Om Sanan

Please note that this HW has been submitted late as I have been sick for the past week and a half with both the flu and COVID. If possible, I would like to request if my tardiness could be excused for this assignment only. Thank you.

CS/CNS/EE 156a Learning Systems
Caltech - Fall 2025
https://caltech.instructure.com/courses/8882
(Learning From Data campus version)

Homework # 2

Due Monday, October 13, 2025, at 2:00 PM PDT

Definitions and notation follow the lectures. All questions have multiple-choice answers ([**a**], [**b**], [**c**], **...***). Collaboration is allowed but without discussing selected or excluded choices. Your solutions must be based on your own work. See the initial "***Course Description and Policies***" handout for important details about collaboration, open book, and chatGPT policies.*

Note about the homework

- Answer each question by deriving the answer (carries 6 points) then selecting from the multiple-choice answers (carries 4 points). You can select 1 or 2 of the multiple-choice answers for each question, but you will get 4 or 2 points, respectively, for a correct answer. See the initial "**Course Description and Policies**" handout for important details.
- The problems range from easy to difficult, and from practical to theoretical. Some problems require running a full experiment to arrive at the answer.
- The answer may not be obvious or numerically close to one of the choices, but one (and only one) choice will be correct if you follow the instructions precisely in each problem. You are encouraged to explore the problem further by experimenting with variations on these instructions, for the learning benefit.
- You are encouraged to take part in the Piazza discussion forum. Please make sure you don't discuss specific answers, or specific excluded answers, before the homework is due.

note - all code is copyable at the bottom of this pdf

• Hoeffding Inequality

Run a computer simulation for flipping 1,000 virtual fair coins. Flip each coin independently 10 times. Focus on 3 coins as follows: c_1 is the first coin flipped, c_{rand} is a coin chosen randomly from the 1,000, and $c_{\min}$ is the coin which had the minimum frequency of heads (pick the earlier one in case of a tie). Let ν_1, ν_{rand}, and $\nu_{\min}$ be the *fraction* of heads obtained for the 3 respective coins out of the 10 tosses.

Run the experiment 100,000 times in order to get a full distribution of ν_1, ν_{rand}, and $\nu_{\min}$ (note that c_{rand} and $c_{\min}$ will change from run to run).

1. The average value of $\nu_{\min}$ is closest to:

[a] 0

[b] 0.01

[c] 0.1

[d] 0.5

[e] 0.67

```
import numpy as np
n_runs, n_coins, n_flips = 100000, 1000, 10
flips = np.random.randint(0, 2, (n_runs, n_coins, n_flips))
freqs = flips.mean(axis=2)
vmin = freqs.min(axis=1)
print(vmin.mean())
```

```
0.037621999999999996
```

2. Which coin(s) has a distribution of ν that satisfies the (single-bin) Hoeffding Inequality?

[a] c_1 only

[b] c_{rand} only

[c] $c_{\min}$ only

[d] c_1 and c_{rand}

[e] $c_{\min}$ and c_{rand}

c_1 and c_rand are chosen independently of their outcomes, so their frequencies nu_1 and nu_rand follow Hoeffding's bound on deviation from 0.5

c_min is selected based on its outcome (the minimum heads), introducing bias

• Error and Noise

Consider the bin model for a hypothesis h that makes an error with probability μ in approximating a deterministic target function f (both h and f are binary-valued functions). If we use the same h to approximate a noisy version of f given by:

$$P(y \mid \mathbf{x}) = \begin{cases} \lambda & y = f(\mathbf{x}) \\ 1 - \lambda & y \neq f(\mathbf{x}) \end{cases}$$

3. What is the probability of error that h makes in approximating y? *Hint: Two wrongs can make a right!*

We have P(h ≠ y) = P(h = f)P(h ≠ y or h = f) + P(h ≠ f)P(h ≠ y or h ≠ f).
If h = f (probability 1−μ) then h ≠ y when noise flips f, which occurs with probability 1−λ.
If h ≠ f (probability μ) then h ≠ y only when noise does not flip f (so y = f), which occurs with probability λ.
Therefore Pr(h ≠ y) = (1−μ)(1−λ) + μλ.

[a] μ

[b] λ

[c] $1-\mu$

[d] $(1-\lambda)*\mu+\lambda*(1-\mu)$

[e] $(1-\lambda)*(1-\mu)+\lambda*\mu$

4. At what value of λ will the performance of h be independent of μ?

[a] 0

[b] 0.5

[c] $1/\sqrt{2}$

[d] 1

[e] No values of λ

P(h ≠ y) = (1−λ)(1−mu) + λmu.
P(h ≠ y) = (1−λ) + μ(2λ−1).
For independence from mu, the coefficient of mu must be zero:
2λ − 1 = 0 ⇒ λ = 0.5.

• Linear Regression

In these problems, we will explore how Linear Regression for classification works. As with the Perceptron Learning Algorithm in Homework # 1, you will create your own target function f and data set $\mathcal{D}$. Take $d=2$ so you can visualize the problem, and assume $\mathcal{X}=[-1,1]\times[-1,1]$ with uniform probability of picking each $\mathbf{x}\in\mathcal{X}$. In each run, choose a random line in the plane as your target function f (do this by taking two random, uniformly distributed points in $[-1,1]\times[-1,1]$ and taking the line passing through them), where one side of the line maps to $+1$ and the other maps to -1. Choose the inputs $\mathbf{x}_n$ of the data set as random points (uniformly in $\mathcal{X}$), and evaluate the target function on each $\mathbf{x}_n$ to get the corresponding output y_n.

5. Take $N=100$. Use Linear Regression to find g and evaluate E_{in}, the fraction of in-sample points which got classified incorrectly. Repeat the experiment 1000 times and take the average (keep the f's and g's as they will be used again in Problem 6). Which of the following values is closest to the average E_{in}? (*Closest* is the option that makes the expression |your answer − given option| closest to 0. Use this definition of *closest* here and throughout.)

[a] 0

[b] 0.001

[c] 0.01

[d] 0.1

[e] 0.5

```python
# Helpers
## Generate target function f(x) = sign(w_f^T x)
def generate_target_function():
    p1, p2 = np.random.uniform(-1, 1, (2, 2))
    w_f = np.array([
        p2[1] - p1[1],
        p1[0] - p2[0],
        p2[0]*p1[1] - p1[0]*p2[1]
    ])
    return w_f

## Compute sign function
def sign(x): return np.where(x >= 0, 1, -1)

## Linear Regression hypothesis
def linear_regression(X, y):
    X_dagger = np.linalg.pinv(X) # computes pseudo-inverse of a matrix
    w = X_dagger @ y
    return w
```

```python
def p5(num_runs=1000, N=100):
    E_in_total = 0
    for _ in range(num_runs):
        w_f = generate_target_function()
        X = np.random.uniform(-1, 1, (N, 2))
        X = np.c_[np.ones(N), X]  # add bias term
        y = sign(X @ w_f)
        w = linear_regression(X, y)
        y_pred = sign(X @ w)
        E_in_total += np.mean(y != y_pred)
    return E_in_total / num_runs
```

Average E_in ≈ 0.026

6. Now, we go to out-of-sample error. For each run of the experiment in Problem 5, generate 1000 fresh points and use them to estimate E_{out} (fraction of misclassified points among the 1000) using the g that you got in that run. Which value is closest to the average of E_{out} over the 1000 runs of the experiment?

   [a] 0

   [b] 0.001

   [c] 0.01

   [d] 0.1

   [e] 0.5

```
def p6(num_runs=1000, N=100, N_out=1000):
    E_out_total = 0
    for _ in range(num_runs):
        w_f = generate_target_function()
        X = np.random.uniform(-1, 1, (N, 2))
        X = np.c_[np.ones(N), X]
        y = sign(X @ w_f)
        w = linear_regression(X, y)
        # test on new data
        X_out = np.random.uniform(-1, 1, (N_out, 2))
        X_out = np.c_[np.ones(N_out), X_out]
        y_out = sign(X_out @ w_f)
        y_pred_out = sign(X_out @ w)
        E_out_total += np.mean(y_out != y_pred_out)
    return E_out_total / num_runs
```

```
Average E_out ≈ 0.031
```

7. Now, take $N = 10$. After finding the weights using Linear Regression, use them as a vector of initial weights for the Perceptron Learning Algorithm. Run PLA until it converges to a final vector of weights that completely separates all the in-sample points. Among the choices below, what is the closest value to the average number of iterations (over 1000 runs) that PLA takes to converge? (When implementing PLA, have the algorithm choose a point randomly from the set of misclassified points at each iteration)

   [a] 1

   [b] 15

   [c] 300

   [d] 5000

   [e] 10000

```
# PLA using Linear Regression weights as initialization
def perceptron_learning(X, y, w_init):
    w = w_init.copy()
    iterations = 0
    while True:
        y_pred = sign(X @ w)
        misclassified = np.where(y_pred != y)[0]
        if len(misclassified) == 0:
            break
        i = np.random.choice(misclassified)
        w += y[i] * X[i]
        iterations += 1
    return iterations
```

```
def p7(num_runs=1000, N=10):
    total_iter = 0
    for t in range(num_runs):
        w_f = generate_target_function()
        X = np.random.uniform(-1, 1, (N, 2))
        X = np.c_[np.ones(N), X]
        y = sign(X @ w_f)
        w_LR = linear_regression(X, y)
        total_iter += perceptron_learning(X, y, w_LR)
    return total_iter / num_runs
```

```
Average PLA iterations ≈ 1.8
```

• Nonlinear Transformation

In these problems, we again apply Linear Regression for classification. Consider the target function:

$$f(x_1, x_2) = \text{sign}(x_1^2 + x_2^2 - 0.6)$$

Generate a training set of $N = 1000$ points on $\mathcal{X} = [-1, 1] \times [-1, 1]$ with a uniform probability of picking each $\mathbf{x} \in \mathcal{X}$. Generate simulated noise by flipping the sign of the output in a randomly selected 10% subset of the generated training set.

8. Carry out Linear Regression without transformation, i.e., with feature vector:

$$(1, x_1, x_2),$$

to find the weight $\mathbf{w}$. What is the closest value to the classification in-sample error E_{in}? (Run the experiment 1000 times and take the average E_{in} to reduce variation in your results.)

[a] 0

[b] 0.1

[c] 0.3

[d] 0.5

[e] 0.8

See below.

9. Now, transform the $N = 1000$ training data into the following nonlinear feature vector:

$$(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$$

Find the vector $\tilde{\mathbf{w}}$ that corresponds to the solution of Linear Regression. Which of the following hypotheses is closest to the one you find? Closest here means agrees the most with your hypothesis (has the highest probability of agreeing on a randomly selected point). Average the probability over 1000 runs to make sure your answer is stable.

[a] $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 1.5x_1^2 + 1.5x_2^2)$

[b] $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 1.5x_1^2 + 15x_2^2)$

[c] $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 0.13x_1x_2 + 15x_1^2 + 1.5x_2^2)$

[d] $g(x_1, x_2) = \text{sign}(-1 - 1.5x_1 + 0.08x_2 + 0.13x_1x_2 + 0.05x_1^2 + 0.05x_2^2)$

[e] $g(x_1, x_2) = \text{sign}(-1 - 0.05x_1 + 0.08x_2 + 1.5x_1x_2 + 0.15x_1^2 + 0.15x_2^2)$

See below.

10. What is the closest value to the classification out-of-sample error E_{out} of your hypothesis from Problem 9? (Estimate it by generating a new set of 1000 points and adding noise, as before. Average over 1000 runs to reduce variation in your results.)

[a] 0

[b] 0.1

[c] 0.3

[d] 0.5

[e] 0.8

See below.

All code.

```
# Problems 1-2
import numpy as np
n_runs, n_coins, n_flips = 100000, 1000, 10
flips = np.random.randint(0, 2, (n_runs, n_coins, n_flips))
freqs = flips.mean(axis=2)
vmin = freqs.min(axis=1)
print(vmin.mean())
```

See next page.

```
# Problems 5-7

# Helpers
## Generate target function f(x) = sign(w_f^T x)
def generate_target_function():
    p1, p2 = np.random.uniform(-1, 1, (2, 2))
    w_f = np.array([
        p2[1] - p1[1],
        p1[0] - p2[0],
        p2[0]*p1[1] - p1[0]*p2[1]
    ])
    return w_f

## Compute sign function
def sign(x): return np.where(x >= 0, 1, -1)

## Linear Regression hypothesis
def linear_regression(X, y):
    X_dagger = np.linalg.pinv(X) # computes pseudo-inverse
of a matrix
    w = X_dagger @ y
    return w

def p5(num_runs=1000, N=100):
    E_in_total = 0
    for _ in range(num_runs):
        w_f = generate_target_function()
        X = np.random.uniform(-1, 1, (N, 2))
        X = np.c_[np.ones(N), X]  # add bias term
        y = sign(X @ w_f)
        w = linear_regression(X, y)
        y_pred = sign(X @ w)
        E_in_total += np.mean(y != y_pred)
    return E_in_total / num_runs

def p6(num_runs=1000, N=100, N_out=1000):
    E_out_total = 0
    for _ in range(num_runs):
        w_f = generate_target_function()
        X = np.random.uniform(-1, 1, (N, 2))
        X = np.c_[np.ones(N), X]
        y = sign(X @ w_f)
        w = linear_regression(X, y)
        # test on new data
        X_out = np.random.uniform(-1, 1, (N_out, 2))
        X_out = np.c_[np.ones(N_out), X_out]
        y_out = sign(X_out @ w_f)
        y_pred_out = sign(X_out @ w)
        E_out_total += np.mean(y_out != y_pred_out)
    return E_out_total / num_runs
# code continues on next page
```

```
# PLA using Linear Regression weights as initialization
def perceptron_learning(X, y, w_init):
    w = w_init.copy()
    iterations = 0
    while True:
        y_pred = sign(X @ w)
        misclassified = np.where(y_pred != y)[0]
        if len(misclassified) == 0:
            break
        i = np.random.choice(misclassified)
        w += y[i] * X[i]
        iterations += 1
    return iterations

def p7(num_runs=1000, N=10):
    total_iter = 0
    for t in range(num_runs):
        w_f = generate_target_function()
        X = np.random.uniform(-1, 1, (N, 2))
        X = np.c_[np.ones(N), X]
        y = sign(X @ w_f)
        w_LR = linear_regression(X, y)
        total_iter += perceptron_learning(X, y, w_LR)
    return total_iter / num_runs

# Run
E_in_avg = p5()
E_out_avg = p6()
PLA_iters = p7()

print("Average E_in = ", E_in_avg)
print("Average E_out = ", E_out_avg)
print("Average PLA iterations = ", PLA_iters)
```

```
# Problems 8-10
"""
rng — np random number generator

target(X) — target label generator sign(x1^2 + x2^2 - 0.6) for rows of X

candidates — 5 candidate weight vectors (bias, x1, x2, x1*x2, x1^2, x2^2) to compare

n_runs — number of independent experiment repetitions

N — number of points per dataset (training or test per run)

M_test — number of fresh points used to test agreement with candidate hypotheses

flip_frac — fraction of labels flipped to simulate 10% label noise

ein_lin — array storing in sample classification error (linear) per run

ein_nl — array storing in sample classification error (nonlinear) per run

agree_acc — running sum of agreement fractions between learned model and each candidate

w_nl_acc — accumulator for learned nonlinear weight vectors
"""
rng = np.random.default_rng(0)

def sign(z): return np.where(z>0, 1, -1)
def target(X): return sign(X[:,0]**2 + X[:,1]**2 - 0.6)

# for p9
candidates = np.array([
   [-1.0, -0.05,  0.08, 0.13,  1.5,  1.5], # a
   [-1.0, -0.05,  0.08, 0.13,  1.5, 15.0], # b
   [-1.0, -0.05,  0.08, 0.13, 15.0,  1.5], # c
   [-1.0, -1.50,  0.08, 0.13,  0.05, 0.05], # d
   [-1.0, -0.05,  0.08, 1.50,  0.15, 0.15], # e
])

n_runs, N, M_test = 1000, 1000, 1000
flip_frac = 0.1

ein_lin = np.empty(n_runs)
ein_nl  = np.empty(n_runs)
agree_acc = np.zeros(len(candidates))
w_nl_acc = np.zeros(6)
# code continues on next page
```

```
for i in range(n_runs):
    X = rng.uniform(-1,1,(N,2)); y = target(X)
    y[rng.choice(N, int(flip_frac*N), replace=False)] *= -1

    # linear model (p8)
    Phi_lin = np.column_stack((np.ones(N), X))
    w_lin = np.linalg.pinv(Phi_lin) @ y
    ein_lin[i] = np.mean(sign(Phi_lin @ w_lin) != y)

    # nonlinear transform (p9)
    Phi_nl = np.column_stack((np.ones(N), X[:,0], X[:,1], X[:,0]*X[:,1], X[:,0]**2, X[:,1]**2))
    w_nl = np.linalg.pinv(Phi_nl) @ y
    w_nl_acc += w_nl
    ein_nl[i] = np.mean(sign(Phi_nl @ w_nl) != y)

    # agreement test vs candidates on fresh points
    X_test = rng.uniform(-1,1,(M_test,2))
    phi_test = np.column_stack((np.ones(M_test), X_test[:,0], X_test[:,1], X_test[:,0]*X_test[:,1], X_test[:,0]**2, X_test[:,1]**2))
    model_pred = sign(phi_test @ w_nl) # (M_test,)
    cand_preds  = sign(phi_test @ candidates.T) # (M_test, n_cands)
    agree_acc += np.mean(cand_preds == model_pred[:,None], axis=0)

avg_ein_lin = ein_lin.mean()
avg_w_nl = w_nl_acc / n_runs
avg_agree = agree_acc / n_runs
best_idx = int(np.argmax(avg_agree))

# p10: estimate E_out for chosen candidate and for average learned weights
eout_chosen = np.empty(n_runs); eout_learned = np.empty(n_runs)
for i in range(n_runs):
    Xo = rng.uniform(-1,1,(N,2)); yo = target(Xo)
    yo[rng.choice(N, int(flip_frac*N), replace=False)] *= -1
    phi_o = np.column_stack((np.ones(N), Xo[:,0], Xo[:,1], Xo[:,0]*Xo[:,1], Xo[:,0]**2, Xo[:,1]**2))
    eout_chosen[i] = np.mean(sign(phi_o @ candidates[best_idx]) != yo)
    eout_learned[i] = np.mean(sign(phi_o @ avg_w_nl) != yo)

print("p8 avg Ein is:", avg_ein_lin)
print("p9 avg learned nonlinear weights:", avg_w_nl)
print("p9 agreement (a..e):", avg_agree)
print("p9 best candidate:", ["a","b","c","d","e"][best_idx])
print("p10 E_out:", eout_chosen.mean())
print("p10 E_out:", eout_learned.mean())
```