CSC4020 HW2 Programming

@117020119 Jiang Jingxin

Spam Classification

In [1]:
```
from scipy import io
import pandas as pd
import numpy as np
```

In [2]:
```
def read_spam_data():
    raw_data = io.loadmat('spamData.mat')
    Xtrain = pd.DataFrame(raw_data['Xtrain'])
    ytrain = pd.Series(np.hstack(raw_data['ytrain']))
    Xtest = pd.DataFrame(raw_data['Xtest'])
    ytest = pd.Series(np.hstack(raw_data['ytest']))
    return Xtrain, ytrain, Xtest, ytest
```

In [3]:
```
Xtrain, ytrain, Xtest, ytest = read_spam_data()
```

Exercise 8.1 Logistic Regression

Loss Function: $f(\mathbf{w}) = NLL(\mathbf{w}) + \lambda \mathbf{w}^T \mathbf{w} = -\sum_{i=1}^{N} [y_i \log \mu_i + (1-y_i)\log(1-\mu_i)] + \lambda \sum_{i=0}^{k} w_i^2$

Gradient: $g(\mathbf{w}) = X^T(\mu - \mathbf{y}) + 2\lambda\mathbf{w}$

In [4]:
```
class LogisticRegression:
    def __init__(self, λ, lr = 1,alpha=0.5, beta=0.5, max_iter=10000, tol = 1e-3,
                add_intercept = True, transform = None):
        self.lr = lr # learning rate
        self.alpha = alpha # step size
        self.beta = beta # step size shrinking factor
        self.max_iter = max_iter # maximal interations
        self.λ = λ # regularization term
        self.iters = 0 # number of iterations
        self.tol = tol # stopping criteria of backtracking line search
        self.add_intercept = add_intercept # add intercept to X by default
        self.transform = transform # type of transformation: stnd, log or binary

    def _transform(self, X):
        if self.transform == "stnd": # standardize
            X = (X-X.mean())/X.std()
        elif self.transform == "log": # logarithmize
            X = np.log(X+0.1)
        elif self.transform == "binary": #binarize
            X = (X>0).astype(np.int32)
        if self.add_intercept:
            X = np.hstack([np.ones((X.shape[0],1)), X])
        return X

    def _sigmoid(self, a):
        return 1 / (1 + np.exp(-a))

    def _loss(self, y, μ, w):
        return (sum(-np.log(μ[y==1]))+sum(-np.log(1-μ[y==0])) + self.λ*sum(w**2)) / y.size

    def _gradient(self, X, y, μ, w):
        return (X.T@(μ-y)) / y.size + 2*self.λ*w / y.size

    def fit(self, X, y):
        self.converged = False

        X = self._transform(X)
        n, k = X.shape

        '''
        gradient descent with backtracking line search
        '''
        prev_w = self.w = np.zeros(k)
        μ = self._sigmoid(X@self.w)
        prev_loss = self.loss = self._loss(y, μ, self.w)
        gradient = self._gradient(X, y, μ, self.w)

        for i in range(self.max_iter):
            norm_gradient = np.sqrt(sum(gradient**2))
            if (norm_gradient < self.tol):
                self.converged = True
                break

            self.iters += 1
            t = self.lr
            self.w = prev_w - t * gradient
            μ = self._sigmoid(X@self.w)
            self.loss = self._loss(y, μ, self.w)

            while (self.loss > prev_loss - self.alpha*t*norm_gradient**2):
                t = t*self.beta;
                self.w = prev_w - t*gradient
                μ = self._sigmoid(X@self.w)
                self.loss = self._loss(y, μ, self.w)

            prev_w = self.w
            prev_loss = self.loss
            gradient = self._gradient(X, y, μ, self.w)

    def predict_prob(self, X):
        X = self._transform(X)
        return self._sigmoid(X@self.w)

    def predict(self, X):
        return (self.predict_prob(X)>0.5).astype(np.int32)

    def predict_error(self,X,y):
        return sum(self.predict(X) != y)/y.size
```

In [5]:
```
from tqdm import tqdm

'''
lambda parameter tuning using cross validation
'''
def CV(X, y, lambdas = 10**np.linspace(-2,1,10), transform = None, k = 10, seed = None):
    np.random.seed(seed)
    errors = np.zeros_like(lambdas)

    idx = np.arange(y.size)
    np.random.shuffle(idx)
    fold = np.array_split(idx,k) # split shuffled index into k folds

    # cross validation using k folds
    for i in tqdm(range(k)):
        for l, lam in enumerate(lambdas):
            test_idx = fold[i]
            train_idx = np.setdiff1d(idx, test_idx)
            mod = LogisticRegression(lam,transform = transform)
            mod.fit(X.loc[train_idx],y[train_idx])
            errors[l] += mod.predict_error(X.loc[test_idx],y[test_idx])
    errors /= k
    Lam = lambdas[np.argmin(errors)]
    return Lam
```

a. Standardize

In [6]:
```
lam_stnd = CV(Xtrain,ytrain,transform="stnd")
```
```
100%|██████████| 10/10 [03:17<00:00, 19.76s/it]
```

In [7]:
```
lam_stnd
```
Out[7]:
```
1.9306977288832496
```

In [8]:
```
clfLR_stnd = LogisticRegression(λ=lam_stnd, transform = "stnd")
clfLR_stnd.fit(Xtrain,ytrain)
```

1. **training error**:

In [9]:
```
clfLR_stnd.predict_error(Xtrain,ytrain)
```
Out[9]:
```
0.07765089722675367
```

1. **test error**:

In [10]:
```
clfLR_stnd.predict_error(Xtest,ytest)
```
Out[10]:
```
0.087890625
```

b. Logarithmize

In [11]:
```
lam_log = CV(Xtrain,ytrain,transform="log")
```
```
100%|██████████| 10/10 [05:51<00:00, 35.12s/it]
```

In [12]:
```
lam_log
```
Out[12]:
```
2.1544346900318834
```

In [13]:
```
clfLR_log = LogisticRegression(λ=lam_log,transform = "log")
clfLR_log.fit(Xtrain,ytrain)
```

1. **training error**:

In [14]:
```
clfLR_log.predict_error(Xtrain,ytrain)
```
Out[14]:
```
0.052528548123980424
```

1. **test error**:

In [15]:
```
clfLR_log.predict_error(Xtest,ytest)
```
Out[15]:
```
0.057291666666666664
```

c. Binarize

In [16]:
```
lam_bin = CV(Xtrain,ytrain,transform="binary")
```
```
100%|██████████| 10/10 [00:42<00:00,  4.25s/it]
```

In [17]:
```
lam_bin
```
Out[17]:
```
0.46415888336127786
```

In [18]:
```
mod = LogisticRegression(λ=lam_bin,transform = "binary")
mod.fit(Xtrain,ytrain)
```

1. **training error**:

In [19]:
```
mod.predict_error(Xtrain,ytrain)
```
Out[19]:
```
0.06394779771615008
```

1. **test error**:

In [20]:
```
mod.predict_error(Xtest,ytest)
```
Out[20]:
```
0.07356770833333333
```

Exercise 8.2 Naive Bayes

a. NaiveBayes

$$
\begin{aligned}
h(\mathbf{x}^n) &= \arg\max_c \hat{P}(C_c)\hat{P}(\mathbf{x}^n|C_c) = \arg\max_c \left(\log \hat{P}(C_c) + \log \hat{P}(\mathbf{x}^n|C_c)\right) \\
&= \arg\max_c \left[\log \frac{N_c}{N} + \sum_{i=1}^{k} (x_{ni}\log \mu_{ic} + (1-x_{ni})\log(1-\mu_{ic}))\right]
\end{aligned}
$$

In [21]:
```
class NaiveBayes:
    def __init__(self, pseudo = 1):
        self.pseudo = pseudo

    def _binarize(self, X):
        return (X>0).astype(np.int32)

    def fit(self, X, y):
        X = self._binarize(X)
        n, k = X.shape
        C = len(np.unique(y)) # number of classes
        self.theta = np.zeros([C,k]) # mean of each feature per class
        self.prior = np.zeros(C)
        for c in range(C):
            self.theta[c] = (X[y==c].sum()+self.pseudo) / (sum(y==c)+self.pseudo*C)
            self.prior[c] = sum(y==c) / n

    def predict(self, X):
        X = self._binarize(X)
        log_prior = np.log(self.prior)
        log_post = X@np.log(self.theta.T) + (1-X)@np.log(1-self.theta.T) + log_prior
        return log_post.idxmax(axis=1)

    def predict_error(self, X, y):
        return sum(self.predict(X) != y)/y.size
```

In [22]:
```
clfNB = NaiveBayes()
clfNB.fit(Xtrain,ytrain)
```

1. **training error**:

In [23]:
```
clfNB.predict_error(Xtrain,ytrain)
```
Out[23]:
```
0.11256117455138662
```

1. **test error**:

In [24]:
```
clfNB.predict_error(Xtest,ytest)
```
Out[24]:
```
0.11002604166666667
```

b. GaussianNB

Under **MLE**: $\hat{\mu}_{ck} = \frac{1}{N_c}\sum_{i=1}^{N} x_{ik}\mathbb{1}_{y_i=c}$, $\hat{\sigma}_{ck}^2 = \frac{1}{N_c}\sum_{i=1}^{N}(x_{ik} - \hat{\mu}_{ck})^2\mathbb{1}_{y_i=c}$

In [25]:
```
from scipy.stats import multivariate_normal
```

In [26]:
```
class GaussianNB:
    def __init__(self, transform = None):
        self.transform = transform

    def _transform(self, X):
        if self.transform == "stnd":
            return (X-X.mean())/X.std()
        elif self.transform == "log":
            return np.log(X+0.1)
        return X

    def fit(self,X,y):
        X = self._transform(X)
        n, k = X.shape
        self.C = len(np.unique(y)) # number of class
        self.theta = np.zeros([self.C,k]) # mean of each feature per class
        self.sigma = np.zeros([self.C,k]) # variance of each feature per class
        self.prior = np.zeros(self.C)
        for c in range(self.C):
            self.theta[c] = X[y==c].sum() / sum(y==c)
            self.sigma[c] = X[y==c].var(ddof=0)
            self.prior[c] = sum(y==c) / n

    def predict(self,X):
        X = self._transform(X)
        log_prior = np.log(self.prior)
        log_post = np.zeros([self.C,X.shape[0]])
        for i in range(self.C):
            log_post[i] = multivariate_normal.logpdf(X, self.theta[i], np.diag(self.sigma[i]))
        log_post =  log_post.T + log_prior
        return np.argmax(log_post,axis=1)

    def predict_error(self,X,y):
        return sum(self.predict(X) != y)/y.size
```

b.i) Standardize

In [27]:
```
clfGNB_stnd = GaussianNB(transform = "stnd")
clfGNB_stnd.fit(Xtrain,ytrain)
```

1. **training error**:

In [28]:
```
clfGNB_stnd.predict_error(Xtrain,ytrain)
```
Out[28]:
```
0.17585644371941273
```

1. **test error**:

In [29]:
```
clfGNB_stnd.predict_error(Xtest,ytest)
```
Out[29]:
```
0.18880208333333334
```

b.ii) Logarithmize

In [30]:
```
clfGNB_log = GaussianNB(transform = "log")
clfGNB_log.fit(Xtrain,ytrain)
```

1. **training error**:

In [31]:
```
clfGNB_log.predict_error(Xtrain,ytrain)
```
Out[31]:
```
0.1634584013050571
```

1. **test error**:

In [32]:
```
clfGNB_log.predict_error(Xtest,ytest)
```
Out[32]:
```
0.18098958333333334
```

The End