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: is the first coin flipped, is a coin chosen randomly from the 1,000, and is the coin which had the minimum frequency of heads (pick the earlier one in case of a tie). Let , , and 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 , , and (note that and will change from run to run).

1. The average value of is closest to:

**Answer: [b] 0.01**

1. Which coin(s) has a distribution of that satisfies the (single-bin) Hoeffding’s inequality?

**Answer: [d] and**

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 that makes an error with probability in approximating a deterministic target function (both and are binary functions). If we use the same to approximate a noisy version of given by:

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

**Answer: [e]**

There are two scenarios where makes an error in approximating :

1. when learning is performed on the deterministic target in the target distribution but the hypothesis gives a false positive or negative, and
2. when hypothesis is correct but learning is performed on the noisy target in the target distribution .

Mathematically, the above can be written as

1. At what value of will the performance of be independent of ?

**Answer: [b] 0.5**

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

For the performance of to be independent of , the second term with must be zero:

**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 and data set . Take so you can visualize the problem and assume with uniform probability of picking each . In each run, choose a random line in the plane as your target function (do this by taking two random, uniformly distributed points in and taking the line passing through them), where one side of the line maps to and the other maps to . Choose the inputs of the data set as random points (uniformly in ) and evaluate the target function on each to get the corresponding output .

1. Take . Use linear regression to find and evaluate , the fraction of in-sample points which got classified incorrectly. Repeat the experiment 1,000 times and take the average (keep the ’s as they will be used again in Problem 6). Which of the following values is closest to the average ? (*Closest* is the option that makes the expression closest to 0. Use this definition of *closest* and here and throughout.)

**Answer: [c] 0.01**

1. Now, generate 1,000 fresh points and use them to estimate the out-of-sample error of 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 ?

**Answer: [c] 0.01**

1. Now, take . 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 , where is the -coordinate of point in data set .

**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 = {}

        self.w = None

        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:

Generate a training set of points on with a uniform probability of picking each . Generate simulated noise by flipping the sign of the output in a randomly selected 10% subset of the generated training set.

1. Carry out linear regression without transformation, i.e., with feature vector

to find the weight . What is the closest value to the classification in-sample error ? (Run the experiment 1,000 times and take the average to reduce variation in your results.)

**Answer: [d] 0.5**

1. Now, transform the training data into the following nonlinear feature vector:

Find the vector 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]**

1. What is the closest value to the classification out-of-sample error 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 = {}

        self.w = None

        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}.")