Benjamin Ye
CS/CNS/EE 156a: Learning Systems (Fall 2023)
October 9, 2023

# Homework 2

| Problem | Answer |
|---|---|
| 1 | [b] |
| 2 | [d] |
| 3 | [e] |
| 4 | [b] |
| 5 | [c] |
| 6 | [c] |
| 7 | [a] |
| 8 | [d] |
| 9 | [a] |
| 10 | [b] |

### Hoeffding's 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, $c_{\text{rand}}$ is a coin chosen randomly from the 1,000, and $c_{\text{min}}$ is the coin which had the minimum frequency of heads (pick the earlier one in case of a tie). Let $\nu_1$, $\nu_{\text{rand}}$, and $\nu_{\text{min}}$ be the *fraction* of heads obtained for the 3 respective coins of the 10 tosses.

Run the experiment 100,000 times to get a full distribution of $\nu_1$, $\nu_{\text{rand}}$, and $\nu_{\text{min}}$ (note that $c_{\text{rand}}$ and $c_{\text{min}}$ will change from run to run).

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

   **Answer: [b] 0.01**

2. Which coin(s) has a distribution of $\nu$ that satisfies the (single-bin) Hoeffding's inequality?

   **Answer: [d] $c_1$ and $c_{\text{rand}}$**

   Hoeffding's inequality requires that the samples are randomly selected and not picked in a particular way such that probabilistic analysis is still valid. Only the first and the randomly selected coins satisfy this criterion. The simulation results below support this theory.

The sample output from the program used to answer problems 1–2 is

```
[HW2 P1-2]
Coin flip statistics over 100,000 trials:
  first: nu=0.50092
  random: nu=0.49990
  minimu: nu=0.03810

Hoeffding's inequality:
   eps |  bound  |      first      |      random     |     minimum
  -----+---------+-----------------+-----------------+-----------------
   0.0 | 2.00000 | 0.24807 (True)  | 0.24503 (True)  | 0.00000 (True)
   0.1 | 1.63746 | 0.40952 (True)  | 0.41167 (True)  | 0.00000 (True)
   0.2 | 0.89866 | 0.23294 (True)  | 0.23492 (True)  | 0.00000 (True)
   0.3 | 0.33060 | 0.08854 (True)  | 0.08702 (True)  | 0.00004 (True)
   0.4 | 0.08152 | 0.01924 (True)  | 0.01916 (True)  | 0.38097 (False)
   0.5 | 0.01348 | 0.00169 (True)  | 0.00220 (True)  | 0.61899 (False)
```

(The Python 3 source code is available on the following page.)

```python
import numpy as np

def coin_flip(n_trials=1, n_coins=1_000, n_flips=10, *, rng=None, seed=None):
    if rng is None:
        rng = np.random.default_rng(seed)
    heads = np.count_nonzero(
        rng.uniform(size=(n_trials, n_coins, n_flips)) < 0.5, axis=2
    )  # [0.0, 0.5) is heads, [0.5, 1.0) is tails
    return np.stack((
        heads[:, 0],
        heads[np.arange(n_trials), rng.integers(n_coins, size=n_trials)],
        heads[np.arange(n_trials), np.argmin(heads, axis=1)],
    )) / 10

def hoeffding_inequality(N, eps, M=1):
    return 2 * M * np.exp(-2 * eps**2 * N)

if __name__ == "__main__":
    n_trials = 100_000
    n_flips = 10
    labels = ("first", "random", "minimum")
    print(f"\n[HW2 P1-2]\nCoin flip statistics over {n_trials:,} trials:")
    nus = coin_flip(n_trials)
    for label, nu in zip(labels, nus.mean(axis=1)):
        print(f"  {label}: {nu=:.5f}")

    print("\nHoeffding's inequality:")
    epss = np.linspace(0, 0.5, 6)
    hist = np.apply_along_axis(
        lambda x: np.histogram(x, bins=np.linspace(-0.05, 1.05, 12))[0], 1, nus
    ) # requires at least 8 GB RAM
    probs = np.hstack((hist[:, (5,)], hist[:, 4::-1] + hist[:, 6:])) / n_trials
    bounds = hoeffding_inequality(n_flips, epss)
    print("   eps |  bound  | ", " | ".join(l.center(15) for l in labels),
          "\n  -----+---------+", "+".join(3 * [17 * "-"]), sep="")
    for eps, bound, prob, satisfy in zip(
            epss, bounds, probs.T, (probs <= bounds).T):
        print(
            f"   {eps:.1f} | {bound:.5f} |",
            " | ".join(
                f"{p:.5f} ({s})".ljust(15) for p, s in zip(prob, satisfy)
            )
        )
```

## Error and Noise

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

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

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

   **Answer: [e] $(\mathbf{1}-\boldsymbol{\lambda}) * (\mathbf{1}-\boldsymbol{\mu}) + \boldsymbol{\lambda} * \boldsymbol{\mu}$**

   There are two scenarios where $h(\mathbf{x})$ makes an error in approximating $y$:

   i. when learning is performed on the deterministic target $y = f(\mathbf{x})$ in the target distribution $P(y|\mathbf{x})$ but the hypothesis $h$ gives a false positive or negative, and
   ii. when hypothesis $h$ is correct but learning is performed on the noisy target $y \neq f(\mathbf{x})$ in the target distribution $P(y|\mathbf{x})$.

   Mathematically, the above can be written as

   $$E = P(y = f(\mathbf{x})|\mathbf{x})P\big(h(\mathbf{x}) \neq f(\mathbf{x})\big) + P(y \neq f(\mathbf{x})|\mathbf{x})P\big(h(\mathbf{x}) = f(\mathbf{x})\big) = (1-\lambda)(1-\mu) + \lambda\mu$$

4. At what value of $\lambda$ will the performance of $h$ be independent of $\mu$?

   **Answer: [b] 0.5**

   Rewriting the error expression from problem 3 by grouping by powers of $\mu$,

   $$E = 1 - \lambda + \mu(2\lambda - 1)$$

   For the performance of $h$ to be independent of $\mu$, the second term with $\mu$ must be zero:

   $$\mu(2\lambda - 1) = 0 \longrightarrow \lambda = \frac{1}{2}$$

## 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 $E_{\text{in}}$, the fraction of in-sample points which got classified incorrectly. Repeat the experiment 1,000 times and take the average (keep the $g$'s as they will be used again in problem 6). Which of the following values is closest to the average $E_{\text{in}}$? (*Closest* is the option that makes the expression |your answer − given option| closest to 0. Use this definition of *closest* and here and throughout.)

   **Answer: [c] 0.01**

6. Now, generate 1,000 fresh points and use them to estimate the out-of-sample error $E_{\text{out}}$ of $g$ that you got in problem 5 (number of misclassified out-of-sample points / total number of out-of-sample points). Again, run the experiment 1,000 times and take the average. Which value is closest to the average $E_{\text{out}}$?

   **Answer: [c] 0.01**

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 (PLA). 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 1,000 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.)

   **Answer: [a] 1**

For the linear regression and perceptron, the target function is a simple line $f(\mathbf{x}) = mx + b$, where $x$ is the $x$-coordinate of point $\mathbf{x}_n$ in data set $\mathcal{D}$. The sample output from the program used to answer problems 5–7 is

```
[HW2 P5-7]
Linear regression statistics over 1,000 runs:
  N=100, E_in=0.037, E_out=0.047

PLA (with linear regression hypothesis) statistics over 1,000 runs:
  N=10, iters=4
```

(The Python 3 source code is available on the following pages.)

```python
import copy

import numpy as np

def target_function_random_line(*, rng=None, seed=None):
    if rng is None:
        rng = np.random.default_rng(seed)
    line = rng.uniform(-1, 1, (2, 2))
    return lambda x: np.sign(
        x[:, 2] - line[0, 1]
        - np.divide(*(line[1] - line[0])[::-1]) * (x[:, 1] - line[0, 0])
    )

def generate_data(N, f, d=2, lb=-1.0, ub=1.0, *, rng=None, seed=None):
    if rng is None:
        rng = np.random.default_rng(seed)
    x = np.hstack((np.ones((N, 1)), rng.uniform(lb, ub, (N, d))))
    return x, f(x)

def validate_binary(w, x, y):
    return np.count_nonzero(np.sign(x @ w) != y, axis=0) / x.shape[0]

def linear_regression(
        N, f, *, N_test=1_000, transform=None, noise=None, rng=None, seed=None,
        hyp=False):
    if rng is None:
        rng = np.random.default_rng(seed)
    x, y = generate_data(N, f, rng=rng)
    if transform:
        x = transform(x)
    if noise:
        i = rng.choice(N, round(noise[0] * N), False)
        y[i] = noise[1](y[i])
    w = np.linalg.pinv(x) @ y

    x_test, y_test = generate_data(N_test, f, rng=rng)
    if transform:
        x_test = transform(x_test)
    if noise:
        i = rng.choice(N_test, round(noise[0] * N_test), False)
        y_test[i] = noise[1](y_test[i])
    return (w, validate_binary(w, x, y),
            validate_binary(w, x_test, y_test))[1 - hyp:]
```

```python
def perceptron(N, f, *, w=None, N_test=1_000, rng=None, seed=None):
    if rng is None:
        rng = np.random.default_rng(seed)
    x, y = generate_data(N, f, rng=rng)
    if w is None:
        w = np.zeros(x.shape[1], dtype=float)
    iters = 0
    while True:
        wrong = np.argwhere(np.sign(x @ w) != y)[:, 0]
        if wrong.size == 0:
            break
        i = np.random.choice(wrong)
        w += y[i] * x[i]
        iters += 1
    return iters, validate_binary(w, *generate_data(N_test, f, rng=rng))

if __name__ == "__main__":
    rng = np.random.default_rng()
    N = 100
    n_runs = 1_000
    print(f"\n[HW2 P5-7]\nLinear regression statistics over {n_runs:,} runs:")
    E_in, E_out = np.mean(
        [linear_regression(N, target_function_random_line(rng=rng), rng=rng)
         for _ in range(n_runs)],
        axis=0
    )
    print(f"  {N=:,}, {E_in=:.3f}, {E_out=:.3f}")

    N = 10
    print("\nPLA (with linear regression hypothesis) statistics over",
          f"{n_runs:,} runs:")
    iters = np.empty(n_runs, dtype=float)
    for i in range(n_runs):
        f = target_function_random_line(rng=rng)
        iters[i] = perceptron(
            N, f,
            w=linear_regression(N, f, rng=copy.copy(rng), hyp=True)[0],
            rng=copy.copy(rng) # ensures same RNG state for PLA and LRA
        )[0]
    print(f"  {N=:,}, iters={iters.mean():,.0f}")
```

**Nonlinear Transformation**

In these problems, we again apply linear regression for classification. Consider the target function:

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

Generate a training set of $N = 1{,}000$ 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 $E_{\text{in}}$? (Run the experiment 1,000 times and take the average $E_{\text{in}}$ to reduce variation in your results.)

**Answer: [d] 0.5**

9. Now, transform the $N = 1{,}000$ 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 over a few runs to make sure your answer is stable.

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

10. What is the closest value to the classification out-of-sample error $E_{\text{out}}$ of your hypothesis from Problem 9? (Estimate it by generating a new set of 1,000 points and adding noise, as before. Average over 1,000 runs to reduce the variation in your results.)

**Answer: [b] 0.1**

The sample output from the program used to answer problems 8–10 is

```
[HW2 P8-10]
Linear regression (with linear feature vector) statistics over 1,000 runs:
  N=1,000, noise=0.100, E_in=0.504

Linear regression (with nonlinear feature vector) hypothesis over 1,000 runs:
  w=[-0.99351, -0.00074, -0.00222, 0.00110, 1.56024, 1.55690]
  g1=[-1, -0.05, 0.08, 0.13, 1.5, 1.5] (prob=0.97117)
  g2=[-1, -0.05, 0.08, 0.13, 1.5, 15] (prob=0.66324)
  g3=[-1, -0.05, 0.08, 0.13, 15, 1.5] (prob=0.66237)
  g4=[-1, -1.5, 0.08, 0.13, 0.05, 0.05] (prob=0.63323)
  g5=[-1, -0.05, 0.08, 1.5, 0.15, 0.15] (prob=0.56053)
  N=1,000, noise=0.100, E_out=0.123
```

(The Python 3 source code is available on the following pages.)

```python
import numpy as np

def target_function_hw2():
    return lambda x: np.sign((x[:, 1:]**2).sum(axis=1) - 0.6)

def generate_data(N, f, d=2, lb=-1.0, ub=1.0, *, rng=None, seed=None):
    if rng is None:
        rng = np.random.default_rng(seed)
    x = np.hstack((np.ones((N, 1)), rng.uniform(lb, ub, (N, d))))
    return x, f(x)

def validate_binary(w, x, y):
    return np.count_nonzero(np.sign(x @ w) != y, axis=0) / x.shape[0]

def linear_regression(
        N, f, *, N_test=1_000, transform=None, noise=None, rng=None, seed=None,
        hyp=False):
    if rng is None:
        rng = np.random.default_rng(seed)
    x, y = generate_data(N, f, rng=rng)
    if transform:
        x = transform(x)
    if noise:
        i = rng.choice(N, round(noise[0] * N), False)
        y[i] = noise[1](y[i])
    w = np.linalg.pinv(x) @ y

    x_test, y_test = generate_data(N_test, f, rng=rng)
    if transform:
        x_test = transform(x_test)
    if noise:
        i = rng.choice(N_test, round(noise[0] * N_test), False)
        y_test[i] = noise[1](y_test[i])
    return (w, validate_binary(w, x, y),
            validate_binary(w, x_test, y_test))[1 - hyp:]
```

```python
if __name__ == "__main__":
    rng = np.random.default_rng()
    f = target_function_hw2()
    N = N_test = n_runs = 1_000
    noise = (0.1, lambda y: -y)
    print("\n[HW2 P8-10]\nLinear regression (with linear feature vector)",
          f"statistics over {n_runs:,} runs:")
    E_in = np.mean(
        [linear_regression(N, f, noise=noise)[0] for _ in range(n_runs)]
    )
    print(f"  {N=:,}, noise={noise[0]:.3f}, {E_in=:.3f}")

    transform = lambda x: np.hstack((x, x[:, 1:2] * x[:, 2:], x[:, 1:2]**2,
                                     x[:, 2:]**2))
    gs = np.array(((-1, -0.05, 0.08, 0.13, 1.5, 1.5),
                   (-1, -0.05, 0.08, 0.13, 1.5, 15),
                   (-1, -0.05, 0.08, 0.13, 15, 1.5),
                   (-1, -1.5, 0.08, 0.13, 0.05, 0.05),
                   (-1, -0.05, 0.08, 1.5, 0.15, 0.15)))
    print("\nLinear regression (with nonlinear feature vector) hypothesis",
          f"over {n_runs:,} runs:")
    w = np.mean(
        [linear_regression(N, f, transform=transform, noise=noise, rng=rng,
                           hyp=True)[0]
         for _ in range(n_runs)],
        axis=0
    )
    print("  w=[", ", ".join(f"{v:.5f}" for v in w), "]", sep="")
    probs = np.zeros((N_test, 5))
    Es_out = np.zeros(N_test)
    for i in range(n_runs):
        x_test, y_test = generate_data(N_test, f, rng=rng)
        x_test = transform(x_test)
        y_test[rng.choice(N_test, round(noise[0] * N_test), False)] *= -1
        h_test = np.sign(x_test @ w)
        probs[i] = validate_binary(gs.T, x_test, h_test[:, None])
        Es_out[i] = np.count_nonzero(h_test != y_test) / N_test
    for i, (g, p) in enumerate(zip(gs, probs.mean(axis=0))):
        print(f"  g{i + 1}=[", ", ".join(f"{v:.2g}" for v in g),
              f"] (prob={1 - p:.5f})", sep="")
    print(f"  {N=:,}, noise={noise[0]:.3f}, E_out={Es_out.mean():.3f}")
```