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_{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 ν_{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 ν_1, ν_{rand}, and ν_{min} (note that c_{rand} and c_{min} will change from run to run).

1. The average value of ν_{min} is closest to:

   Answer: [b] 0.01

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

   Answer: [d] c_1 and c_{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.

Sample program output

```
[Homework 2 Problem 1]
100,000 trials, 1,000 coins, 10 flips:
                   coin  fraction of heads
              first coin           0.500833
             random coin           0.500068
min. frequency of heads           0.037435

[Homework 2 Problem 2]
Hoeffding inequality:
 epsilon    bound  first coin       random coin       min. frequency of heads
     0.0 2.000000     0.24841 True      0.24746 True                  0.00000  True
     0.1 1.637462     0.40758 True      0.40982 True                  0.00000  True
     0.2 0.898658     0.23573 True      0.23482 True                  0.00000  True
     0.3 0.330598     0.08672 True      0.08731 True                  0.00003  True
     0.4 0.081524     0.01965 True      0.01832 True                  0.37844 False
     0.5 0.013476     0.00191 True      0.00227 True                  0.62153 False
```

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

Python 3 source code

```
import numpy as np
import pandas as pd

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
    indices = np.arange(N_trials)
    return np.stack((
        heads[:, 0],
        heads[indices, rng.integers(N_coins, size=N_trials)],
        heads[indices, np.argmin(heads, axis=1)],
    )) / N_flips

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

if __name__ == "__main__":
    rng = np.random.default_rng()

    N_trials = 100_000
    N_coins = 1_000
    N_flips = 10
    nus = coin_flip(N_trials, N_coins, N_flips, rng=rng)
    coins = ("first coin", "random coin", "min. frequency of heads")
    df = pd.DataFrame({"coin": coins, "fraction of heads": nus.mean(axis=1)})
    print(f"\n[Homework 2 Problem 1]\n"
          f"{N_trials:,} trials, {N_coins:,} coins, {N_flips:,} flips:\n",
          df.to_string(index=False), sep="")

    epsilons = np.linspace(0, 0.5, 6)
    histograms = 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
    probabilities = np.hstack((
        histograms[:, (5,)],
        histograms[:, 4::-1] + histograms[:, 6:]
    )) / N_trials
    bounds = hoeffding_inequality(N_flips, epsilons)
    satisfies = probabilities < bounds
    data = {"epsilon": epsilons, "bound": bounds}
    for i in range(nus.shape[0]):
        data[coins[i]] = probabilities[i]
        data[i * " "] = satisfies[i]
    print("\n[Homework 2 Problem 2]\n"
          "Hoeffding inequality:\n",
          pd.DataFrame(data).to_string(index=False), sep="")
```

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 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 λ will the performance of h be independent of μ?

   Answer: [b] 0.5

   Rewriting the error expression from Problem 3 by grouping by powers of μ,

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

   For the performance of h to be independent of μ, the second term with μ 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_{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_{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_{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_{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 $\mathcal{D}$.

Sample program output

```
[Homework 2 Problems 5–6]
For the linear regression model, the average in-sample and out-of-sample errors
over 1,000 runs are 0.028130 and 0.039050, respectively.

[Homework 2 Problem 7]
With initial weights from linear regression, the perceptron takes an average of 4
iterations to converge.
```

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

Python 3 source code

```
import numpy as np

class Perceptron:
    def __init__(self, w=None, *, vf=None):
        self.set_parameters(w, vf=vf)

    def get_error(self, x, y):
        if self.vf is not None and self.w is not None:
            return self.vf(self.w, x, y)

    def set_parameters(self, w=None, *, vf=None, update=False) -> None:
        if update:
            self.vf = vf or self.vf
            self._w = self._w if w is None else w
        else:
            self.vf = vf
            self._w = w

    def train(self, x, y):
        self.iters = 0
        self.w = (np.zeros(x.shape[1], dtype=float) if self._w is None
                  else self._w)
        while True:
            wrong = np.argwhere(np.sign(x @ self.w) != y)[:, 0]
            if wrong.size == 0:
                break
            index = np.random.choice(wrong)
            self.w += y[index] * x[index]
            self.iters += 1
        if self.vf:
            return self.vf(self.w, x, y)

class LinearRegression:
    def __init__(
            self, *, vf=None, regularization=None, transform=None, noise=None,
            rng=None, seed=None, **kwargs):
        self.rng = np.random.default_rng(seed) if rng is None else rng
        self.set_parameters(vf=vf, regularization=regularization,
                            transform=transform, noise=noise, **kwargs)

    def get_error(self, x, y):
        if self.transform:
            x = self.transform(x)
        if self.noise:
            N = x.shape[0]
            index = self.rng.choice(N, round(self.noise[0] * N), False)
            y[index] = self.noise[1](y[index])

        if self.vf is not None and self.w is not None:
            return self.vf(self.w, x, y)
```

```
    def set_parameters(
            self, *, vf=None, regularization=None, transform=None, noise=None,
            update=False, **kwargs):
        self._reg_params = {}
        if update:
            self.noise = noise or self.noise
            self.regularization = regularization or self.regularization
            if self.regularization == "weight_decay" \
                    and "weight_decay_lambda" in kwargs:
                self._reg_params["lambda"] = kwargs["weight_decay_lambda"]
            self.transform = transform or self.transform
            self.vf = vf or self.vf
        else:
            self.noise = noise
            self.regularization = regularization
            if regularization == "weight_decay":
                self._reg_params["lambda"] = kwargs["weight_decay_lambda"]
            self.transform = transform
            self.vf = vf

    def train(self, x, y):
        if self.transform:
            x = self.transform(x)
        if self.noise:
            N = x.shape[0]
            index = self.rng.choice(N, round(self.noise[0] * N), False)
            y[index] = self.noise[1](y[index])
        if self.regularization is None:
            self.w = np.linalg.pinv(x) @ y
        elif self.regularization == "weight_decay":
            self.w = np.linalg.inv(
                x.T @ x
                + self._reg_params["lambda"] * np.eye(x.shape[1], dtype=float)
            ) @ x.T @ y
        if self.vf is not None:
            return self.vf(self.w, x, y)

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

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

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

if __name__ == "__main__":
    rng = np.random.default_rng()

    N_train = 100
    N_test = 9 * N_train
    N_runs = 1_000
    f = target_function_random_line(rng=rng)
    reg = LinearRegression(vf=validate_binary, rng=rng)
    errors = np.zeros(2, dtype=float)
    for _ in range(N_runs):
        E_in = reg.train(*generate_data(N_train, f, bias=True, rng=rng))
        errors += (
            E_in,
            reg.get_error(*generate_data(N_test, f, bias=True, rng=rng))
        )
    errors /= N_runs
    print("\n[Homework 2 Problems 5–6]\n"
          "For the linear regression model, the average in-sample and "
          f"out-of-sample errors over {N_runs:,} runs are "
          f"{errors[0]:.6f} and {errors[1]:.6f}, respectively.")

    N_train = 10
    pla = Perceptron(vf=validate_binary)
    iters = 0
    for _ in range(N_runs):
        f = target_function_random_line(rng=rng)
        x_train, y_train = generate_data(N_train, f, bias=True, rng=rng)
        reg.train(x_train, y_train)
        pla.set_parameters(w=reg.w, update=True)
        pla.train(x_train, y_train)
        iters += pla.iters
    print("\n[Homework 2 Problem 7]\n"
          "With initial weights from linear regression, the perceptron "
          f"takes an average of {iters / N_runs:.0f} iterations to "
          "converge.")
```

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_{in}? (Run the experiment 1,000 times and take the average E_{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_{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 and Python 3 source code are on the following pages. ⟶)

Sample program output

```
[Homework 2 Problem 8]
For the linear regression model with 10% noise, the average in-sample error over
1,000 runs is 0.505238.

[Homework 2 Problem 9]
The average weight vector over 1,000 runs is
w=[-0.991809, -0.001878, 0.001049, -0.001620, 1.556251, 1.557031].

Closest hypothesis:
choice                                  g  probability
   [a]  [-1, -0.05, 0.08, 0.13, 1.5, 1.5]     0.971637
   [b]   [-1, -0.05, 0.08, 0.13, 1.5, 15]     0.663314
   [c]   [-1, -0.05, 0.08, 0.13, 15, 1.5]     0.662705
   [d] [-1, -1.5, 0.08, 0.13, 0.05, 0.05]     0.632994
   [e] [-1, -0.05, 0.08, 1.5, 0.15, 0.15]     0.561165

[Homework 2 Problem 10]
The average out-of-sample error over 1,000 runs is 0.123321.
```

Python 3 source code

```
import numpy as np
import pandas as pd

class LinearRegression:
    def __init__(
            self, *, vf=None, regularization=None, transform=None, noise=None,
            rng=None, seed=None, **kwargs):
        self.rng = np.random.default_rng(seed) if rng is None else rng
        self.set_parameters(vf=vf, regularization=regularization,
                            transform=transform, noise=noise, **kwargs)

    def get_error(self, x, y):
        if self.transform:
            x = self.transform(x)
        if self.noise:
            N = x.shape[0]
            index = self.rng.choice(N, round(self.noise[0] * N), False)
            y[index] = self.noise[1](y[index])

        if self.vf is not None and self.w is not None:
            return self.vf(self.w, x, y)
```

```
    def set_parameters(
            self, *, vf=None, regularization=None, transform=None, noise=None,
            update=False, **kwargs):
        self._reg_params = {}
        if update:
            self.noise = noise or self.noise
            self.regularization = regularization or self.regularization
            if self.regularization == "weight_decay" \
                    and "weight_decay_lambda" in kwargs:
                self._reg_params["lambda"] = kwargs["weight_decay_lambda"]
            self.transform = transform or self.transform
            self.vf = vf or self.vf
        else:
            self.noise = noise
            self.regularization = regularization
            if regularization == "weight_decay":
                self._reg_params["lambda"] = kwargs["weight_decay_lambda"]
            self.transform = transform
            self.vf = vf

    def train(self, x, y):
        if self.transform:
            x = self.transform(x)
        if self.noise:
            N = x.shape[0]
            index = self.rng.choice(N, round(self.noise[0] * N), False)
            y[index] = self.noise[1](y[index])
        if self.regularization is None:
            self.w = np.linalg.pinv(x) @ y
        elif self.regularization == "weight_decay":
            self.w = np.linalg.inv(
                x.T @ x
                + self._reg_params["lambda"] * np.eye(x.shape[1], dtype=float)
            ) @ x.T @ y
        if self.vf is not None:
            return self.vf(self.w, x, y)

def target_function_homework_2(x):
    f = lambda x: np.sign((x[:, -2:] ** 2).sum(axis=1) - 0.6)
    return f if x is None else f(x)

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

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

```
if __name__ == "__main__":
    rng = np.random.default_rng()

    N_train = N_runs = 1_000
    N_test = 9 * N_train
    noise = (0.1, lambda y: -y)
    reg = LinearRegression(vf=validate_binary, noise=noise, rng=rng)
    E_in = 0
    for _ in range(N_runs):
        E_in += reg.train(*generate_data(N_train, target_function_homework_2,
                                         bias=True, rng=rng))
    print("\n[Homework 2 Problem 8]\n"
          f"For the linear regression model with {noise[0]:.0%} noise, "
          f"the average in-sample error over {N_runs:,} runs is "
          f"{E_in / N_runs:.6f}.")

    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)))
    w = np.zeros_like(gs[0])
    reg.set_parameters(vf=validate_binary, transform=transform, noise=noise,
                       update=True)
    for _ in range(N_runs):
        reg.train(*generate_data(N_train, target_function_homework_2,
                                 bias=True, rng=rng))
        w += reg.w
    w /= N_runs
    counters = np.zeros(6, dtype=float)
    for _ in range(N_runs):
        x_test, y_test = generate_data(N_test, target_function_homework_2,
                                       bias=True, 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)
        counters += (*validate_binary(gs.T, x_test, h_test[:, None]),
                     np.count_nonzero(h_test != y_test) / N_test)
    counters /= N_runs
    df = pd.DataFrame({
        "choice": [f"[{chr(97 + i)}]" for i in range(5)],
        "g": [f"[{', '.join(f'{c:.2g}' for c in g)}]" for g in gs],
        "probability": 1 - counters[:5]
    })
    print("\n[Homework 2 Problem 9]\n"
          f"The average weight vector over {N_runs:,} runs is "
          "w=[", ", ".join(f"{v:.6f}" for v in w), "].\n\n"
          "Closest hypothesis:\n",
          df.to_string(index=False), sep="")
    print("\n[Homework 2 Problem 10]\n"
          f"The average out-of-sample error over {N_runs:,} runs is "
          f"{counters[5]:.6f}.")
```