CSE 546 HW #2

Sam Kowash

November 10, 2018

Acknowledgments: I collaborated on parts of this homework with Tyler Blanton, Michael Ross, and Nick Ruof.

1 A Taste of Learning Theory

1. Let $X \in \mathbb{R}^d$ a random feature vector, and $Y \in \{1, \dots, K\}$ a random label for $K \in \mathbb{N}$ with joint distribution P_{XY}. We consider a randomized classifier $\delta(x)$ which maps a value $x \in \mathbb{R}^d$ to some $y \in \{1, \dots, K\}$ with probability $\alpha(x, y) \equiv P(\delta(x) = y)$ subject to $\sum_{y=1}^K \alpha(x, y) = 1$ for all x. The risk of the classifier δ is

$$R(\delta) \equiv \mathbb{E}_{XY,\delta}\left[\mathbf{1}\{\delta(X) \neq Y\}\right],$$

which we should interpret as the expected rate of misclassification. A classifier δ is called deterministic if $\alpha(x, y) \in \{0, 1\}$ for all x, y. Further, we call a classifier δ_* a Bayes classifier if $\delta_* \in \arg\inf_\delta R(\delta)$.

If we first take the expectation over outcomes of δ (by conditioning on X and Y), we find

$$R(\delta) = \mathbb{E}_{XY}\left[1 - \alpha(X, Y)\right],$$

since the indicator function is 1 except for the single outcome where $\delta(x) = y$, which occurs with probability $\alpha(x, y)$. It is then clear that minimizing $R(\delta)$ is equivalent to *maximizing* $\mathbb{E}_{XY}[\alpha(X, Y)]$; the assignments of $\alpha(x, y)$ which do this are our Bayes optimal classifiers.

Define $p_k(x) = \mathbb{P}(Y = k \mid X = x)$ and $\alpha_k(x) = \alpha(X, k)$. Then

$$\mathbb{E}_{XY}(\alpha(X, Y)) = \mathbb{E}_X\left[\mathbb{E}_{Y|X}\left[\alpha(x, Y) \mid X = x\right]\right] = \mathbb{E}_X\left[\sum_{k=1}^K p_k(X)\alpha_k(X)\right]$$

Fixing an x for a moment, note that

$$\sum_{k=1}^K p_k\alpha_k \leqslant \sum_{k=1}^K \alpha_k \max_i\{p_i\} = \max_i\{p_i\} \sum_{k=1}^K \alpha_k = \max_i\{p_i\},$$

so any δ which saturates this bound for each x is a Bayes classifier. This is clearly achieved by any assignment satisfying $\alpha(x, i) = 0$ for $i \notin \arg\max_k p_k(x)$. A deterministic member of this class is the rule that always predicts the smallest $i \in \arg\max_k p_k(x)$.

2. Suppose we grab n data samples (x_i, y_i) i.i.d. from P_{XY} where $y_i \in \{-1, 1\}$ and $x_i \in \mathcal{X}$ where $\mathcal{X}$ is some set. Let $f : \mathcal{X} \to \{-1, 1\}$ be a deterministic classifier with true risk

$$R(f) = \mathbb{E}_{XY}\left[\mathbf{1}(f(X) \neq Y)\right].$$

and empirical risk

$$\hat{R}_n(f) = \frac{1}{n}\sum_{i=1}^n \mathbf{1}(f(x_i) \neq y_i).$$

(a) We wish to estimate the true risk of some classifier $\tilde{f}$. The empirical risk $\hat{R}_n(\tilde{f})$ is an average of n i.i.d. random variables—the indicator functions $\mathbf{1}(f(x_i) \neq y_i)$—and the true risk is the expected value of each of those random variables. This satisfies the hypotheses of Hoeffding's inequality, so for any $\epsilon \geqslant 0$ we have

$$\mathbb{P}\left(\left|\hat{R}_n(\tilde{f}) - R(\tilde{f})\right| \geqslant \epsilon\right) \leqslant 2e^{-2n\epsilon^2}.$$

We can rewrite this probability in terms of its complement, giving

$$\begin{aligned} 1 - \mathbb{P}\left(\left|\hat{R}_n(\tilde{f}) - R(\tilde{f})\right| < \epsilon\right) &\leqslant 2e^{-2n\epsilon^2} \\ \mathbb{P}\left(\left|\hat{R}_n(\tilde{f}) - R(\tilde{f})\right| < \epsilon\right) &\geqslant 1 - 2e^{-2n\epsilon^2}. \end{aligned}$$

Thus, with confidence $1 - \delta$, the empirical risk falls within A of the true loss, where

$$A = \sqrt{-\frac{\ln \delta/2}{2n}}.$$

(b) Let $\mathcal{F} = \{f_1, \ldots, f_k\}$ be a set of classifying rules that is devised based on a prior hypothesis about P_{XY}. Since the classifiers are functions of a single variable, the set of indicator variables remains independent; the distribution of the ith indicator is not "tainted" by information about any other. The hypotheses of Hoeffding's inequality are just as well satisfied by any member of $\mathcal{F}$ as they were by the arbitrary $\tilde{f}$ considered above, and the previous confidence interval holds. In particular, it holds for the best-in-class function

$$f^* = \arg\min_{f \in \mathcal{F}} R(f).$$

(c) We now consider the empirical risk minimizer

$$\hat{f} = \arg\min_{f \in \mathcal{F}} \hat{R}_n(f).$$

Although we used our data to select $\hat{f}$ from the class $\mathcal{F}$, there is still no dependence between the indicators $\mathbf{1}\{f(x_i) \neq y_i\}$, because the value of $f(x_i)$ does not depend on any y_i or $x_{j \neq i}$ and the draws $\{(x_i, y_i)\}$ are independent of each other by hypothesis. Thus, we can proceed just as in the original case and obtain the same confidence interval on the deviation of $\hat{R}_n(\hat{f})$ from $R(\hat{f})$.

(d) I'm not sure that there is such an example with the previous assumption that $\mathcal{F}$ is finite.

If $\mathcal{X}$ is finite, then there is nonzero probability to sample the same element twice. To achieve zero empirical risk, P_{XY} must never allow us to sample the same x with different ys. But, since there is nonzero probability to sample all of $\mathcal{X}$ for $n \geqslant |\mathcal{X}|$, zero empirical risk is only possible if $\mathcal{F}$ contains a function that maps every x to its sole allowed y, but that function clearly has both zero empirical risk *and* zero true risk, contradicting the problem statement.

If $\mathcal{X}$ is countable, the double-sampling issue still demands that P_{XY} uniquely determine y for each x. We don't have to worry about sampling the entire set, but we now seem to need an infinite function class to achieve zero empirical risk. If uncountable, it seems even worse. If this is possible, I'm really curious as to how.

(e) To find a confidence interval that can be satisfied by all $f \in \mathcal{F}$ simultaneously with probability at least $1 - \delta$, note that for $\epsilon \geqslant 0$,

$$\mathbb{P}\left[\bigcup_{i=1}^{k} |\hat{R}_n(f_i) - R(f_i)| \geqslant \epsilon\right] \leqslant \sum_{i=1}^{k} \mathbb{P}\left(|\hat{R}_n(f_i) - R(f_i)| \geqslant \epsilon\right)$$

by the union bound, and applying Hoeffding's inequality to each term as justified above,

$$\mathbb{P}\left[\bigcup_{i=1}^{k} |\hat{R}_n(f_i) - R(f_i)| \geqslant \epsilon\right] \leqslant \sum_{i=1}^{k} 2e^{-2n\epsilon^2} = 2ke^{-2n\epsilon^2}.$$

Using De Morgan's law to identify the complement of the event on the left-hand side, we rewrite just as before to get

$$\begin{aligned} 1 - \mathbb{P}\left[\bigcap_{i=1}^{k} |\hat{R}_n(f_i) - R(f_i)| < \epsilon\right] &\leqslant 2ke^{-2n\epsilon^2} \\ \mathbb{P}\left[\bigcap_{i=1}^{k} |\hat{R}_n(f_i) - R(f_i)| < \epsilon\right] &\geqslant 1 - 2ke^{-2n\epsilon^2}. \end{aligned}$$

From this we can see that

$$\mathbb{P}\left(\text{for all } f \in \mathcal{F},\ |\hat{R}_n(f) - R(f)| \leqslant B\right) \geqslant 1 - \delta,$$

where

$$\begin{aligned} 1 - \delta &= 1 - 2ke^{-2nB^2} \\ \ln\frac{\delta}{2k} &= -2nB^2 \\ B &= \sqrt{-\frac{\ln \delta/(2k)}{2n}}. \end{aligned}$$

(f) The confidence interval for the ERM should be exactly the same as that found in part (a), as could also be seen by considering the degenerate class of one function, which gives $k = 1$ in the above result and reproduces that of part (a).

(g) Because $\hat{f}$ minimizes empirical risk and f^* the true risk, we have

$$\begin{aligned} 0 &\leqslant R(\hat{f}) - R(f^*) \\ 0 &\leqslant \hat{R}_n(f^*) - \hat{R}_n(\hat{f}), \end{aligned}$$

and so

$$0 \leqslant R(\hat{f}) - R(f^*) \leqslant [\hat{R}_n(f^*) - R(f^*)] - [\hat{R}_n(\hat{f}) - R(\hat{f})].$$

Then also

$$0 \leqslant R(\hat{f}) - R(f^*) \leqslant \left|\hat{R}_n(f^*) - R(f^*)\right| + \left|\hat{R}_n(\hat{f}) - R(\hat{f})\right|.$$

If we assume $\hat{f} \neq f^*$, then the previous result tells us that for any threshold δ,

$$\mathcal{P}\left[\left(\left|\hat{R}_n(f^*) - R(f^*)\right| \leqslant B\right) \cap \left(\left|\hat{R}_n(\hat{f}) - R(\hat{f})\right| \leqslant B\right)\right] \geqslant 1 - \delta,$$

where

$$B = \sqrt{-\frac{\ln \delta/4}{2n}}$$

since we are considering a class of $k = 2$ functions. Thus, we can say that with probability at least $1 - \delta$,

$$\left|\hat{R}_n(f^*) - R(f^*)\right| + \left|\hat{R}_n(\hat{f}) - R(\hat{f})\right| \leqslant C = 2B = 2\sqrt{-\frac{\ln \delta/4}{2n}}.$$

(h) Fix $f \in \mathcal{F}$ and suppose $R(f) \geqslant \epsilon$. This implies that the probability for an arbitrary (x, y) to be correctly classified by f is no greater than $1 - \epsilon$, so the probability of correctly classifying a set of n points (that is, $\mathbb{P}(\hat{R}_n(f) = 0)$) is no greater than $(1 - \epsilon)^n$. Note (recalling $0 < \epsilon$) that if $\epsilon < 1$,

$$\ln\left((1-\epsilon)^n\right) = n \ln(1 - \epsilon) = -n \sum_{k=1}^{\infty} \frac{\epsilon^k}{k} \leqslant -n\epsilon,$$

so that

$$(1 - \epsilon)^n \leqslant e^{-n\epsilon}.$$

We patch the hole in the above by noting that this relation holds if we plug in $\epsilon = 1$ as well. Thus,

$$\mathbb{P}(\hat{R}_n(f) = 0) \leqslant e^{-n\epsilon}.$$

Unsure how to continue this to compute regret.

(i) Regret.

2 Programming

3. We generated $n = 500$ points $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$ according to the model $y_i = w^T x_i + \epsilon_i$, where

$$w_j = \begin{cases} j/k & \text{if } j \in \{1, \dots, k\} \\ 0 & \text{otherwise} \end{cases}$$

and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ with $k = 100$, $d = 1000$, and $\sigma = 1$. We then computed a regularization path on this data for the parameter λ in the lasso objective as shown in Fig. 2.1 and observed the false discovery and true positive rates for the relevant/irrelevant features as seen in Fig. 2.2. As λ increases (to the left in Fig. 2.1), fewer and fewer features are selected, and vice versa, with most features selected by $\lambda \sim 10^{-3}$. Also as λ grows, the false discovery rate falls as features are eliminated, and the true positive rate falls to some extent with it.

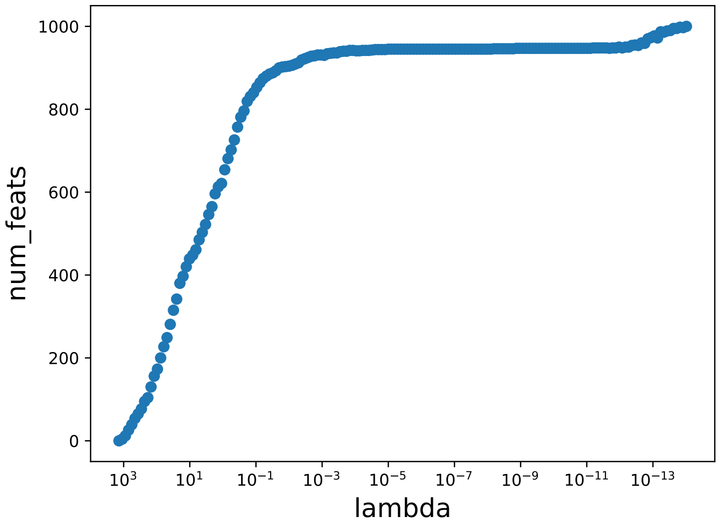

Figure 2.1: Number of nonzero features vs. λ for lasso on synthetic data

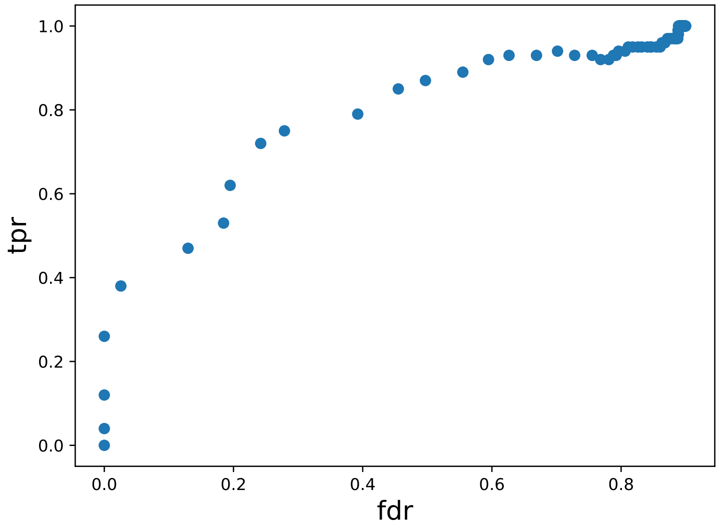

Figure 2.2: TPR vs. FDR for lasso on synthetic data

4. (a) We next computed a regularization path on the provided Yelp data, depicted in Fig. 2.4, and used a validation holdout set to assess regression performance, shown in 2.3. We found that validation error was best around $\lambda = 1.11$.

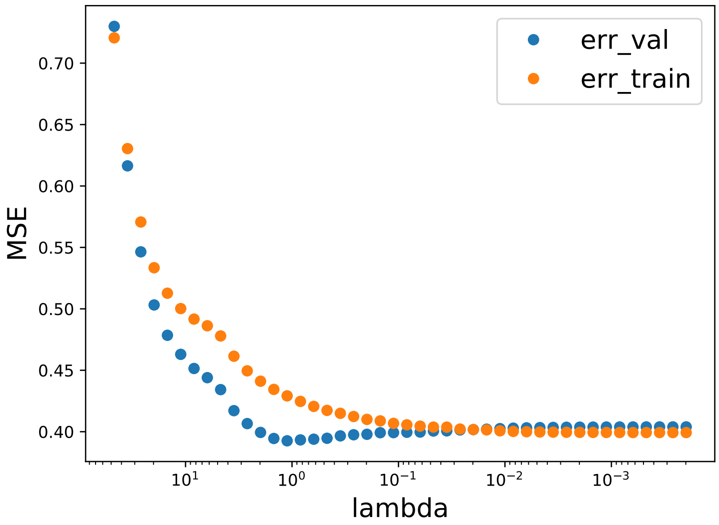

Figure 2.3: Validation and training error vs λ on Yelp data

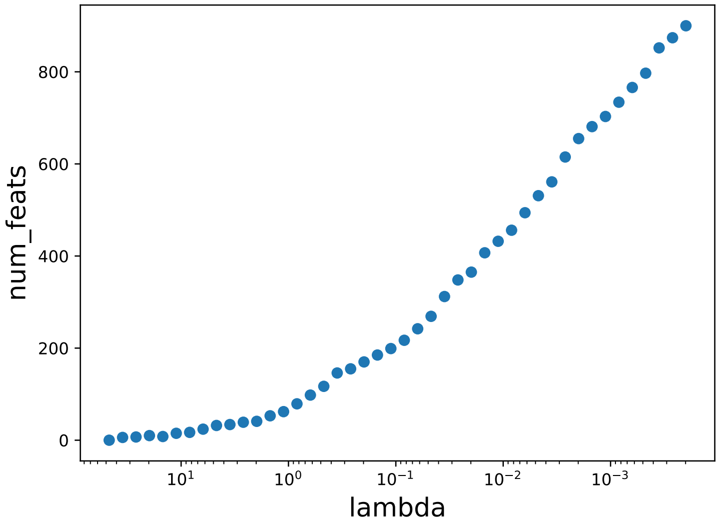

Figure 2.4: Number of nonzero features vs. λ on Yelp data

(b)

(c)

5. We lastly consider binary classification between 2s and 7s in the MNIST set via regularized logistic regression. We choose a balanced target set $Y \in \{-1, 1\}$, where $Y = -1$ for 2s and $Y = 1$ for 7s, so that our data are $\{(x_i, y_i)\}_{i=1}^n \subset \mathbb{R}^d \times \mathbb{Z}_2$. The L_2-regularized negative log likelihood objective to be minimized is

$$J(w, b) = \frac{1}{n} \sum_{i=1}^{n} \log \left[1 + \exp\left(-y_i(b + x_i^T w)\right)\right] + \lambda \|w\|_2^2.$$

For convenience, we define the functions

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

.

(a) To do gradient descent, we need to know some gradients. First,

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

Next,

$$\begin{aligned}
\nabla_b J(w,b) &= -\frac{1}{n}\sum_{i=1}^{n} \frac{y_i \exp\left[-y_i(b+x_i^T w)\right]}{1+\exp\left[-y_i(b+x_i^T w)\right]} \\
\nabla_b J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} (\mu_i - 1) y_i.
\end{aligned}$$

We'll also want some Hessians, for Newton's method.

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

Lastly,

$$\begin{aligned}
\nabla_b^2 J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} (\nabla_b \mu_i) y_i \\
\nabla_b \mu_i &= \frac{y_i \exp\left[-y_i(b+x_i^T w)\right]}{\left(1+\exp\left[-y_i(b+x_i^T w)\right]\right)^2} = \mu_i(1-\mu_i) y_i \\
\nabla_b^2 J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} \mu_i(1-\mu_i) y_i^2.
\end{aligned}$$

```
#!/usr/bin/env python
# lasso.py
import numpy as np
import matplotlib.pyplot as plt


def descend_step(X, Y, w_curr, lam):
    d = X.shape[1]
    w = w_curr
    b = np.mean(Y - np.matmul(X,np.transpose(w)))

    xks = {k: X[:,k] for k in range(d)}
    a = {k: 2*np.matmul(xks[k],xks[k]) for k in range(d)}

    for k in range(d):
        wk = np.copy(w)
        xk = xks[k]
        wk[k] = 0 #faster way to knock out a column

        c = 2*np.matmul(xk, Y - b - np.matmul(X, np.transpose(wk)))

        if np.abs(c) <= lam:
            w[k] = 0
        else:
            w[k] = (c - np.sign(c)*lam)/a[k]

    return w



def lasso_descend(X, Y, w_init, lam, thresh):
    n = X.shape[0]
    d = X.shape[1]

    if len(w_init) != d:
        print("Initial guess dimension mismatch. Setting all zeros.")
        w_init = np.zeros(d)


    #do at least one step
    w_last = np.copy(w_init)
    w_curr = descend_step(X, Y, w_init, lam)

    i=0
    while np.max(np.abs(w_curr-w_last)) > thresh:
        i += 1
        print("on descent step %s" %i)
        w_last = np.copy(w_curr)
    w_curr = descend_step(X, Y, w_curr, lam)
b = np.mean(Y - np.matmul(X,np.transpose(w_curr)))
return w_curr, b
```

```
#!/usr/bin/env python
# synth_lasso.py
import numpy as np
import matplotlib.pyplot as plt
import sys
from datetime import datetime
from lasso import *






n = 500
d = 1000
k = 100
sig = 1

w_true = np.append(np.arange(1,k+1)/k, np.zeros(d-k))

X_train = np.random.randn(n,d)
Y_train = np.matmul(X_train, np.transpose(w_true)) + np.random.randn(n)*sig

lam_max = np.max(2*np.abs(np.matmul(Y_train - np.mean(Y_train), X_train)))


lams = []
ws = []
num_feats = []
tprs = []
fdrs = []
xdrs = []
lam = lam_max

r = float(sys.argv[1])


it = 0
while (max(num_feats) if num_feats else 0) < d:
    it += 1
    print(f"On iter {it} with {num_feats[-1] if num_feats else 0} "
          f"features and lambda={lam:.5f}")
    lams.append(lam)
    w = lasso_descend(X_train, Y_train, ws[-1] if ws else np.zeros(d),
                      lam, 1e-3)[0]

    total_feats = np.count_nonzero(w)
    true_feats = np.count_nonzero(np.logical_and(w != 0, w_true != 0))
    false_feats = np.count_nonzero(np.logical_and(w != 0, w_true == 0))
    if total_feats != (true_feats + false_feats):
        print("Something␣is␣terribly␣wrong.")

    num_feats.append(total_feats)
    tprs.append(true_feats/k)
    fdrs.append(false_feats/total_feats if total_feats != 0 else 0)
    ws.append(w)
    lam *= r

lams = np.array(lams)
num_feats = np.array(num_feats)
tprs = np.array(tprs)
fdrs = np.array(fdrs)


ftime = datetime.now().time()
stamp = f"{ftime.hour:02d}_{ftime.minute:02d}_{ftime.second:02d}"
with open(f'data/synth-{stamp}','w') as f:
    f.writelines([f"{lams[i]:12e}␣{num_feats[i]:4d}␣"
                  f"{tprs[i]:14.8f}␣{fdrs[i]:14.8f}\n"
                  for i in range(len(lams))])
```

```
#!/usr/bin/env python
# yelp_lasso.py
import numpy as np
import matplotlib.pyplot as plt
import sys
from datetime import datetime
from lasso import *





# Load data according to provided example
X = np.genfromtxt("data/upvote_data.csv", delimiter=",")
Y = np.loadtxt("data/upvote_labels.txt", dtype=np.int)
feature_names = open("data/upvote_features.txt").read().splitlines()

print("Data␣loaded.")

d = X.shape[1]

X_train = X[:4000]
Y_train = np.sqrt(Y[:4000])

X_val = X[4000:5000]
Y_val = np.sqrt(Y[4000:5000])

X_test = X[5000:]
Y_test = np.sqrt(Y[5000:])


lam_max = np.max(2*np.abs(np.matmul(Y_train - np.mean(Y_train), X_train)))


lams = []
num_feats = []
val_errs = []
train_errs = []
ws = []
bs = []

lam = lam_max
r = float(sys.argv[1])

it = 0
while (max(num_feats) if num_feats else 0) < .9*d:
    it += 1
    print(f"On␣iter␣{it}␣with␣{num_feats[-1]␣if␣num_feats␣else␣0}␣"
          f"features and lambda={lam:5e}")
    lams.append(lam)
    w,b = lasso_descend(X_train, Y_train, (ws[-1] if ws else np.zeros(d)),
                        lam, 5e-2)
    ws.append(w)
    bs.append(b)


    val_pred = np.matmul(X_val,w) + b
    train_pred = np.matmul(X_train, w) + b

    val_diff = val_pred - Y_val
    train_diff = train_pred - Y_train

    feats = np.count_nonzero(w)
    val_err = np.matmul(val_diff, val_diff)
    train_err = np.matmul(train_diff, train_diff)

    num_feats.append(feats)
    val_errs.append(val_err)
    train_errs.append(train_err)

    lam *= r



ftime = datetime.now().time()
stamp = f"{ftime.hour:02d}_{ftime.minute:02d}_{ftime.second:02d}"
with open(f"data/yelp_sqrt-{stamp}", "w") as f:
    f.writelines([f"{lams[i]:12e} {num_feats[i]:4d} {val_errs[i]:14.6f} "
                  f"{train_errs[i]:14.6f}\n" for i in range(len(lams))])
```

```
#!/usr/bin/env python
# grad_descend.py
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST
from datetime import datetime
import sys


def load_test():
    mndata = MNIST('../../mnist/data/')
    X_test, labels_test = map(np.array, mndata.load_testing())
    X_test = X_test/255.

    return X_test, labels_test

def load_train():
    mndata = MNIST('../../mnist/data/')
    X_train, labels_train = map(np.array, mndata.load_training())
    X_train = X_train/255.

    return X_train, labels_train

def grad_w(mu, X, Y, w, lam):
    n = len(mu)
    return np.dot((mu-1)*Y,X)/n + 2*lam*w

def grad_b(mu, X, Y, w, lam):
    n = len(mu)
    return np.dot(mu-1, Y)/n

def hess_w(mu, X, Y, w, lam):
    n = len(mu)
    d = X.shape[1]
    hess = np.zeros(d,d)
    for i in range(n):
        hess += mu[i]*(1-mu[i])*Y[i]**2*np.outer(X[i],X[i])

    return hess/n + 2*lam*np.identity(d)

def hess_b(mu, X, Y, w, lam):
    n = len(mu)

    return np.sum(mu*(1-mu)*Y**2)/n


def objective(mu, X, Y, w, lam):
    n = len(mu)
    return np.sum(-1*np.log(mu))/n + lam*np.matmul(w,w)


def pred(X, w, b):
    return np.sign(b+ np.matmul(X,w))


def gdescend(X_train, Y_train, X_test, Y_test, lam=0.1, eta = 0.4, tol = 1e-4):
    d = X_train.shape[1]
    n_train = X_train.shape[0]
    n_test = X_test.shape[0]

    j_train = []
    j_test = []
    e_train = []
    e_test = []

    w = np.zeros(d)
    b = 0


    mu_train = 1/(1+np.exp(-1*Y_train*(b + np.matmul(X_train, w))))
    mu_test = 1/(1+np.exp(-1*Y_test*(b + np.matmul(X_test, w))))
    j_train.append(objective(mu_train, X_train, Y_train, w, lam))
    j_test.append(objective(mu_test, X_test, Y_test, w, lam))
    e_train.append(np.count_nonzero(pred(X_train, w, b) - Y_train)/n_train)
    e_test.append(np.count_nonzero(pred(X_test, w, b) - Y_test)/n_test)
    i=0
    print(f"Step␣{i}")


    while (len(j_train) < 2 or
           (np.abs(j_train[-1]-j_train[-2]) if len(j_train) > 1 else 0) > tol):
        i += 1
        print(f"Step␣{i}:␣delta␣=␣{(np.abs(j_train[-1]-j_train[-2])␣"
              f"if␣len(j_train)␣>␣1␣else␣0)}")
        gb = grad_b(mu_train, X_train, Y_train, w, lam)
        gw = grad_w(mu_train, X_train, Y_train, w, lam)

        b -= eta*gb
        w -= eta*gw

        mu_train = 1/(1+np.exp(-1*Y_train*(b + np.matmul(X_train, w))))
        mu_test = 1/(1+np.exp(-1*Y_test*(b + np.matmul(X_test, w))))

        j_train.append(objective(mu_train, X_train, Y_train, w, lam))
        j_test.append(objective(mu_test, X_test, Y_test, w, lam))
        e_train.append(np.count_nonzero(pred(X_train, w, b) - Y_train)/n_train)
        e_test.append(np.count_nonzero(pred(X_test, w, b) - Y_test)/n_test)

    return j_train,j_test,e_train,e_test




def sgdescend(X_train, Y_train, X_test, Y_test, batch, lam=0.1, eta=0.4, tol=1e-4)
    d = X_train.shape[1]
    n_train = X_train.shape[0]
    n_test = X_test.shape[0]

    j_train = []
    j_test = []
    e_train = []
    e_test = []

    w = np.zeros(d)
    b = 0



    mu_train = 1/(1+np.exp(-1*Y_train*(b + np.matmul(X_train, w))))
    mu_test = 1/(1+np.exp(-1*Y_test*(b + np.matmul(X_test, w))))

    j_train.append(objective(mu_train, X_train, Y_train, w, lam))
    j_test.append(objective(mu_test, X_test, Y_test, w, lam))
    e_train.append(np.count_nonzero(pred(X_train, w, b) - Y_train)/n_train)
    e_test.append(np.count_nonzero(pred(X_test, w, b) - Y_test)/n_test)
    i=0
    print(f”Step␣{i}”)


    while (len(j_train) < 2 or
           (np.abs(j_train[-1]-j_train[-2]) if len(j_train) > 1 else 0) > tol):
        i += 1
        print(f”Step␣{i}:␣delta␣=␣{(np.abs(j_train[-1]-j_train[-2])␣”
              f”if␣len(j_train)␣>␣1␣else␣0)}”)

        batch_ind = np.random.randint(0,n_train,batch)
        X_batch = X_train[batch_ind]
        Y_batch = Y_train[batch_ind]
        mu_batch = 1/(1+np.exp(-1*Y_batch*(b + np.matmul(X_batch, w))))

        gb = grad_b(mu_batch, X_batch, Y_batch, w, lam)
        gw = grad_w(mu_batch, X_batch, Y_batch, w, lam)
        b -= eta*gb
        w -= eta*gw

        mu_train = 1/(1+np.exp(-1*Y_train*(b + np.matmul(X_train, w))))
        mu_test = 1/(1+np.exp(-1*Y_test*(b + np.matmul(X_test, w))))

        j_train.append(objective(mu_train, X_train, Y_train, w, lam))
        j_test.append(objective(mu_test, X_test, Y_test, w, lam))

        e_train.append(np.count_nonzero(pred(X_train, w, b) - Y_train)/n_train)
        e_test.append(np.count_nonzero(pred(X_test, w, b) - Y_test)/n_test)

    return j_train,j_test,e_train,e_test




X_train, labels_train = load_train()
X_test, labels_test = load_test()

train_twosev = np.where(np.logical_or(labels_train == 2, labels_train == 7))
test_twosev =  np.where(np.logical_or(labels_test == 2, labels_test == 7))


X_train = X_train[train_twosev]
labels_train = labels_train[train_twosev]

X_test = X_test[test_twosev]
labels_test = labels_test[test_twosev]

codes = {2: -1, 7: 1}
Y_train = np.array([codes[i] for i in labels_train])
Y_test = np.array([codes[i] for i in labels_test])



if sys.argv[1] == 'gd':
    j_train,j_test,e_train,e_test = gdescend(X_train, Y_train, X_test, Y_test,
                                             eta=float(sys.argv[2]))
    now = datetime.now().time()
    stamp = f"{now.hour:02d}_{now.minute:02d}_{now.second:02d}"
    np.savez(f'data/gdescent-{stamp}', j_train=j_train, j_test=j_test,
             e_train=e_train, e_test=e_test)

elif sys.argv[1] == 'sgd':
```

```
j_train,j_test,e_train,e_test = sgdescend(X_train, Y_train, X_test,
                                          Y_test, eta=float(sys.argv[2]),
                                          batch=int(sys.argv[3]))
now = datetime.now().time()
stamp = f"{now.hour:02d}_{now.minute:02d}_{now.second:02d}"
np.savez(f'data/sgdescent-{sys.argv[3]}-{stamp}', j_train=j_train,
         j_test=j_test, e_train=e_train, e_test=e_test)
```