Homework # I

Daniel dos Santos
Universidade Federal do Rio de Janeiro

April 26, 2023

The Learning Problem

Problem 01

What types of Machine Learning, if any, best describe the following three scenarios:

(i) A coin classification system is created for a vending machine. The developers obtain exact coin specifications from the U.S. Mint and derive a statistical model of the size, weight, and denomination, which the vending machine then uses to classify coins.

(ii) Instead of calling the U.S. Mint to obtain coin information, an algorithm is presented with a large set of labeled coins. The algorithm uses this data to infer decision boundaries which the vending machine then uses to classify its coins.

(iii) A computer develops a strategy for playing Tic-Tac-Toe by playing repeatedly and adjusting its strategy by penalizing moves that eventually lead to losing.

[a] (i) Supervised Learning, (ii) Unsupervised Learning, (iii) Reinforcement Learning

[b] (i) Supervised Learning, (ii) Not learning, (iii) Unsupervised Learning

[c] (i) Not learning, (ii) Reinforcement Learning, (iii) Supervised Learning

[d] (i) Not learning, (ii) Supervised Learning, (iii) Reinforcement Learning

[e] (i) Supervised Learning, (ii) Reinforcement Learning, (iii) Unsupervised Learning

Solution

(i) The developers can leverage the known characteristics of a coin to inform their decision-making process, without needing to learn from any data.
(ii) Labeled data can be used to train a model to approximate an unknown function in a supervised learning setting, where the model learns to make predictions based on pre-labeled inputs and outputs.
(iii) The Tic-Tac-Toe example demonstrates how an agent can improve its decision-making abilities through reinforcement learning, by receiving feedback in the form of rewards or penalties as it interacts with the environment.

Answer: **[d]**

Problem 02

Which of the following problems are best suited for Machine Learning?

(i) Classifying numbers into primes and non-primes.

(ii) Detecting potential fraud in credit card charges.

(iii) Determining the time it would take a falling object to hit the ground.

(iv) Determining the optimal cycle for traffic lights in a busy intersection.

[a] (ii) and (iv)

[b] (i) and (ii)

[c] (i), (ii), and (iii)

[d] (iii)

[e] (i) and (iii)

Solution

(i) There is a deterministic way of knowing if a number is prime or not, making this problem unsuited for Machine Learning;
(ii) By using data from past good and bad payors you can create a model that analyzes a set of features, this way you can effectively predict credit risk for a given client;
(iii) A mathematical formula can solve this problem without requiring the use of Machine Learning.
(iv) The optimization of traffic lights in a busy intersection can be accomplished by using historical traffic data to train machine learning models, as well as by conducting simulations that can inform the development of reinforcement learning algorithms.

Answer: **[a]**

Bins and Marbles

Problem 03

We have 2 opaque bags, each containing 2 balls. One bag has 2 black balls and the other has a black ball and a white ball. You pick a bag at random and then pick one of the balls in that bag at random. When you look at the ball, it is black. You now pick the second ball from that same bag. What is the probability that this ball is also black?

[a] 1/4

[b] 1/3

[c] 1/2

[d] 2/3

[e] 3/4

Solution

Let $A =$ "First ball is BLACK" and $B =$ "Second ball is BLACK". We need to calculate the probability of the event B given that the event A has occurred, i.e., $P(B|A)$.
We can use the formula:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

Using the Law of Total Probability, we can calculate $P(A)$ as:

$$\begin{aligned} P(A) &= P(A|\text{"Bag 1"})P(\text{"Bag 1"}) + P(A|\text{"Bag 2"})P(\text{"Bag 2"}) \\ &= 1 * \frac{1}{2} + \frac{1}{2} * \frac{1}{2} \\ &= \frac{3}{4} \end{aligned}$$

Similarly, we can calculate $P(A \cap B)$ as:

$$\begin{aligned} P(A \cap B) &= P(A \cap B|\text{"Bag 1"})P(\text{"Bag 1"}) + P(A \cap B|\text{"Bag 2"})P(\text{"Bag 2"}) \\ &= 1 * \frac{1}{2} + 0 * \frac{1}{2} \\ &= \frac{1}{2} \end{aligned}$$

Substituting these values, we get:

$$\begin{aligned} P(B|A) &= \frac{P(A \cap B)}{P(A)} \\ &= \frac{1}{2} * \frac{4}{3} \\ &= \frac{2}{3} \end{aligned}$$

Answer: **[d]** $\frac{2}{3}$

Problem 04 and Problem 05

Consider a sample of 10 marbles drawn from a bin containing red and green marbles. The probability that any marble we draw is red is $\mu = 0.55$ (independently, with replacement). We address the probability of getting no red marbles ($\nu = 0$) in the following cases:

Problem 04

We draw only one such sample. Compute the probability that $\nu = 0$. The closest answer is ('closest answer' means: $\|$your answer $-$ given option$\|$ is closest to 0):

[a] 7.331×10^{-6}

[b] 3.405×10^{-4}

[c] 0.289

[d] 0.450

[e] 0.550

Solution

$P(\nu = 0) = (1 - \mu)^{10} = (1 - 0.55)^{10} = 3.405 \times 10^{-4}$

Answer: **[b]** 3.405×10^{-4}

Problem 05

We draw $1,000$ independent samples. Compute the probability that (at least) one of the samples has $\nu = 0$. The closest answer is:

[a] 7.331×10^{-6}

[b] 3.405×10^{-4}

[c] 0.289

[d] 0.450

[e] 0.550

Solution

From the previous problem, we know that the probability of one sample of size 10 not having any red marble is $P(\nu = 0) = 3.405 \times 10^{-4}$. Let X be a random variable that counts the number of samples of the same size that have no red marbles ($\nu = 0$) in 1000 repetitions:

$$X \sim Bin(1000; 3.405 \times 10^{-4})$$

The probability of any of the 1000 samples not having any red marble is:

$$\begin{aligned} P(X = 0) &= \binom{1000}{0} (3.405 \times 10^{-4})^0 (1 - 3.405 \times 10^{-4})^{1000} \\ &= (1 - 3.405 \times 10^{-4})^{1000} \\ &= 0.7113\ldots \end{aligned}$$

Now we can calculate the complement of the probability to find the probability of at least one sample having at least one red marble in it:

$$1 - P(X = 0) = 1 - 0.7113... = 0.2886... \approx 0.289$$

Therefore, the probability of at least one sample from the 1000 having at least one red marble is approximately 0.289.

Answer: **[c]** 0.289

The Perceptron Learning Algorithm

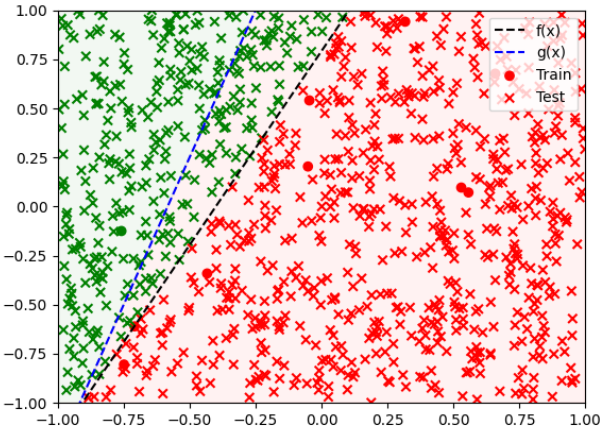
In this problem, you will create your own target function f and data set $\mathcal{D}$ to see how the Perceptron Learning Algorithm works. Take $d = 2$ so you can visualize the problem, and assume $X = [-1, 1] \times [-1, 1]$ with uniform probability of picking each $x \in \mathcal{X}$.

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 x_n of the data set as random points (uniformly in X), and evaluate the target function on each x_n to get the corresponding output y_n.

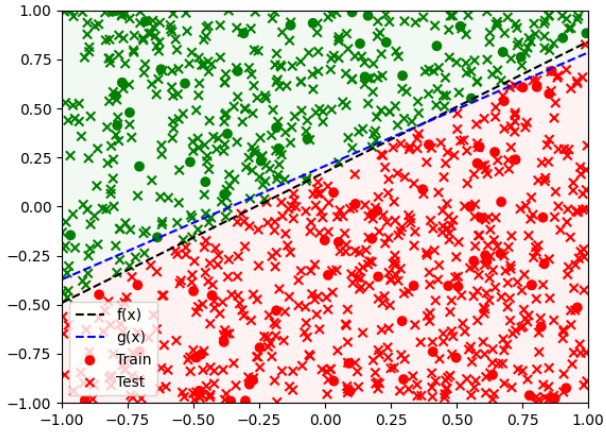
Now, in each run, use the Perceptron Learning Algorithm to find g. Start the PLA with the weight vector w being all zeros (consider $sign(0) = 0$, so all points are initially misclassified), and at each iteration have the algorithm choose a point randomly from the set of misclassified points. We are interested in two quantities: the number of iterations that PLA takes to converge to g, and the disagreement between f and g which is $P[f(x) \neq g(x)]$ (the probability that f and g will disagree on their classification of a random point). You can either calculate this probability exactly, or approximate it by generating a sufficiently large, separate set of points to estimate it.

In order to get a reliable estimate for these two quantities, you should repeat the experiment for 1000 runs (each run as specified above) and take the average over these runs.

Solution



Example of experiment (N=10)



Example of experiment (N=100)

Problem 7

Take $N = 10$. How many iterations does it take on average for the PLA to converge for $N = 10$ training points? Pick the value closest to your results (again, 'closest' means: $\|$your answer $-$ given option$\|$ is closest to 0).

[a] 1

[b] 15

[c] 300

[d] 5000

[e] 10000

Solution

Answer: **[b]** 10.947

Problem 8

Which of the following is closest to $P[f(x) \neq g(x)]$ for $N = 10$?

[a] 0.001

[b] 0.01

[c] 0.1

[d] 0.5

[e] 0.8

Solution

Answer: **[c]** 0.108

Problem 9

Now, try $N = 100$. How many iterations does it take on average for the PLA to converge for $N = 100$ training points? Pick the value closest to your results.

[a] 50

[b] 100

[c] 500

[d] 1000

[e] 5000

Solution

Answer: **[b]** 114.494

Problem 10

Which of the following is closest to $P[f(x) \neq g(x)]$ for $N = 100$?

[a] 0.001

[b] 0.01

[c] 0.1

[d] 0.5

[e] 0.8

Solution

Answer: **[b]** 0.0136

Appendices

The Perceptron Learning Algorithm

```
import random
import math
import statistics
from typing import Tuple, List

import matplotlib.pyplot as plt


class TargetFunction():
    """
    Class representing the target function that separates the data points into two classes.
    """

    def __init__(self, p: Tuple[float], q: Tuple[float]):
        """
        Initializes the TargetFunction with two points p and q.

        Args:
        - p: A tuple representing a point in 2D space.
        - q: A tuple representing a point in 2D space.
        """
        self.__m, self.__b = self.initialize(p, q)

    def initialize(self, p: Tuple[float], q: Tuple[float]) -> Tuple[float]:
        """
        Calculates the slope and the y-intercept of the line connecting points p and q.

        Args:
        - p: A tuple representing a point in 2D space.
        - q: A tuple representing a point in 2D space.

        Returns:
        - A tuple representing the slope and y-intercept of the line connecting points p and q.
        """
        m = (q[1] - p[1]) / (q[0] - p[0])
        b = m * q[0] - q[1]
        return m, b

    def classify(self, point: Tuple[float]) -> float:
        """
        Classifies a point based on its location relative to the target function.

        Args:
        - point: A tuple representing a point in 2D space.

        Returns:
        - 1 if the point is above the target function, -1 otherwise.
        """
        if point[1] > self.m * point[0] + self.b:
            return 1
        return -1

    @property
    def m(self) -> float:
        """
        Returns the slope of the target function.
        """
        return self.__m

    @property
    def b(self) -> float:
        """
        Returns the y-intercept of the target function.
```

```
        """
        return self.__b


class Perceptron():
    """
    Class representing a Perceptron model.
    """

    def __init__(self):
        """
        Initializes the Perceptron model with zero weights and bias.
        """
        self.__weights = (0, 0)
        self.__bias = 0

    def predict(self, inputs: List[float]) -> int:
        """
        Predicts the class of a data point.

        Args:
        - inputs: A list representing the data point.

        Returns:
        - 1 if the point is above the target function, -1 otherwise..
        """
        wtx = sum([w * x for w, x in zip(self.weights, inputs)]) + self.bias
        return int(math.copysign(1, wtx))

    def train(self, inputs: List[Tuple[float]], outputs: List[int]) -> int:
        """
        Trains the Perceptron model on a set of labeled data points using the perceptron learning algorithm.

        Args:
        - inputs: A list of tuples representing the features of each data point.
        - outputs: A list of integers representing the labels of each data point.

        Returns:
        - The number of iterations needed to converge.
        """
        predictions = [self.predict(i) for i in inputs]
        missclassified_points = self.__missclassified_points(
            inputs, outputs, predictions
        )

        iterations = 0
        while len(missclassified_points) > 0:
            iterations += 1
            predictions = [self.predict(x) for x in inputs]
            missclassified_points = self.__missclassified_points(
                inputs, outputs, predictions
            )
            if len(missclassified_points) > 0:
                # Random select a missclassfied point to update the weights
                random_missclassfied_point = random.choice(
                    missclassified_points
                )
                x = random_missclassfied_point[0]
                y = random_missclassfied_point[1]
                self.__update_weights(x, y)

        return iterations

    def test(self, inputs: List[Tuple[float]], outputs: List[int]) -> float:
        """
        Tests the accuracy of the Perceptron on the given input and output data.

        Args:
        - inputs: A list of input points, where each point is a tuple of floats.
        - outputs: A list of expected output classifications for each input point.
```

```
        Returns:
        - The ratio of misclassified points to the total number of input points.
        """
        predictions = [self.predict(x) for x in inputs]
        missclassified_points = self.__missclassified_points(
            inputs, outputs, predictions
        )

        return len(missclassified_points) / len(outputs)

    def __missclassified_points(self, inputs: List[Tuple[float]], outputs: List[int], predictions: List[int]):
        """
        Returns a list of misclassified points.

        Args:
        - inputs: A list of input points, where each point is a tuple of floats.
        - outputs: A list of expected output classifications for each input point.
        - predictions: A list of predicted output classifications for each input point.

        Returns:
        - A list of misclassified input points and their corresponding output and predicted classifications.
        """
        return list(
            filter(lambda x: x[1] != x[2], zip(inputs, outputs, predictions))
        )

    def __update_weights(self, x: Tuple[float], y: float) -> None:
        """
        Updates the weights and bias of the Perceptron based on a misclassified point.

        Args:
        - x: The misclassified input point as a tuple of floats.
        - y: The expected output classification of the misclassified point.

        """
        self.__bias += y
        self.__weights = [w + y * xi for w, xi in zip(self.weights, x)]

    @property
    def weights(self) -> float:
        """
        Returns the current weights of the Perceptron.
        """
        return self.__weights

    @property
    def bias(self) -> float:
        """
        Returns the current bias of the Perceptron.
        """
        return self.__bias


class Experiment():
    """
    A class representing an experiment that generates random points and trains a perceptron to classify them.
    """

    def __init__(self, input_size: int, experiment_size: int = 1000, test_size: int = 1000):
        self.__input_size = input_size
        self.__experiment_size = experiment_size
        self.__test_size = test_size
        self.__target_function = None
        self.__perceptron = None
        self.__iterations = []
        self.__errors = []

        self.__inputs = None
        self.__outputs = None
```

```
        self.__test_inputs = None
        self.__test_outputs = None

    def start(self) -> None:
        for _ in range(self.__experiment_size):
            self.__initialize()
            self.__inputs = self.__generate_points(self.__input_size)
            self.__outputs = self.__classify_points(self.__inputs)
            n_iterations = self.perceptron.train(self.__inputs, self.__outputs)

            self.__test_inputs = self.__generate_points(self.__test_size)
            self.__test_outputs = self.__classify_points(self.__test_inputs)
            accuracy = self.perceptron.test(
                self.__test_inputs, self.__test_outputs
            )

            self.__errors.append(accuracy)
            self.__iterations.append(n_iterations)

    def plot(self) -> None:
        intercept = -self.perceptron.bias/self.perceptron.weights[1]
        slope = -self.perceptron.weights[0]/self.perceptron.weights[1]

        x = self.__linespace(-1, 1, n=1000)
        y_pred = list(map(lambda x: slope * x + intercept, x))
        y = list(map(lambda x: self.target_function.m *
                 x + self.target_function.b, x))

        plt.plot(x, y, 'k--', label="f(x)")
        plt.plot(x, y_pred, 'b--', label="g(x)")
        plt.fill_between(x, 1, y_pred, color='green', alpha=0.05)
        plt.fill_between(x, y_pred, -1, color='red', alpha=0.05)

        x = [x[0] for x in self.__inputs]
        y = [x[1] for x in self.__inputs]
        c = ['g' if y == 1 else 'r' for y in self.__outputs]
        plt.scatter(x, y, c=c, marker="o", label="Train")

        x = [x[0] for x in self.__test_inputs]
        y_pred = [x[1] for x in self.__test_inputs]
        c = ['g' if y_pred == 1 else 'r' for y_pred in self.__test_outputs]
        plt.scatter(x, y_pred, c=c, marker="x", label="Test")

        plt.xlim((-1, 1))
        plt.ylim((-1, 1))
        plt.legend()

    def __initialize(self) -> None:
        p, q = self.__generate_points(2)
        self.__target_function = TargetFunction(p, q)
        self.__perceptron = Perceptron()

    def __generate_points(self, size: int = 10) -> List[Tuple[float]]:
        return [(random.uniform(-1, 1), random.uniform(-1, 1)) for _ in range(size)]

    def __classify_points(self, inputs: List[Tuple[float]]) -> List[float]:
        return [self.target_function.classify(i) for i in inputs]

    def __linespace(self, lower: float, upper: float, n: int = 100) -> List[float]:
        return [lower + x*(upper - lower)/n for x in range(n)]

    @property
    def perceptron(self) -> Perceptron:
        return self.__perceptron

    @property
    def target_function(self) -> TargetFunction:
        return self.__target_function

    @property
```

```
    def mean_error(self) -> float:
        return statistics.mean(self.__errors)

    @property
    def mean_iterations(self) -> float:
        return statistics.mean(self.__iterations)


def run_experiments():
    experiment = Experiment(input_size=10)
    experiment.start()
    experiment.plot()
    plt.savefig("N10.png")
    print(
        f"Approx P(f(x)g(x)): {experiment.mean_error},\
        \nMean iterations: {experiment.mean_iterations}"
    )
    # plt.show()
    plt.clf()

    experiment = Experiment(input_size=100)
    experiment.start()
    experiment.plot()
    plt.savefig("N100.png")
    print(
        f"Approx P(f(x)g(x)): {experiment.mean_error},\
        \nMean iterations: {experiment.mean_iterations}"
    )
    # plt.show()
    plt.clf()


if __name__ == "__main__":
    run_experiments()
```