EE 240: Pattern Recognition and Machine Learning
Homework 2

Due date: May 6, 2023

Description: These questions will explore multiple topics in logistic regression, perceptron learning, and support vector machines.

Reading assignment: ESL: Ch. 2, 3, 4, & 12, AML: Ch. 3 & 8

Homework and lab assignment submission policy:
All homework and lab assignments must be submitted online via https://eLearn.ucr.edu.
Submit your homeworks as a single Python notebook that is free of any typos or errors. Talk to your TA to make sure that the Python version match.
Homework solutions should be written and submitted individually, but discussions among students are encouraged.
All assignments should be submitted by the due date. You will incur 25% penalty for every late day.

H2.1 Logistic regression: Suppose we have labeled data $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^d$, $y \in \{0, 1\}$. We want to compute $\mathbf{w}$ that minimizes the following negative log-likelihood function:

$$\underset{\mathbf{w}}{\text{minimize}} \sum_{i=1}^{N} y_i \log(1 + \exp(-\mathbf{w}^T\mathbf{x}_i)) + (1 - y_i)\log(1 + \exp(\mathbf{w}^T\mathbf{x}_i)). \tag{1}$$

Let us denote the objective of this function as $J(\mathbf{w})$. (Note that we could also define $y_i \in \{-1, 1\}$ and rewrite the objective in (1) as $J(\mathbf{w}) = \sum_{i=1}^{N} \log(1 + \exp(-y_i\mathbf{w}^T\mathbf{x}_i))$.)

(a) Show the the following log-logistic function is convex: **(10 pts)**

$$f(\mathbf{w}) = \log(1 + \exp(-\mathbf{w}^T\mathbf{x})).$$

You may assume that $\mathbf{x}$ is a scalar and show that the second derivative of $f(\mathbf{w})$ w.r.t. $\mathbf{w}$ is always positive.

(b) Next you will write a Python script for logistic regression that solves (1) using gradient descent algorithm with a fixed step size α. You will then use your code to learn to classify images of digits from the MNIST dataset. **(40 pts)**

You need to implement the logistic regression routine using basic Python functions. Do not use any built-in function for logistic regression. You can show that the expression for $\nabla_{\mathbf{w}} J = -\sum_{i=1}^{N}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i$, where $h_{\mathbf{w}}(\mathbf{x}) = \dfrac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}}$.
MNIST data set contains 70,000 handwritten digits, each of size 28×28 pixels. You can learn more about this dataset at http://yann.lecun.com/exdb/mnist/. The first 60,000 samples are training data and the last 10,000 are test data. You can load the data and divide into training and test sets using code below.

```
from sklearn.datasets import fetch_openml

# load data
# mnist = fetch_openml('mnist_784')
X, y = fetch_openml("mnist_784", version=1, return_X_y=True, as_frame=False)
```

```
# note that images are stored as rows in the data array (70000,784)
print(X.shape)
print(y.shape)

# split data into training and testing set
train_x = X[0:60000,:]
train_y = y[0:60000]

test_x = X[60000:70000,:]
test_y = y[60000:70000]
```

Then choose the data corresponding to digits "0" and "1".

```
# Choose only two digits
class_0 = '3'
class_1 = '1'

X0 = train_x[train_y==class_0,:]
X1 = train_x[train_y==class_1,:]
y0 = np.zeros(X0.shape[0],int)
y1 = np.ones(X1.shape[0],int)

train_x = np.concatenate((X0,X1),axis=0)
train_y = np.concatenate((y0,y1),axis=0)
print('Number of training samples is {0} with {1} labels 0 and {2} labels 1'
.format(train_x.shape[0],y0.shape[0],y1.shape[0]))

X0 = test_x[test_y==class_0,:]
X1 = test_x[test_y==class_1,:]
y0 = np.zeros(X0.shape[0],int)
y1 = np.ones(X1.shape[0],int)

test_x = np.concatenate((X0,X1),axis=0)
test_y = np.concatenate((y0,y1),axis=0)
print('Number of test samples is {0} with {1} labels 0 and {2} labels 1'
.format(test_x.shape[0],y0.shape[0],y1.shape[0]))
```

After that you need to append **1** before the feature vectors.

```
# append 1 for the constant term b (remember our model is w^T x + b)
train_x = np.insert(train_x,0,1,axis = 1)
test_x = np.insert(test_x,0,1,axis = 1)
```

Then write your codes for logistic regression.

```
# Define a sigmoid function
def sigmoid(x):
    return 1/(1+np.exp(-x))
sigmoid_vec = np.vectorize(sigmoid)

# Define the gradient function
Your code goes here ...
```

```
# Write your code for logistic regression
# with gradient descent with a fixed step size
Your code goes here ...

# w is the output of logistic regression (includes weights and constant term)
# length of w for MNIST case will be 785
predict = np.round(sigmoid_vec(np.matmul(test_x.astype(float),w)))
acc =100.0*np.sum(test_y == predict)/test_y.shape[0]
print(acc)
```

H2.2 Perceptron learning algorithm (PLA) and kernel extension: In this problem you will implement PLA to build a simple binary classification system. Suppose we have labeled data $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^{d+1}$ with $\mathbf{x}_i(1) = 1$ and $y \in \{-1, +1\}$. Let us define $h_{\mathbf{w}}(\mathbf{x}_i) = \text{sign}\left(\mathbf{w}^T\mathbf{x}_i\right)$. We want to compute $\mathbf{w}$ using PLA. **(50 pts)**

(a) Data Processing: Generate two examples of 2D *linearly separable* dataset with $N = 100$ samples each. (To do this, you will first generate a weight vector and constant term, $\mathbf{w}$, and then assign ± 1 labels to your data samples as $y_i = h_{\mathbf{w}}(x_i)$.) Let us call the two datasets "Data1" and "Data2". For Data1, randomly select 80% of the samples for training and the remaining 20% for testing on Data1 (80/20). For Data2, randomly select 30% of the samples for training and the remaining 70% for testing (30/70).

(b) Implementation: Write a script for PLA in Python by initializing $\mathbf{w} = 0$ and using the following update rule:

```
for i=1,...,N
    $\mathbf{w} = \mathbf{w} + \frac{1}{2}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))\mathbf{x}_i.$
```

(c) Training: Use your PLA implementation to train two linear classifiers for Data1 and Data2 using the respective training samples. Plot the training samples, test samples, and separation lines for both examples in two separate plots.

(d) Testing: Using the test samples in each example, compute precision, recall, and f1-score for your classifiers.
Precision is defined as $\frac{tp}{tp + fp}$; tp and fp denote the number of *true* and *false* positives.
Recall is defined as $\frac{tp}{tp + fn}$; fn denotes the number of *false* negatives.
f1-score is defined as $2\,\frac{precision \times recall}{precision + recall}$; it can be viewed as a measure of accuracy and ranges between 0 (worst) and 1 (best).

(e) Kernel perceptron: Note that we can define the weight vector as $\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i$. We can view this as the dual perceptron in which we initialize $\alpha_i = 0$ for $i = 1, \ldots, N$ and update them as follows.

```
for j=1,...,N
    $\hat{y} = \text{sign}\left(\sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}_j\right)$
    if $\hat{y} \neq y_j$, update $\alpha_j = \alpha_j + 1$
```

Implement the kernel perceptron and demonstrate that the classifier you learn is identical to the one in step (b).

H2.3 Use a second degree polynomial kernel to classify four samples in 2D space, distributed in an XOR pattern (i.e., points on opposite diagonals of a rectangle have same labels, as shown in the table and figure below.) **(40 pts)**

y	$\mathbf{x}(1)$	$\mathbf{x}(2)$
1	-1	-1
-1	-1	1
-1	1	-1
1	1	1

Table 1: Table for class labels and sample values in an XOR pattern

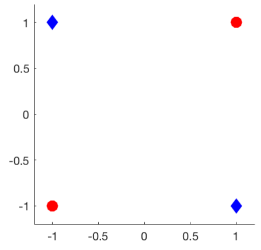

Figure 1: Blue diamonds and red circles denote samples for two classes

(a) Map 2D feature vectors $\mathbf{x} = \begin{bmatrix}\mathbf{x}(1) & \mathbf{x}(2)\end{bmatrix}^T$ using kernel function $K(\mathbf{x}_1, \mathbf{x}_2) = (1+\mathbf{x}_1^T\mathbf{x}_2)^2$, which results in 6D feature vectors: $\phi(\mathbf{x}) = \begin{bmatrix}1 & \sqrt{2}\,\mathbf{x}(1) & \sqrt{2}\,\mathbf{x}(2) & \mathbf{x}(1)^2 & \mathbf{x}(2)^2 & \sqrt{2}\,\mathbf{x}(1)\mathbf{x}(2)\end{bmatrix}^T$[1]. Show that the features vectors are linearly separable in the 5D space.

(b) The dual objective for SVM can be written as

$$L(\alpha) = -\frac{1}{2}\alpha^T G\alpha + \sum_{i=1}^{4}\alpha_i,$$

where G is a 4×4 matrix with $G(i,j) = y_i y_j \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$. Compute G matrix and α that maximize $L(\alpha)$.
Hint: You will get a simple linear equation that the optimal solution must satisfy. Compute the G matrix and solve the linear problem to get your answer.

(c) Show that α computed in the previous step satisfies the following constrains: $\alpha_i \geq 0$ and $\sum_{i=1}^{4}\alpha_i y_i = 0$.

(d) Compute the SVM weight vector $\mathbf{w} = \sum_i \alpha_i y_i \phi(\mathbf{x}_i)$ and an expression for $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x}) = \sum_i \alpha_i y_i \phi(\mathbf{x}_i)^T\phi(\mathbf{x})$. Plot $\mathbf{w}$ and discuss which features in the 6D space have highest effect on classification.

H2.4 Support vector machines (SVM): **(60 pts)**
In this problem you will use SVMs to build a simple text classification system. This data set contains 70,000 handwritten digits, each of size 28×28 pixels[2]. To begin, you need to get the MNIST dataset.

```
from sklearn.datasets import fetch_openml
# mnist = fetch_openml('mnist_784')
X, y = fetch_openml("mnist_784", version=1, return_X_y=True, as_frame=False)
```

Each row of `X` corresponds to one image. You can use the following to plot the jth image:

```
import matplotlib.pyplot as plt
plt.title('The jth image is a {label}'.format(label=int(y[j])))
plt.imshow(X[j].reshape((28,28)), cmap='gray')
plt.show()
```

[1]You can also use kernel function $K(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1^T\mathbf{x}_2)^2$, which results in 3D feature vectors: $\phi(\mathbf{x}) = \begin{bmatrix}\mathbf{x}(1)^2 & \mathbf{x}(2)^2 & \sqrt{2}\,\mathbf{x}(1)\mathbf{x}(2)\end{bmatrix}^T$.
[2]You can learn more about this dataset at http://yann.lecun.com/exdb/mnist/.

In this problem you will build a classifier to classify between the (sometimes very similar) images of the digits "4" and "9". To get just this data, you can use

```
X4 = X[y=='4',:]
X9 = X[y=='9',:]
```

There are a little under 7,000 examples from each class. You should begin by using this data to form three distinct datasets: the first 4,000 points from each class will be used for designing the classifier (this is the *training* set), and the remainder will be used to evaluate the performance of our classifier (this is the *testing* set).

Since SVMs involve tuning a parameter C, you will also need to set this parameter. We will do this in a principled way using the so-called "holdout method". This means that you take the training set and divide it into two parts: you use the first to fit the classifier, and the second—which we call the *holdout* set—to gauge performance for a given value of C. You will want to decide on a finite set of "grid points" on which to test C (I would suggest a logarithmic grid of values to test both $C \ll 1$ and $C \gg 1$). For each value of C, you will train an SVM on the first part of the set, and then compute the error rate on the holdout set. In the end, you can then choose the value of C that results in a classifier that gives the smallest error on the holdout set

Note: Once you have selected C, you may then retrain on all the entire training set (including the holdout set), but you should never use the testing set until every parameter in your algorithm is set. These are used exclusively for computing the final test error.

To train the SVM , you can use the built in solver from scikit-learn. As an example, to train a linear SVM on the full dataset with a value of $C = 1$, we would use

```
from sklearn import svm
clf= svm.SVC(C=1.0,kernel='linear')
clf.fit(X,y)
```

Once you have trained the classifier, you can calculate the probability of error for the resulting classifier on the full dataset via

```
Pe = 1 - clf.score(X,y)
```

(a) Train an SVM using the kernels $k(u, v) = (u^T v + 1)^p$, $p = 1, 2$, that is, the inhomogeneous linear and quadratic kernel. To do this, you will want to set the parameters `kernel='poly'` and `degree=1` or `degree=2`.

For each kernel, report the best value of C, the test error, and the number of data points that are support vectors (returned via `clf.support_vectors_`). Turn in your code.

(b) Repeat the above using the radial basis function kernel $k(u, v) = e^{-\gamma\|u-v\|^2}$ (by setting the parameters `kernel='rbf', gamma=gamma`. You will now need to determine the best value for both C and γ. Report the best value of C and γ, the test error, and the number of support vectors.

(c) For each kernel, turn in a 4×4 subplot showing images of the 16 support vectors that violate the margin by the greatest amount (these are, in a sense, the "hardest" examples to classify), and explain how these are determined. Above each subplot, indicate the true label (4 or 9).

To help get started with this, you can use the following code:

```
f, axarr = plt.subplots(4, 4)
axarr[0, 0].imshow(X[j].reshape((28,28)), cmap='gray')
axarr[0, 0].set_title('{label}'.format(label=int(y[j])))
...
plt.show()
```

Maximum points: 200