Benjamin Ye
CS/CNS/EE 156a: Learning Systems (Fall 2023)
October 30, 2023

Homework 5

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

Linear Regression Error

Consider a noisy target $y = \mathbf{w}^{*\mathsf{T}}\mathbf{x} + \epsilon$, where $\mathbf{x} \in \mathbb{R}^d$ (with the added coordinate $x_0 = 1$), $y \in \mathbb{R}$, $\mathbf{w}^*$ is an unknown vector, and ϵ is a noise term with zero mean and σ^2 variance. Assume ϵ is independent of $\mathbf{x}$ and of all other ϵ's. If linear regression is carried out using a training data set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, and outputs the parameter vector $\mathbf{w}_{\text{lin}}$, it can be shown that the expected in-sample error E_{in} with respect to $\mathcal{D}$ is given by

$$\mathbb{E}_{\mathcal{D}}[E_{\text{in}}(\mathbf{w}_{\text{lin}})] = \sigma^2\left(1 - \frac{d+1}{N}\right)$$

1. For $\sigma = 0.1$ and $d = 8$, which among the following choices is the smallest number of examples N that will result in an expected E_{in} greater than 0.008?

   Answer: [c] 100

   Solving for N by turning the equation above into an inequality,

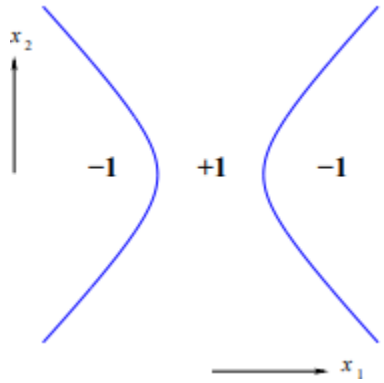
$$N \geq \frac{d+1}{1 - \mathbb{E}_{\mathcal{D}}[E_{\text{in}}(\mathbf{w}_{\text{lin}})]/\sigma^2} = \frac{8+1}{1 - 0.008/0.1^2} = 45$$

Nonlinear Transforms

In linear classification, consider the feature transform $\Phi: \mathbb{R}^2 \to \mathbb{R}^2$ (plus the added zeroth coordinate) given by

$$\Phi(1, x_1, x_2) = (1, x_1^2, x_2^2)$$

2. Which of the following sets of constraints on the weights in the $\mathcal{Z}$ space could correspond to the hyperbolic decision boundary in $\mathcal{X}$ depicted in the following figure?



You may assume that $\tilde{w}_0$ can be selected to achieve the desired boundary.

Answer: [d] $\tilde{w}_1 < 0$, $\tilde{w}_2 > 0$

When $\tilde{\mathbf{w}}$ has been determined, $\text{sgn}(\tilde{\mathbf{w}}^\top \mathbf{x}) = \text{sgn}(\tilde{w}_0 + \tilde{w}_1 x_1^2 + \tilde{w}_2 x_2^2)$ should give the correct classification for $\Phi(1, x_1, x_2)$ in $\mathcal{X}$.

Let us take the center of the figure above to be an arbitrary reference point, i.e., the hyperbola has its center at $(0, 0)$, since $\mathcal{X}$ can always be translated such that this is the case. As x_1^2 increases (moving away horizontally from the origin), the function tends to -1 so $\tilde{w}_1$ should be negative. As x_2^2 increases (moving away vertically from the origin), the function tends to $+1$, so $\tilde{w}_2$ should be positive.

Analytically, this is equivalent to considering the general form for a hyperbola:

$$\frac{x_1^2}{a^2} - \frac{x_2^2}{b^2} = c^2$$

where a, b, and c are arbitrary non-zero constants.

By moving all terms to the right-hand side such that the left-hand side of the equation above is 0 (to determine where the hyperbolic decision boundary is) and taking the sign of the right-hand sign, i.e., $\text{sgn}(c^2 - x_1^2/a^2 + x_2^2/b^2)$, we obtain a formulation equivalent in form to $\text{sgn}(\tilde{w}_0 + \tilde{w}_1 x_1^2 + \tilde{w}_2 x_2^2)$ for $\mathcal{X}$. We can see that $\tilde{w}_0 = c^2$ is positive, $\tilde{w}_1 = -a^{-2}$ is negative, and $\tilde{w}_2 = b^{-2}$ is positive.

Now, consider the 4th order polynomial transform from the input space $\mathbb{R}^2$:

$$\Phi_4 : \mathbf{x} \to (1, x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3, x_1^4, x_1^3x_2, x_1^2x_2^2, x_1x_2^3, x_2^4)$$

3. What is the smallest value among the following choices that is *not* smaller than the VC dimension of a linear model in this transformed space?

   Answer: [c] 15

   For Φ_4, $\mathcal{Z}$ has $\tilde{d} = 14$ dimensions (not counting the zeroth coordinate), so the VC dimension of $\mathcal{Z}$ is $d_{\text{VC}} = \tilde{d} + 1 = 15$.

Gradient Descent

Consider the nonlinear error surface $E(u, v) = (ue^v - 2ve^{-u})^2$. We start at the point $(u, v) = (1, 1)$ and minimize this error using gradient descent in the uv space. Use $\eta = 0.1$ (learning rate, not step size).

4. What is the partial derivative of $E(u, v)$ with respect to u, i.e., $\partial E/\partial u$?

   Answer: [e] $2(e^v + 2ve^{-u})(ue^v - 2ve^{-u})$

   By applying the chain rule, the partial derivative is

$$\frac{\partial E}{\partial u} = 2(ue^v - 2ve^{-u})\frac{\partial}{\partial u}(ue^v - 2ve^{-u}) = 2(ue^v - 2ve^{-u})(e^v + 2ve^{-u})$$

5. How many iterations (among the given choices) does it take for the error $E(u, v)$ to fall below 10^{-14} for the first time? In your programs, make sure to use double precision to get the needed accuracy.

   Answer: [d] 10

6. After running enough iterations such that the error has just dropped below 10^{-14}, what are the closest values (in Euclidean distance) among the following choices to the final (u, v) you got in Problem 5?

   Answer: [e] $[0.045, 0.024]$

7. Now, we will compare the performance of "coordinate descent". In each iteration, we have two steps along the two coordinates. Step 1 is to move only along the u coordinate to reduce the error (assume first-order approximation holds like in gradient descent), and step 2 is the reevaluate and move along only the v coordinate to reduce the error (again, assume first-order approximation holds). Use the same learning rate of $\eta = 0.1$ as we did in gradient descent. What will the error $E(u, v)$ be closest to after 15 full iterations (30 steps)?

   Answer: [a] 10^{-1}

See the following page for derivations and implementation details for Problems 5–7.

The other missing piece required for the gradient descent method is the partial derivative of E with respect to v, which is

$$\frac{\partial E}{\partial v} = 2(ue^v - 2ve^{-u})(ue^v - 2e^{-u})$$

The sample output from the program used to answer Problems 5–7 is

```
[HW5 P5-7]
Performance of descent methods for eta=0.1:
  Gradient descent: iters=10, x=(0.045, 0.024)
  Coordinate descent: iters=15, x=(6.297, -2.852), E(x)=1.398e-01
```

The Python 3 source code is below.

```
import numpy as np

def gradient_descent(E, dE, x, *, eta=0.1, tol=1e-14, max_iters=100_000):
    iters = 0
    while E(x) > tol and iters < max_iters:
        x -= eta * dE(x)
        iters += 1
    return x, iters

def coordinate_descent(
        E, dE_dx, x, *, eta=0.1, tol=1e-14, max_iters=100_000):
    iters = 0
    while E(x) > tol and iters < max_iters:
        for i in range(len(x)):
            x[i] -= eta * dE_dx[i](x)
        iters += 1
    return x, iters

if __name__ == "__main__":
    E = lambda x: (x[0] * np.exp(x[1]) - 2 * x[1] * np.exp(-x[0])) ** 2
    dE_du = lambda x: (2 * (x[0] * np.exp(x[1]) - 2 * x[1] * np.exp(-x[0]))
                       * (np.exp(x[1]) + 2 * x[1] * np.exp(-x[0])))
    dE_dv = lambda x: (2 * (x[0] * np.exp(x[1]) - 2 * x[1] * np.exp(-x[0]))
                       * (x[0] * np.exp(x[1]) - 2 * np.exp(-x[0])))

    print(f"\n[HW5 P5-7]\nPerformance of descent methods for eta=0.1:")
    x, iters = gradient_descent(E, lambda x: np.array((dE_du(x), dE_dv(x))),
                                np.array((1, 1), dtype=float))
    print(f"  Gradient descent: {iters=}, x=({x[0]:.3f}, {x[1]:.3f})")
    x, iters = coordinate_descent(E, (dE_du, dE_dv),
                                  np.array((1, 1), dtype=float), max_iters=15)
    print(f"  Coordinate descent: {iters=}, x=({x[0]:.3f}, {x[1]:.3f}), "
          f"{E(x)=:.3e}")
```

Logistic Regression

In this problem, you will create your own target function f (probability in this case) and data set $\mathcal{D}$ to see how logistic regression works. For simplicity, we will take f to be a 0/1 probability so y is a deterministic function of $\mathbf{x}$.

Take $d = 2$ so you can visualize the problem and let $\mathcal{X} = [-1, 1] \times [-1, 1]$ with uniform probability of picking each $\mathbf{x} \in \mathcal{X}$. Choose a line in the plane as the boundary between $f(\mathbf{x}) = 1$ (where y must be $+1$) and $f(\mathbf{x}) = 0$ (where y must be -1) by taking two random, uniformly distributed points from $\mathcal{X}$ and taking the line passing through them as the boundary between $y = \pm 1$. Pick $N = 100$ training points at random from $\mathcal{X}$ and evaluate the outputs y_n for each of these points $\mathbf{x}_n$.

Run logistic regression with stochastic gradient descent to find g and estimate E_{out} (the cross-entropy error) by generating a sufficiently large, separate set of points to evaluate the error. Repeat the experiment for 100 runs with different targets and take the average. Initialize the weight vector of logistic regression to all zeros in each run. Stop the algorithm when $\left\|\mathbf{w}^{(t-1)} - \mathbf{w}^{(t)}\right\| < 0.01$, where $\mathbf{w}^{(t)}$ denotes the weight vector at the end of epoch t. An epoch is a full pass through the N data points (use a random permutation of 1, 2, …, N to present the data points to the algorithm within each epoch and use different permutations for different epochs). Use a learning rate of $\eta = 0.01$.

8. Which of the following is closest to E_{out} for $N = 100$?

   Answer: [d] 0.100

9. How many epochs does it take on average for logistic regression to converge for $N = 100$ using the above initialization and termination rules and the specified learning rate? Pick the value that is closest to your results.

   Answer: [a] 350

For the stochastic gradient descent, 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 8–9 is

```
[HW5 P8-9]
Stochastic gradient descent statistics over 100 runs:
  N=100, epochs=330, E_out=0.103
```

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

```python
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 stochastic_gradient_descent(
        N, f, eta=0.01, tol=0.01, *, N_test=1_000, rng=None, seed=None,
        hyp=False):
    if rng is None:
        rng = np.random.default_rng(seed)
    xs, ys = generate_data(N, f, bias=True, rng=rng)
    w = np.zeros(xs.shape[1], dtype=float)
    epoch = 0
    while True:
        _w = w.copy()
        ri = rng.permutation(np.arange(N))
        for x, y in zip(xs[ri], ys[ri]):
            _w += eta * y * x / (1 + np.exp(y * x @ _w))
        dw = _w - w
        w = _w
        epoch += 1
        if np.linalg.norm(dw) < tol:
            break

    x_test, y_test = generate_data(N_test, f, bias=True, rng=rng)
    E_out = np.log(1 + np.exp(-y_test[:, None] * x_test @ w)).mean()
    return (w, epoch, E_out)[1 - hyp:]

if __name__ == "__main__":
    rng = np.random.default_rng()
    N = 100
    n_runs = 100
    print("\n[HW5 P8-9]\nStochastic gradient descent statistics over "
          f"{n_runs:,} runs:")
    epochs, E_out = np.mean(
        [stochastic_gradient_descent(N, target_function_random_line(rng=rng),
                                     rng=rng) for _ in range(n_runs)],
        axis=0
    )
    print(f"  {N=}, {epochs=:.0f}, {E_out=:.3f}")
```

PLA as SGD

10. The perceptron learning algorithm can be implemented as SGD using which of the following error functions $\mathrm{e}_n(\mathbf{w})$? Ignore the points $\mathbf{w}$ at which $\mathrm{e}_n(\mathbf{w})$ is not twice differentiable.

Answer: [e] $\mathbf{e_n(w) = -\min(0,\ y_n w^\top x_n)}$

In the PLA, the weight vector update for point n is

$$\mathbf{w}(t+1) = \mathbf{w}(t) + y_n(t)\mathbf{x}_n(t)$$

In SGD, the weight vector update for point n is

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla \mathrm{e}_n\big(\mathbf{w}(t)\big)$$

To implement the PLA as SGD, we set $\eta = 1$ (as there is no "learning rate" in the PLA) and seek an error function e_n that satisfies $\nabla \mathrm{e}_n(\mathbf{w}) = -y_n\mathbf{x}_n$. The obvious candidate is $\mathrm{e}_n(\mathbf{w}) = -y_n\mathbf{w}^\top\mathbf{x}_n$, but it is important to note that $\mathrm{e}_n(\mathbf{w})$ is negative only when $\mathbf{x}_n$ is misclassified using the weights in the current iteration.

Therefore, the error function is actually $\mathrm{e}_n(\mathbf{w}) = -\min(0,\ y_n\mathbf{w}^\top\mathbf{x_n})$ so that no update is performed when $\mathbf{x}_n$ is correctly classified.