad \text{by part f.} \\
&= |\sup_{f\in\mathcal{F}} \widehat{R}_n(f) - R(\widehat{f})| + (\widehat{R}_n(\widehat{f}) - \inf_{f\in\mathcal{F}} R(f)) \\
&= |\sup_{f\in\mathcal{F}} \widehat{R}_n(f) - R(\widehat{f})| + |\sup_{f\in\mathcal{F}} \widehat{R}_n(\widehat{f}) - R(f)| \\
&\leq |\sup_{f\in\mathcal{F}} \widehat{R}_n(f) - R(\widehat{f})| + |\sup_{f\in\mathcal{F}} \widehat{R}_n(f) - R(f)| \\
&= 2|\sup_{f\in\mathcal{F}} \widehat{R}_n(f) - R(\widehat{f})| \\
&\leq 2 \times \sqrt{\frac{2}{n} \log\left(\frac{2|\mathcal{F}|}{\delta}\right)} \quad \text{by part e.}
\end{aligned}$$

Thus, $C = 2 \times \sqrt{\frac{1}{2n} \log\left(\frac{2|\mathcal{F}|}{\delta}\right)}$.

h. By definition, we have $\widehat{R}_n(f) = \frac{1}{n}\sum_{i=1}^n \mathbf{1}(f(x_i) \neq y_i)$. Consider $\widehat{R}_n(f) = 0$.

$$\widehat{R}_n(f) = \frac{1}{n}\sum_{i=1}^n 0 = \frac{n \times 0}{n} = 0$$

Therefore, $P(\widehat{R}_n(f) = 0)$ can occur only when all classification are correct.

$$\begin{aligned}
P(\widehat{R}_n(f) = 0) &= \prod_{i=1}^n (1 - P(f(x_i) \neq y_i)) \\
&= (1 - P(f(X) \neq Y))^n \\
&= (1 - P(f(X) \neq Y))^n \\
&= (1 - \mathbb{E}_{XY}[\mathbf{1}(f(X) \neq Y)])^n \\
&= (1 - R(f))^n \\
&\leq (1-\epsilon)^n \qquad \text{by assumption } R(f) > \epsilon
\end{aligned}$$

Next, show that $(1-\epsilon)^n \leq e^{-n\epsilon}$. Using the expression $1 + x \leq e^x$, we have the following expression.

$$\begin{gathered}
1 - \epsilon \leq e^{-\epsilon} \\
(1-\epsilon)^n \leq e^{-n\epsilon}
\end{gathered}$$

This completes the proof on $\mathbb{P}(\widehat{R}_n(f) = 0) \leq (1-\epsilon)^n \leq e^{-n\epsilon}$.

Next, consider the reverse case of $\widehat{R}_n(\widehat{f}) = 0 \implies R(\widehat{f}) - R(f^*) \leq \frac{\log(|\mathcal{F}|/\delta)}{n}$. We will have $R(\widehat{f}) - R(f^*) \geq \frac{\log(|\mathcal{F}|/\delta)}{n}$. Based on this new assumption,

i. From part h, we know that $R(\widehat{f}) - R(f^*) \leq \frac{\log(|\mathcal{F}|/\delta)}{n}$. We can check if learning is possible for each $|\mathcal{F}|$ by calculating the limitation of $\frac{\log(|\mathcal{F}|/\delta)}{n}$ as n becomes large.

(a) $|\mathcal{F}|$ is a constant or $|\mathcal{F}| = c$.

$$\lim_{n\to 0} \frac{\log(c/\delta)}{n} = \lim_{n\to 0} \frac{\log(c)}{n} - \lim_{n\to 0} \frac{\log(\delta)}{n} = 0 - 0 = 0$$

Since the limitation on the upper bound is zero, it is possible to learn and the excess risk is a function of $O(n^{-1})$.

(b) $|\mathcal{F}| = n^p$ for some constant p.

$$\lim_{n\to 0} \frac{\log(n^p/\delta)}{n} = \lim_{n\to 0} \frac{p\log(n)}{n} - \lim_{n\to 0} \frac{\log(\delta)}{n} = \lim_{n\to 0} \frac{p\log(n)}{n} = p \lim_{n\to 0} \frac{1/n}{1} = 0$$

Since the limitation on the upper bound is zero, it is possible to learn and the excess risk is a function of $O(n^{-1}\log(n))$.

(c) $|\mathcal{F}| = \exp(\sqrt{n})$.

$$\lim_{n\to 0} \frac{\log(\exp(\sqrt{n})/\delta)}{n} = \lim_{n\to 0} \frac{\sqrt{n}\log(e)}{n} - \lim_{n\to 0} \frac{\log(\delta)}{n} = \lim_{n\to 0} \frac{\sqrt{n}\log(e)}{n} = \lim_{n\to 0} \frac{n^{1/2}}{n} = \lim_{n\to 0} n^{-1/2} = 0$$

Since the limitation on the upper bound is zero, it is possible to learn and the excess risk is a function of $O(n^{-1/2})$.

(d) $|\mathcal{F}| = \exp(10n)$.

$$\lim_{n\to 0} \frac{\log(\exp(10n)/\delta)}{n} = \lim_{n\to 0} \frac{10n\log(e)}{n} - \lim_{n\to 0} \frac{\log(\delta)}{n} = 10\log(e) \lim_{n\to 0} \frac{n}{n} = 10$$

Since the limitation on the upper bound greater than zero, it is not possible to learn.

2 Programming: Lasso

3.
Plot 1:

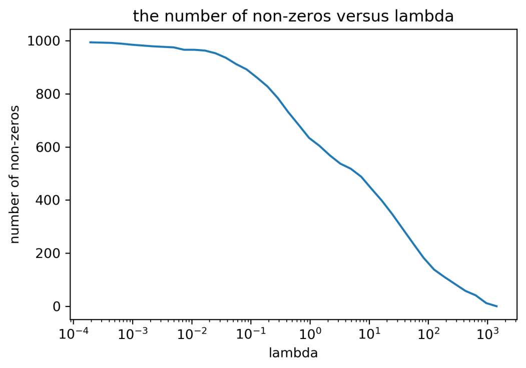

Figure 1: The number of non-zeros versus λ

Plot 2:

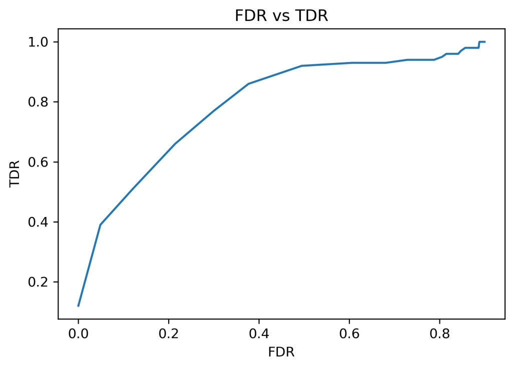

Figure 2: FDR versus TDR

It is clearly shown that number of nonzeros varies inversely with the size of lambda. At maximum lambda, none of the features is selected. Thus, value of both TDR and FDR are zeros. As lambda decreases, number of nonzeros or features selected increases. Both TDR and FDR increases, while size of lambda decreases.

4.

a. Plot 1: Squared Error of Training and Validation Data

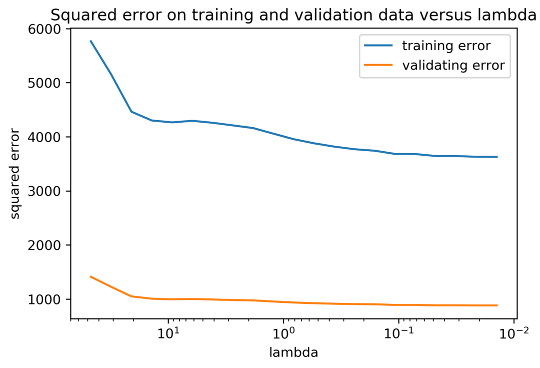

Figure 3: Squared Error of Training and Validation Data

Plot 2: Mean squared error of Training and Validation Data (which I tried to get from the Squared Error)

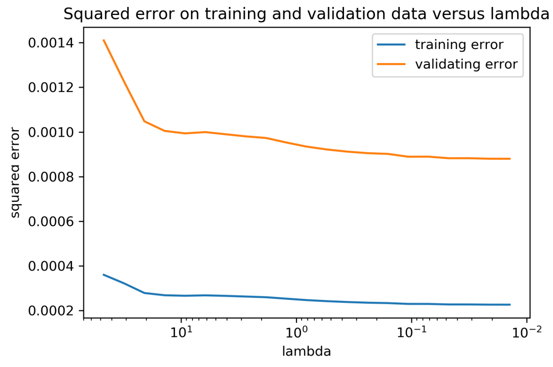

Figure 4: Mean Squared Error of Training and Validation Data

Plot 3: Number of nonzeros versus lambda

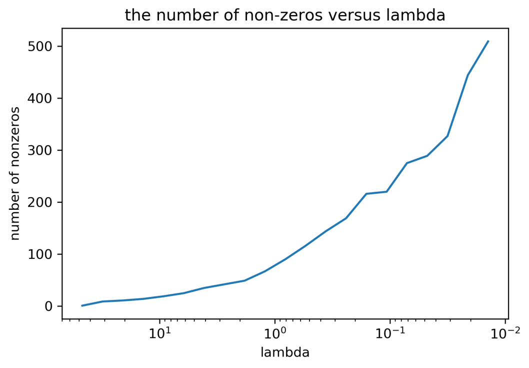

Figure 5: Mean Squared Error of Training and Validation Data

b. Mean squared error for training data: 0.00022675350839222399
Mean squared error for validation data: 0.0008799982085554286

Mean squared error for testing data: 0.959019323607585

c. Ten largest weight:

   (a) -27.01946528
   (b) -15.1325619
   (c) -10.2800546
   (d) -8.52877712
   (e) -7.92047534
   (f) -7.46106309
   (g) -7.17076691
   (h) -6.90774795
   (i) -6.08067917
   (j) -5.53699377

   Their corresponding features:

   (a) log(UserNumReviews)
   (b) UserAverageStars*UserCoolVotes
   (c) sqrt(UserNumReviews*UserFunnyVotes)
   (d) sq(ReviewNumWords*UserNumReviews)
   (e) log(ReviewNumWords*ReviewInFall)
   (f) sqrt(ReviewInFall*UserNumReviews)
   (g) sqrt(ReviewNumLineBreaks*UserNumReviews)
   (h) log(UserNumReviews*BusinessNumCheckins)
   (i) sqrt(UserCoolVotes*IsMexican)
   (j) sqrt(ReviewInWinter*UserCoolVotes)

2.1 Programming: Binary Logistic Regression

5.

a. Consider $\mu_i(w,b)$, we can write the exponential term as follows.

$$\exp(-y_i(b+x_i^T w)) = \frac{1}{\mu_i(w,b)} - 1 = \frac{1-\mu_i(w,b)}{\mu_i(w,b)}$$

Consider the gradients $\nabla_w \mu_i(w,b)$ and $\nabla_b \mu_i(w,b)$.

$$\nabla_w \mu_i(w,b) = \frac{-\exp(-y_i(b+x_i^T w)) \times -y_i x_i}{[1+\exp(-y_i(b+x_i^T w))]^2} = \frac{[1-\mu_i(w,b)](y_i x_i)[\mu_i(w,b)]^2}{\mu_i(w,b)} = \mu_i(w,b)[1-\mu_i(w,b)](y_i x_i^T)$$

$$\nabla_b \mu_i(w,b) = \frac{-\exp(-y_i(b+x_i^T w)) \times -y_i}{[1+\exp(-y_i(b+x_i^T w))]^2} = \frac{[1-\mu_i(w,b)]y_i[\mu_i(w,b)]^2}{\mu_i(w,b)} = \mu_i(w,b)[1-\mu_i(w,b)]y_i$$

Derive the gradients $\nabla_w J(w,b)$ and $\nabla_b J(w,b)$.

$$\begin{aligned}
\nabla_w J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} \frac{\exp(-y_i(b+x_i^T w)) \times (-y_i x_i)}{1+\exp(-y_i(b+x_i^T w))} + 2\lambda w \\
&= \frac{1}{n}\sum_{i=1}^{n} \frac{1-\mu_i(w,b)}{\mu_i(w,b)} \times \mu_i(w,b) \times (-y_i x_i) + 2\lambda w \\
&= \frac{1}{n}\sum_{i=1}^{n} (1-\mu_i(w,b))(-y_i x_i) + 2\lambda w \\
\nabla_b J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} \frac{\exp(-y_i(b+x_i^T w)) \times (-y_i)}{1+\exp(-y_i(b+x_i^T w))} \\
&= \frac{1}{n}\sum_{i=1}^{n} \frac{1-\mu_i(w,b)}{\mu_i(w,b)} \times \mu_i(w,b) \times (-y_i) \\
&= \frac{1}{n}\sum_{i=1}^{n} (1-\mu_i(w,b))(-y_i)
\end{aligned}$$

Derive the Hessians $\nabla_w^2 J(w,b)$ and $\nabla_b^2 J(w,b)$.

$$\begin{aligned}
\nabla_w^2 J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} y_i x_i \nabla_w \mu_i(w,b) + 2\lambda I \\
&= \frac{1}{n}\sum_{i=1}^{n} y_i x_i \mu_i(w,b)[1-\mu_i(w,b)](y_i x_i^T) + 2\lambda I \\
&= \frac{1}{n}\sum_{i=1}^{n} (y_i)^2 \mu_i(w,b)[1-\mu_i(w,b)](x_i x_i^T) + 2\lambda I
\end{aligned}$$

$$\nabla_b^2 J(w,b) = \frac{1}{n}\sum_{i=1}^{n} y_i \nabla_b \mu_i(w,b) = \frac{1}{n}\sum_{i=1}^{n} y_i \mu_i(w,b)[1-\mu_i(w,b)]y_i = \frac{1}{n}\sum_{i=1}^{n} (y_i)^2 \mu_i(w,b)[1-\mu_i(w,b)]$$

b. Gradient Descent

(a) Plot of J(w,b)

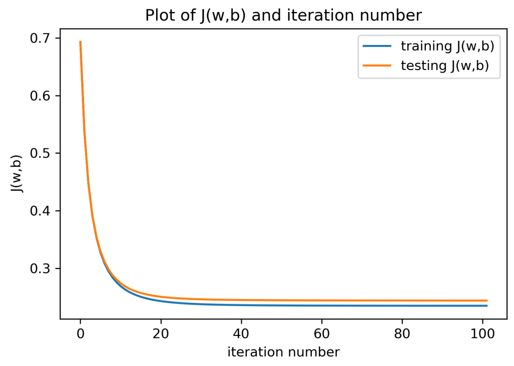

Figure 6: J(w,b) versus iteration number

(b) Plot of misclassification

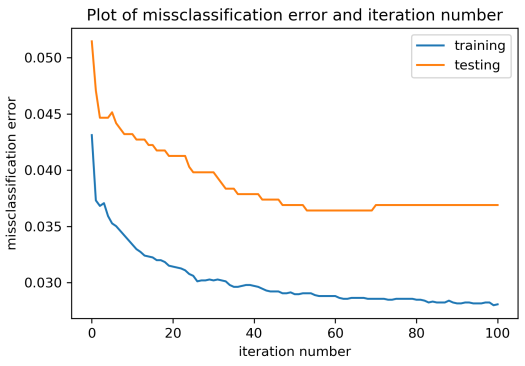

Figure 7: misclassification versus iteration number

c. Stochastic Gradient Descent with batch size = 1

(a) Plot of J(w,b)

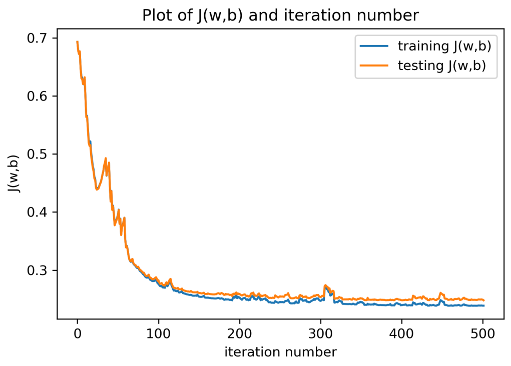

Figure 8: J(w,b) versus iteration number

(b) Plot of misclassification

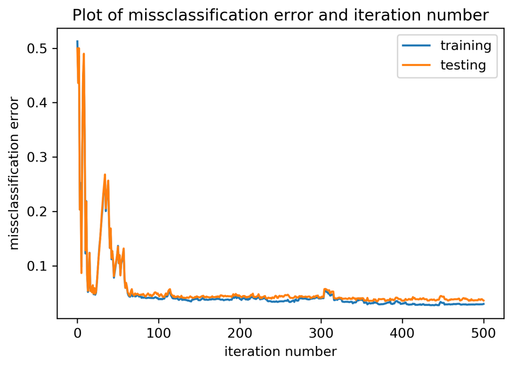

Figure 9: misclassification versus iteration number

d. Stochastic Gradient Descent with batch size = 100

(a) Plot of J(w,b)

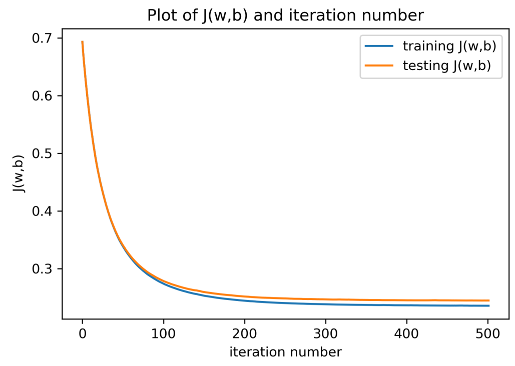

Figure 10: J(w,b) versus iteration number

(b) Plot of misclassification

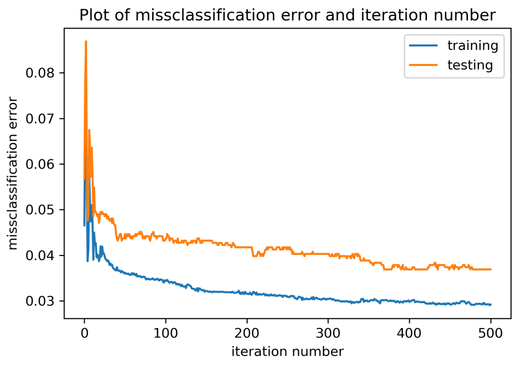

Figure 11: misclassification versus iteration number

e. Newton’s Method

   (a) Plot of J(w,b)

   (b) Plot of misclassification

The code for Problem 3:

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(546)

def cal_lambdaMax(x,y):
    mean_y = np.mean(y)
    lambda_array = np.zeros(d)
    for k in range(d):
        x_k = x[:,k]
        lambda_k = 2 * np.abs(np.sum(x_k * (y - mean_y), 0))
        lambda_array[k] = lambda_k
    return max(lambda_array)

def coord(tuning, delta, w_input, x, y):
    w_new = np.copy(w_input)
    w_prev = np.copy(w_input)
    no = 1
    check_delta = True
    print(tuning)
    while check_delta:
        w_prev = np.copy(w_new)
        no += 1
        b = np.mean(y - np.dot(x,w_prev))
        for j in range(d):
            a = 2 * np.sum(x[:,j]**2)
            wj = np.delete(w_new, j)
            xij = np.delete(x, j, axis=1)
            wj_xij = np.matmul(xij,wj)
            ck = 2 * np.dot(x[:,j],y - (b + wj_xij))

            if ck < -tuning:
                w_new[j] = (ck + tuning)/a
            elif ck > tuning:
                w_new[j] = (ck - tuning)/a
            else:
                w_new[j] = 0
        check_delta = any(np.abs(w_new - w_prev) > delta)
        if no > 30:
            check_delta = False
        print(no, end=",") # counting iterations, for reference
    return w_new

## Generate Data
n = 500
d = 1000
k = 100
sigma = 1
rate = 1.5
delta = 0.001
x = np.random.randn(n,d)
noise = np.random.randn(n) * (sigma**2)
w = np.zeros((d))
for j in range(k):
    w[j] = (j+1)/k
y = np.dot(x,w.T) + noise

lambda_max = cal_lambdaMax(x,y)
w_first = np.zeros((d,1))

lambda_array = lambda_max * (1/rate) ** np.arange(0,40)
nonzero = np.zeros(len(lambda_array))
fdr = []
tdr = []
w_old = w_first[:]
true_indices = np.arange(0,k)
for i in range(len(lambda_array)):
    print('i = ', i)
    tuning = lambda_array[i]
    print('lambda = ', tuning)
    w_next = coord(tuning, delta, w_old, x, y)
    w_old = w_next
    nnz = np.count_nonzero(w_next)
    nonzero[i] = nnz
    if nnz == 0:
        continue
    indices_nnz = np.where(w_next)[0]
    fdr.append(len(set(indices_nnz) - set(true_indices))/nnz)
    tdr.append(len(set(true_indices) & set(indices_nnz))/k)

print(nonzero)

fig1 = 'nnz_lambda.png'
plt.figure(dpi = 300)
plt.plot(lambda_array, nonzero)
plt.xscale('log')
plt.xlabel('lambda')
plt.ylabel('number of non-zeros')
plt.title('the number of non-zeros versus lambda')
plt.savefig(fig1)
plt.show()

fig2 = 'fdr_tdr.png'
plt.figure(dpi = 300)
plt.plot(fdr, tdr)
plt.xlabel('FDR')
plt.ylabel('TDR')
plt.title('FDR vs TDR')
plt.savefig(fig2)
plt.show()
```

The code for Problem 4:

```
"""
Problem 4
"""
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(546)

def cal_lambdaMax(x,y):
    mean_y = np.mean(y)
    lambda_array = np.zeros(d)
    for k in range(d):
        x_k = x[:,k]
        lambda_k = 2 * np.abs(np.sum(x_k * (y - mean_y), 0))
        lambda_array[k] = lambda_k
    return max(lambda_array)

def coord(tuning, delta, w_input, x, y):
    w_new = np.copy(w_input)
    w_prev = np.copy(w_input)
    no = 1
    check_delta = True
    print(tuning)
    while check_delta:
        w_prev = np.copy(w_new)
        no += 1
        b = np.mean(y - np.dot(x,w_prev))
        for j in range(d):
            a = 2 * np.sum(x[:,j]**2)
            wj = np.delete(w_new, j)
            xij = np.delete(x, j, axis=1)
            wj_xij = np.dot(xij,wj)
            ck = 2 * np.dot(x[:,j],y - (b + wj_xij))

            if ck < -tuning:
                w_new[j] = (ck + tuning)/a
            elif ck > tuning:
                w_new[j] = (ck - tuning)/a
            else:
                w_new[j] = 0
        print("w_max␣=",np.max(w_new))
        check_delta = any(np.abs(w_new - w_prev) > delta)
#           if no > 30:
#               check_delta = False
        print(no, end=",") # counting iterations, for reference
    max_delta = max(np.abs(w_new - w_prev))
    print(max_delta) # sanity check
    return w_new

# Param
d = 1000
rate = 1.5
delta = 0.1
```

```
# Load a csv of floats:
X = np.genfromtxt("upvote_data.csv", delimiter=",")
# Load a text file of integers:
y = np.loadtxt("upvote_labels.txt", dtype=np.int)
# Load a text file of strings:
featureNames = open("upvote_features.txt").read().splitlines()

## pre-processing
y = np.sqrt(y)

n = X.shape[0]
n_train = 4000
n_valid = 1000

X_train = X[0:n_train,:]
y_train = y[0:n_train]
X_valid = X[n_train:(n_train+n_valid),:]
y_valid = y[n_train:(n_train+n_valid)]
X_test = X[(n_train+n_valid):n,:]
y_test = y[(n_train+n_valid):n]

lambda_max = cal_lambdaMax(X_train, y_train)
w_first = np.zeros((d,1))
lambda_array = lambda_max * (1/rate) ** np.arange(0,21)
w_old = w_first[:]
nonzero = np.zeros(len(lambda_array))
sq_error_tr = np.zeros(len(lambda_array))
sq_error_val = np.zeros(len(lambda_array))
w_store = []
## Choosing the appropriate lambda
for i in range(len(lambda_array)):
    print('i = ', i)
    tuning = lambda_array[i]
    w_new = coord(tuning, delta, w_old, X_train, y_train)
    w_store.append(w_new)
    y_est_train = np.dot(w_new.T, X_train.T)
    y_est_val = np.dot(w_new.T, X_valid.T)
    nonzero[i] = np.count_nonzero(w_new)
    sq_error_tr[i] = np.sum((y_est_train - y_train)**2)
    sq_error_val[i] = np.sum((y_est_val - y_valid)**2)
    w_old = w_new

print(sq_error_tr)
print(sq_error_val)

fig3 = 'sq_error_yelp.png'
fig4 = 'nnz_lambda_yelp.png'

plt.figure(dpi = 300)
plt.plot(lambda_array, sq_error_tr/(X_train.shape[0]**2), label = "training error")
plt.plot(lambda_array, sq_error_val/(X_valid.shape[0]**2), label = "validating error")
plt.xscale('log')
plt.gca().invert_xaxis()
plt.legend()
plt.xlabel('lambda')
plt.ylabel('squared error')
plt.title('Squared error on training and validation data versus lambda')
plt.savefig(fig3)

plt.figure(dpi = 300)
plt.plot(lambda_array, nonzero)
plt.gca().invert_xaxis()
plt.xscale('log')
plt.xlabel('lambda')
plt.ylabel('number of nonzeros')
plt.title('the number of non-zeros versus lambda')
plt.savefig(fig4)


###
# Part b - Get lambda from best validation performance
# Find w_new using train and validation data
# Find error for train, val, test
###
lambda_index = np.argmin(sq_error_val/(X_valid.shape[0]**2))
tuning = lambda_array[lambda_index]
w_select = w_store[lambda_index]
y_est_test = np.dot(w_select.T, X_test.T)
sq_error_train_model = sq_error_tr[lambda_index]/(X_train.shape[0]**2)
sq_error_val_model = sq_error_val[lambda_index]/(X_valid.shape[0]**2)
sq_error_test_model = np.mean((y_est_test - y_test)**2)
print(sq_error_train_model)
print(sq_error_val_model)
print(sq_error_test_model)

###
# Part c - Check w_select
# Find w_new using train and validation data
# Find error for train, val, test
###
sort_indices = np.argsort(w_select.T)
ten_indices = sort_indices[0][0:10]
print('ten largest weight:')
w_select[ten_indices]
print('ten features:')
for i in range(10):
    print(featureNames[ten_indices[i]])
```

The code for Problem 5:

```
"""
Problem 5
"""
from mnist import MNIST
import numpy as np
import matplotlib.pyplot as plt

def load_dataset():
    mndata = MNIST('./python-mnist/data')
    X_train, labels_train = map(np.array, mndata.load_training())
    X_test, labels_test = map(np.array, mndata.load_testing())
    X_train = X_train/255.0
    X_test = X_test/255.0
    return X_train, X_test, labels_train, labels_test

def sigmoid(y,x,wt):
    sig = 1/(1 + np.exp(-y * np.matmul(x, wt)))
    return sig

def gradient_descent(tuning, step, x_train, y_train, x_test, y_test):
    j_train = []
    j_test = []
    miss_train = []
    miss_test = []
    d = x_train.shape[1]
    w = np.zeros(d)
    b = 0
    no = 0
    jvalue_train = np.mean(np.log(1 + np.exp(-y_train * (b + np.matmul(x_train, w))))) + tur
    j_train.append(jvalue_train)
    j_test_value = np.mean(np.log(1 + np.exp(-y_test * (b + np.matmul(x_test, w))))) + tunin
    j_test.append(j_test_value)
    while no <= 100 :
        print("iter␣=␣", no)
        print("Jvalue␣=␣", jvalue_train)
        no += 1
        mu = sigmoid(y_train,x_train,w)
        gradw_j = (1/x_train.shape[0]) * np.dot(-y_train * (1 - mu), x_train) + 2 * tuning *
        gradb_j = np.mean(-y_train * (1 - mu))

        w = w - step * gradw_j
        b = b - step * gradb_j

        jvalue_train = np.mean(np.log(1 + np.exp(-y_train * (b + np.matmul(x_train, w))))) +
        jvalue_test = np.mean(np.log(1 + np.exp(-y_test * (b + np.matmul(x_test, w))))) + tu
        pred_y_train = b + np.matmul(x_train, w)
        pred_y_test = b + np.matmul(x_test, w)
        sign_y_train = np.sign(pred_y_train)
        sign_y_test = np.sign(pred_y_test)

        tr_error = (np.size(sign_y_train == y_train) - np.sum(sign_y_train == y_train)) / np
            sign_y_train == y_train)
        te_error = (np.size(sign_y_test == y_test) - np.sum(sign_y_test == y_test)) / np.siz
            sign_y_test == y_test)
        miss_train.append(tr_error)
        miss_test.append(te_error)
        j_train.append(jvalue_train)
        j_test.append(jvalue_test)
    return j_train, j_test, miss_train, miss_test, w

def sgd(tuning, step, x_train, y_train, x_test, y_test, batch):
    j_train = []
    j_test = []
    miss_train = []
    miss_test = []
    d = x_train.shape[1]
    w = np.zeros(d)
    b = 0
    no = 0
    jvalue_train = np.mean(np.log(1 + np.exp(-y_train * (b + np.matmul(x_train, w))))) + tun
    j_train.append(jvalue_train)
    j_test_value = np.mean(np.log(1 + np.exp(-y_test * (b + np.matmul(x_test, w))))) + tunin
    j_test.append(j_test_value)
    while no <= 500 :
        print("iter␣=␣", no)
        print("Jvalue␣=␣", jvalue_train)
        no += 1

        select = np.random.choice(np.arange(x_train.shape[0]), batch)
        mu = sigmoid(y_train[select],x_train[select,:],w)
        gradw_j = (1/batch) * np.dot(-y_train[select] * (1 - mu), x_train[select,:]) + 2 * t
        gradb_j = np.mean(-y_train[select] * (1 - mu))

        w = w - step * gradw_j
        b = b - step * gradb_j

        jvalue_train = np.mean(np.log(1 + np.exp(-y_train * (b + np.matmul(x_train, w))))) +
        jvalue_test = np.mean(np.log(1 + np.exp(-y_test * (b + np.matmul(x_test, w))))) + tu
        pred_y_train = b + np.matmul(x_train, w)
        pred_y_test = b + np.matmul(x_test, w)
        sign_y_train = np.sign(pred_y_train)
        sign_y_test = np.sign(pred_y_test)

        tr_error = (np.size(sign_y_train == y_train) - np.sum(sign_y_train == y_train)) / np
            sign_y_train == y_train)
        te_error = (np.size(sign_y_test == y_test) - np.sum(sign_y_test == y_test)) / np.siz
            sign_y_test == y_test)

        miss_train.append(tr_error)
        miss_test.append(te_error)
        j_train.append(jvalue_train)
        j_test.append(jvalue_test)
    return j_train, j_test, miss_train, miss_test, w

def newton():
    return w_new

## Obtain data only for 2 and 7
```

```
X_train, X_test, labels_train, labels_test = load_dataset()
seven_train = labels_train == 7
two_train = labels_train == 2
seven_test = labels_test == 7
two_test = labels_test == 2

x_train = np.concatenate((X_train[seven_train], X_train[two_train]))
y_train = np.concatenate((labels_train[seven_train], labels_train[two_train]))
y_train = y_train.astype(int)
len_seven_train = len(labels_train[seven_train])
y_train[0:(len_seven_train+1)] = 1
y_train[len_seven_train:len(y_train)] = -1
x_test = np.concatenate((X_test[seven_test], X_test[two_test]))
y_test = np.concatenate((labels_test[seven_test], labels_test[two_test]))
y_test = y_test.astype(int)
len_seven_test = len(labels_test[seven_test])
y_test[0:(len_seven_test+1)] = 1
y_test[len_seven_test+1:len(y_test)] = -1
xtrain_avg = np.mean(x_train, axis = 0)

###
# Part b. Gradient Descent
###
n_feature = x_train.shape[1]
w_first = np.zeros(n_feature)
b_first = 0.0
tuning = 0.1
step = 0.1
j_train, j_test, miss_train, miss_test, w = gradient_descent(tuning, step, x_train, y_train,

fig = 'jvalue_gradient.png'
plt.figure(dpi = 300)
plt.plot(j_train, label = "training J(w,b)")
plt.plot(j_test, label = "testing J(w,b)")
plt.legend()
plt.xlabel('iteration number')
plt.ylabel('J(w,b)')
plt.title('Plot of J(w,b) and iteration number')
plt.savefig(fig)

fig = 'missclass_gradient.png'
plt.figure(dpi = 300)
plt.plot(miss_train, label = "training")
plt.plot(miss_test, label = "testing")
plt.legend()
plt.xlabel('iteration number')
plt.ylabel('misclassification error')
plt.title('Plot of misclassification error and iteration number')
plt.savefig(fig)

###
# Part c. Stochastic Gradient Descent with bach size = 1
###
batch = 1
step = 0.01
```

```
j_train, j_test, miss_train, miss_test, w = sgd(tuning, step, x_train, y_train, x_test, y_te
fig = 'jvalue_sgd.png'
plt.figure(dpi = 300)
plt.plot(j_train, label = "training J(w,b)")
plt.plot(j_test, label = "testing J(w,b)")
plt.legend()
plt.xlabel('iteration number')
plt.ylabel('J(w,b)')
plt.title('Plot of J(w,b) and iteration number')
plt.savefig(fig)

fig = 'missclass_sgd.png'
plt.figure(dpi = 300)
plt.plot(miss_train, label = "training")
plt.plot(miss_test, label = "testing")
plt.legend()
plt.xlabel('iteration number')
plt.ylabel('misclassification error')
plt.title('Plot of misclassification error and iteration number')
plt.savefig(fig)

###
# Part d. Stochastic Gradient Descent with bach size = 100
###
batch = 100
step = 0.01
j_train, j_test, miss_train, miss_test, w = sgd(tuning, step, x_train, y_train, x_test, y_te
fig = 'jvalue_sgd100.png'
plt.figure(dpi = 300)
plt.plot(j_train, label = "training J(w,b)")
plt.plot(j_test, label = "testing J(w,b)")
plt.legend()
plt.xlabel('iteration number')
plt.ylabel('J(w,b)')
plt.title('Plot of J(w,b) and iteration number')
plt.savefig(fig)

fig = 'missclass_sgd100.png'
plt.figure(dpi = 300)
plt.plot(miss_train, label = "training")
plt.plot(miss_test, label = "testing")
plt.legend()
plt.xlabel('iteration number')
plt.ylabel('misclassification error')
plt.title('Plot of misclassification error and iteration number')
plt.savefig(fig)


###
# Part e. Newton's Method
###
```