The breast cancer dataset from the scikit-learn library will serve as our testbed. The breast cancer dataset is a classic binary classification dataset. The dataset contains 569 samples labeled “malignant” or “benign,” denoting the state of the tumor cells. Each tumor cell is described by a 30-dimensional feature vector extracted from digitized images of the cell nuclei. Given a new cell (represented by a 30-dimensional feature vector), the goal is to predict whether the cell is malignant or benign.

1. Boosting the Perceptron

The breast cancer dataset is not linearly separable. However, a linear classifier learned, e.g., by the perceptron algorithm, is easy to implement and interpret. Here, we will investigate boosting the performance of the Perceptron by using it as a weak learner in the Adaboost algorithm.

• Implement the perceptron algorithm. You may use the scikit-learn implementation to check your work. First, load the dataset and split it into training and test datasets; the test data consists of 114 samples.

Python code below:

from sklearn.datasets import load\_breast\_cancer

from sklearn.model\_selection import train\_test\_split

data = load\_breast\_cancer()

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

(a) Since the data are not linearly separable, we do not expect the algorithm to converge. Describe how you choose your stopping criterion. (5 pts)

(b) Once you have implemented your algorithm, you may check your accuracy against a standard implementation such as sklearn’s Perceptron function. In particular, check your implementation’s accuracy against the classification accuracy (0-1 loss) of sklearn’s implementation on the same test data (from above). Explain any hyperparameter choices you make in calling sklearn’s Perceptron and whether these cause a discrepancy in the accuracy. (15 pts)

(c) [Exercise 15.2 from the textbook]: Consider the data X = [x1, · · ·, xm] ⊤, where xi ∼ D was iid. For this part alone, assume that, with probability 1, the data are linearly separable with margin ρ and that ||x||≤ R. Prove that the Perceptron converges in at most (R/ρ)^2 steps. (10 pts)

(d) Give your reasoning for how you would now choose the training dataset size for the weak learner. Recall that a weak learner should have an error < 1/2−γ, with γ > 0. Derive a bound for γ in terms of the sample size such that the probability of the weak learner making an error of 1/2 − γ is > 1 − δ, for some δ > 0. State any PAC learnability assumptions you make. (10 pts)

(e) Now, we shall implement AdaBoost with your Perceptron code from part (b) as the weak learner. Implement your AdaBoost algorithm.

i) Plot the accuracy (training and test) as a function of the number of iterations (5 pts).

ii) Plot the confidence margin of your predictions vs. the number of iterations (5 pts).

(f) Let ft be your Perceptron output at iteration t and Dt+1 be the updated distribution of the training points. Sample uniformly from Dt+1 in your code and plot the percentage of points misclassified by ft, ft−1 at each t. Analytically derive what this should be. (15 pts) [Extension of 10.3 from the book]

(g) Describe your stopping criterion for AdaBoost (2 pts).

(h) Explain your plots in part (e) using the results discussed in class about the effect of Boosting on i) the generalization error and ii) the margin. (10 pts)

The algorithm that works best (in terms of test accuracy) for our dataset is a form of Gradient-Boosted Decision Tree. Not having done this before, we are at the mercy of linear classifiers, although we know a simple linear combination of the complicated cell features may not tell us if the cell is cancerous. For this problem, logistic regression and SVMs seem to be good performers, but we need to kernel-ize these algorithms. Since we have not done kernel methods yet, we tried Boosting a linear classifier in the previous problem and ended up with a nonlinear classifier that (hopefully) does better. But, even with nonlinear decision boundaries, there might be points with low confidence, i.e., yh(x) close to 0. With our dataset especially, a low confidence prediction can be grave, as misclassification is likely. Hence, it is common in medical applications for a classifier to return a “reject” or no prediction (the value 0) for a given data. (See Ripley 1996, Herbei and Wegkamp 2006, Bartlett and Wegkamp 2008, Grandvalet et al 2008, Fumera and Roli 2002 etc).

Our classifier will return the option 0 (reject) whenever |yh(x)| ≤ ρ. Recall the true risk/error when considering the loss associated with a misclassification to be 1: P(Y h(X) < 0). First, replace the “0”s in the training labels with “-1”s to avoid confusion with the reject option (0).

(a) Let the class conditional density η(x) = P(Y = 1|X = x). Suppose the loss value of returning the reject option, 0, is c < 1/2. As with the 0-1 loss, the cost of misclassification, with confidence, yh(x) < −ρ, is 1. Derive an expression for the generalization error or Bayes risk, R(h). (5 pts)

(b) The minimizer h∗ of the generalization risk is known to form [Chow et al 1970]

h∗(x) = -1, when η(x) < δ;

0, when δ ≤ η(x) ≤ 1 – δ;

1, when η(x) > 1 − δ.

Show that δ = c minimizes the risk from part (a) with the minimum risk being EX[min{η(X), 1− η(X), c}]. (10 pts)

(c) Bartlett and Wegkamp 2008 define the following loss (2):

ℓ(z, h) = 1 − (1 − c)yh(x)/c, when yh(x) < 0;

1 − yh(x), when 0 ≤ yh(x) ≤ 1;

0 otherwise.

The above loss in (2) is greater than the discontinuous loss in part (a).

When d < 1/2 ≤ ρ ≤ 1 − d, they show that the excess risk with this loss for any h upper bounds the excess risk with the loss in part (a). Write down an optimization problem for the ERM of this loss (2) using bounded, affine functions, i.e., hw,b(x) = w⊤x + b, ||w|| ≤ r. Show that this optimization is convex. (10 pts)

(d) Derive the KKT conditions for the problem in part (c). (10 pts)

(e) Implement the ERM algorithm for the problem in part (c). You could start by modifying the loss function and the returned model in svm.py provided. Another option is to use a standard quadratic convex program solver. Here is an example code you can modify to plug in the objective function and constraints derived above.

import cvxpy as cp

import numpy as np

# Define the variables and constants

n = 3 # Number of variables

m = 2 # Number of constraints

# Define the quadratic objective function components

Q = np.array([[2.0, 1.0, 0.0],

[1.0, 2.0, 1.0],

[0.0, 1.0, 2.0]])

c = np.array([-2.0, -4.0, -6.0])

# Define the inequality constraints (Ax <= b)

A = np.array([[-1.0, 1.0, 0.0],

[1.0, 2.0, 3.0]])

b = np.array([1.0, 2.0])

# Define the decision variables

x = cp.Variable(n)

# Define the objective function

objective = cp.Minimize(0.5 \* cp.quad\_form(x, Q) + c.T @ x)

# Define the constraints

constraints = [A @ x <= b]

# Create the problem instance

problem = cp.Problem(objective, constraints)

# Solve the problem

problem.solve()

# Check if the problem is solved successfully

if problem.status == cp.OPTIMAL:

# Print the optimal value and solution

print("Optimal value =", problem.value)

print("Optimal solution x =", x.value)

else:

print("The problem did not converge to an optimal solution.")

Fine-tune the choice of ρ so that the test error (remember to modify this according to the loss (2)) is lower than your best test error in Problem 1. Take c = 1/3. Describe how you chose ρ and discuss the improvement in the test error [15 pts].