```
1.
(1). sign(w(t)·X*) \neq y*, then sign(w<sup>T</sup>(t)·X*) \neq y*
     y * w^{T}(t) x * < 0
(2). W(t+1) = W(t) + y*X*, then W^{T}(t+1) = W^{T}(t) + y*X*
     y * w^{T}(t+1) \times * = y * (w^{T}(t) + y * \times *) \times *
                    = y*WT(t) X*+ y*y* X*X*
     As y*y* must be positive, y*y*x*x*>0
     So y*w^{T}(t+1)X* > y*w^{T}(t)X*
2.
(a), 185
    import math
    \delta = 0.05
    c = 0.1
    size = math.log(\delta / 2) / (-2 * c ** 2)
    size = math.ceil(size)
(b) 4612
    import math
    \delta = 0.05
    c = 0.02
    size = math.log(\delta / 2) / (-2 * c ** 2)
    size = math.ceil(size)
(c) 204938
     import math
```

 $\delta = 0.05$ c = 0.003

size = math.log(δ / 2) / (-2 * c ** 2)

size = math.ceil(size)

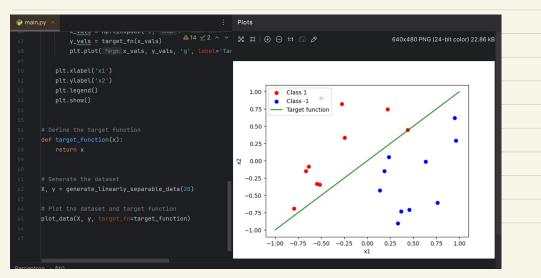
```
3 Probability = 5.62237059758089e-13
   import math
   \delta = 0.85
   N = 20
   prob = 2 * math.exp(-2 * \delta ** 2* N)
4.
(a) import numpy as np
     import matplotlib.pyplot as plt
     class Perceptron:
       def __init__(self, eta=0.5, n_iter=10):
          self.eta = eta
          self.n iter = n iter
        def fit(self, X, y):
          self.w_ = np.zeros(1 + X.shape[1])
          self.errors_ = []
          for _ in range(self.n_iter):
             errors = \Pi
             # iterate samples one by one and update the weights
             for xi, target in zip(X, y):
               update = self.eta * (target - self.predict(xi))
                self.w_[0] += update
                self.w_[1:] += update * xi
                errors.append(int(update != 0.0))
             self.errors .append(sum(errors) if len(errors) > 0 else 0.)
          return self
        def net_input(self, X):
           """Calculate net input before activation"""
          return np.dot(X, self.w [1:]) + self.w [0]
        def predict(self, X):
          return np.where(np.dot(X, self.weights) \geq 0.0, 1, -1)
```

```
# Generate a linearly separable dataset
def generate_linearly_separable_data(n):
  X = np.random.uniform(low=-1, high=1, size=(n, 2))
  y = np.where(X[:, 1] > X[:, 0], 1, -1)
  return X, y
# Plot the dataset and target function
def plot_data(X, y, target_fn=None):
  plt.scatter(X[y == 1, 0], X[y == 1, 1], c='r', label='Class 1')
  plt.scatter(X[y == -1, 0], X[y == -1, 1], c='b', label='Class -1')
  if target_fn is not None:
     x vals = np.linspace(-1, 1, 100)
     y_vals = target_fn(x_vals)
     plt.plot(x_vals, y_vals, 'g', label='Target function')
  plt.xlabel('x1')
  plt.ylabel('x2')
  plt.legend()
  plt.show()
# Define the target function
def target_function(x):
  return x
```

Generate the dataset

X, y = generate_linearly_separable_data(20)

Plot the dataset and target function plot_data(X, y, target_fn=target_function)



```
self.errors = []
             for in range(self.n iter):
               error = 0
               for xi, target in zip(X, y):
                  xi = np.insert(xi, 0, 1) # Insert a bias
       term
                  update = self.eta * (target -
       self.predict(xi))
                  self.weights += update * xi
                  error += int(update != 0.0)
               self.errors.append(error)
               if error == 0:
                  break
          def predict(self, X):
             return np.where(np.dot(X, self.weights) >=
       0.0, 1, -1)
       # Generate a linearly separable dataset
       def generate_linearly_separable_data(n):
          X = np.random.uniform(low=-1, high=1,
       size=(n, 2))
          y = np.where(X[:, 1] > X[:, 0], 1, -1)
          return X, y
 1.0
 0.5
0.0
-0.5
-1.0
                                            Target function
                                            Final hypothesis
     -1.00 -0.75 -0.50 -0.25
                             0.00
                                                0.75
                              x1
```

import numpy as no

class Perceptron:

(h) import matplotlib.pvplot as plt

self.eta = eta

def fit(self, X, y):

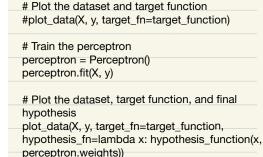
Initialize weights to zero

self.n iter = n iter

def __init__(self, eta=0.5, n_iter=10):

self.weights = np.zeros(X.shape[1] + 1) #

```
# Plot the dataset, target function, and
hypothesis
def plot_data(X, y, target_fn=None,
hypothesis_fn=None):
  plt.scatter(X[y == 1, 0], X[y == 1, 1], c='r',
label='Class 1')
  plt.scatter(X[y == -1, 0], X[y == -1, 1], c='b',
label='Class -1')
  if target fn is not None:
     x_vals = np.linspace(-1, 1, 100)
     y_vals = target_fn(x_vals)
     plt.plot(x vals, y vals, 'g', label='Target
function')
  if hypothesis fn is not None:
     x vals = np.linspace(-1, 1, 100)
     y_vals = hypothesis_fn(x_vals)
     plt.plot(x vals, y vals, 'm', label='Final
hypothesis')
  plt.xlabel('x1')
  plt.ylabel('x2')
  plt.legend()
  plt.show()
# Define the target function
def target_function(x):
  return x
# Define the final hypothesis function
def hypothesis function(x, w):
  return (-w[0] - w[1] * x) / w[2]
```



num updates = len(perceptron.errors) - 1 #

print("Number of updates:", num_updates)

Report the number of updates

Subtract 1 for the initial weights

X, y = generate_linearly_separable_data(20)

Generate the dataset

f is close to g, but it's still obvious that f is not the same with g.

class Perceptron:
def __init__(self, eta=0.5, n_iter=10):
 self.eta = eta
 self.n_iter = n_iter

def fit(self, X, y):
 self.w_ = np.zeros(1 + X.shape[1])
 self.errors_ = []

errors = []
iterate samples one by one and update the weights
for xi, target in zip(X, y):

update = self.eta * (target - self.predict(xi))

self.w_[0] += update

self.w_[1:] += update * xi

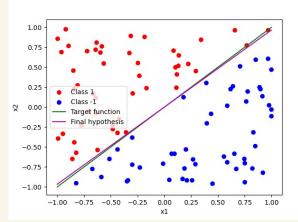
self.errors_.append(sum(errors) if len(errors) > 0 else
0.)
return self

for in range(self.n iter):

def predict(self, X): return np.where(np.dot(X, self.w_[1:]) + self.w_[0] >= 0.0, 1, -1)

Generate a linearly separable dataset
def generate_linearly_separable_data(n):
 X = np.random.uniform(low=-1, high=1, size=(n, 2))
 y = np.where(X[:, 1] > X[:, 0], 1, -1)
 return X, y

errors.append(int(update != 0.0))



f is closer to a comparing to (b).

Plot the dataset, target function, and hypothesis def plot_data(X, y, target_fn=None, hypothesis_fn=None):
 plt.scatter(X[y == 1, 0], X[y == 1, 1], c='r', label='Class 1')
 plt.scatter(X[y == -1, 0], X[y == -1, 1], c='b', label='Class -1')

if target_fn is not None:

x_vals = np.linspace(-1, 1, 100)

y_vals = target_fn(x_vals)

plt.plot(x_vals, y_vals, 'g', label='Target function')

if hypothesis_fn is not None:
 x_vals = np.linspace(-1, 1, 100)
 y_vals = hypothesis_fn(x_vals)
 plt.plot(x_vals, y_vals, 'm', label='Final hypothesis')

plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()

Define the target function def target_function(x):
return x

Define the final hypothesis function def hypothesis_function(x, w): return (-w[0] - w[1] * x) / w[2]

Generate the dataset
X, y = generate_linearly_separable_data(100)

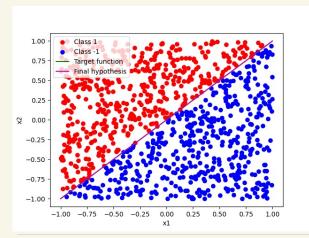
Plot the dataset and target function #plot_data(X, y, target_fn=target_function)

Train the perceptron perceptron = Perceptron() perceptron.fit(X, y)

Plot the dataset, target function, and final hypothesis plot_data(X, y, target_fn=target_function, hypothesis_fn=lambda x: hypothesis_function(x, perceptron.w_))

Report the number of updates num_updates = len(perceptron.errors_) - 1 # Subtract 1 for the initial weights print("Number of updates:", num_updates)

```
(d), import numpy as np
       import matplotlib.pyplot as plt
       class Perceptron:
          def __init__(self, eta=0.5, n_iter=10):
            self.eta = eta
            self.n iter = n iter
          def fit(self, X, v):
            self.w_ = np.zeros(1 + X.shape[1])
            self.errors = []
            for _ in range(self.n_iter):
               errors = []
               # iterate samples one by one and update the weights
               for xi, target in zip(X, y):
                  update = self.eta * (target - self.predict(xi))
                  self.w_[0] += update
                  self.w_[1:] += update * xi
                  errors.append(int(update != 0.0))
               self.errors_.append(sum(errors) if len(errors) > 0 else
       0.)
             return self
          def predict(self, X):
             return np.where(np.dot(X, self.w [1:]) + self.w [0] >= 0.0,
       1. -1)
       # Generate a linearly separable dataset
       def generate linearly separable data(n):
          X = np.random.uniform(low=-1, high=1, size=(n, 2))
          y = np.where(X[:, 1] > X[:, 0], 1, -1)
          return X, y
```



f is much closer to g than in (b), and f and g are almost the same.

```
# Plot the dataset, target function, and
hypothesis
def plot_data(X, y, target_fn=None,
hypothesis fn=None):
  plt.scatter(X[y == 1, 0], X[y == 1, 1], c='r',
label='Class 1')
  plt.scatter(X[y == -1, 0], X[y == -1, 1], c='b',
label='Class -1')
  if target fn is not None:
     x vals = np.linspace(-1, 1, 100)
     y_vals = target_fn(x_vals)
     plt.plot(x_vals, y_vals, 'g', label='Target
function')
  if hypothesis_fn is not None:
     x vals = np.linspace(-1, 1, 100)
     y_vals = hypothesis_fn(x_vals)
     plt.plot(x_vals, y_vals, 'm', label='Final
hypothesis')
  plt.xlabel('x1')
  plt.ylabel('x2')
  plt.legend()
  plt.show()
# Define the target function
def target function(x):
   return x
```

Define the final hypothesis function def hypothesis_function(x, w): return (-w[0] - w[1] * x) / w[2]

Generate the dataset X, y = generate_linearly_separable_data(1000)

Plot the dataset and target function #plot_data(X, y, target_fn=target_function)

Train the perceptron perceptron = Perceptron() perceptron.fit(X, y)

Plot the dataset, target function, and final hypothesis plot_data(X, y, target_fn=target_function, hypothesis_fn=lambda x: hypothesis_function(x, perceptron.w_))

Report the number of updates num_updates = len(perceptron.errors_) - 1 # Subtract 1 for the initial weights print("Number of updates:", num_updates)

import numpy as np	# Plot the dataset, target function, and
(e) import matplotlib.pyplot as plt	hypothesis
h	def plot_data(X, y, target_fn=None,
class Perceptron:	hypothesis_fn=None):
definit(self, eta=0.5, n_iter=10):	# Plot only the first two dimensions
self.eta = eta	plt.scatter(X[y == 1, 0], X[y == 1, 1], c='r',
	label='Class 1')
self.n_iter = n_iter	plt.scatter(X[y == -1, 0], X[y == -1, 1], c='b',
	label='Class -1')
def fit(self, X, y):	
$self.w_ = np.zeros(1 + X.shape[1])$	if target_fn is not None:
self.errors_ = []	
	x_vals = np.linspace(-1, 1, 100)
for _ in range(self.n_iter):	y_vals = target_fn(x_vals)
errors = []	plt.plot(x_vals, y_vals, 'g', label='Target
# iterate samples one by one and	function')
update the weights	
	if hypothesis_fn is not None:
for xi, target in zip(X, y):	x_vals = np.linspace(-1, 1, 100)
update = self.eta * (target -	y_vals = hypothesis_fn(x_vals)
self.predict(xi))	plt.plot(x_vals, y_vals, 'm', label='Final
self.w_[0] += update	hypothesis')
self.w_[1:] += update * xi	11, POLITOGIO /
errors.append(int(update != 0.0))	plt vlobal/lv1\
self.errorsappend(sum(errors) if	plt.xlabel('x1')
len(errors) > 0 else 0.)	plt.ylabel('x2')
return self	plt.legend()
return seit	plt.show()
def predict(self, X):	# Define the target function
return np.where(np.dot(X, self.w_[1:]) -	+ def target function(x):
self.w_[0] >= 0.0, 1, -1)	return x
# Generate a linearly separable dataset in	# Define the final hypothesis function
R^10	def hypothesis_function(x, w):
def generate_linearly_separable_data(n):	return (-w[0] - w[1] * x) / w[2]
	return (-w[o] - w[r] X) / w[z]
X = np.random.uniform(low=-1, high=1,	# O
size=(n, 10))	# Generate the dataset
y = np.where(X[:, 9] > X[:, 0], 1, -1)	X, y = generate_linearly_separable_data(1000)
return X, y	# Plot the dataset and target function
	#plot_data(X, y, target_fn=target_function)
	#plot_data(x, y, target_in=target_idiretion)
	# Train the percentran
	# Train the perceptron
	perceptron = Perceptron()
	perceptron.fit(X, y)
Number of undates - a	
Number of updates = 9	# Plot the dataset, target function, and final
	hypothesis
	plot_data(X, y, target_fn=target_function,
	hypothesis_fn=lambda x:
	hypothesis_function(x, perceptron.w_))
	# Report the number of updates
	num_updates = len(perceptron.errors_) - 1 #
	Subtract 1 for the initial weights
	print("Number of updates:", num_updates)
	print(Number of updates: , num_updates)

```
(f). The bigger N is (assume the dataset is linear separable), the more accuracy we gain, and the longer running time it will take.
```

The bigger of is, the less accurate the prediction is, and the longer running time it will take.

5.

$$\mu = 0.05$$

probability_1 = probability(n,
$$\mu$$
)

$$\mu = 0.6$$
 $\mu = 0.6$

probability_2 = probability(n,
$$\mu$$
)

$$\mu = 0.9$$

 $\mu = 0.9$

```
(b). import math
      def probability(n, μ):
         v = 0
         p = math.comb(n, v) * (\mu ** v) * ((1 - \mu) ** (n - v))
         return p
      n = 10
      sample = 1000
      \mu = 0.05
      \mu = 0.05
      probability_not_zero = 1 - probability(n, μ)
      probability_at_least_one_zero = 1 - probability_not_zero ** sample
      print(probability_at_least_one_zero)
      \# \mu = 0.6
      \mu = 0.6
      probability_not_zero = 1 - probability(n, μ)
      probability_at_least_one_zero = 1 - probability_not_zero ** sample
      print(probability at least one zero)
      \# \mu = 0.9
      \mu = 0.9
      probability_not_zero = 1 - probability(n, μ)
      probability_at_least_one_zero = 1 - probability_not_zero ** sample
      print(probability at least one zero)
Output: 1.0
         0.09955221269675618
         1.0000000327803349e-07
```

```
(c). import math
     def probability(n, μ):
       v = 0
       p = math.comb(n, v) * (\mu ** v) * ((1 - \mu) ** (n - v))
       return p
     n = 10
     sample = 1000000
     \# \mu = 0.05
     \mu = 0.05
     probability_not_zero = 1 - probability(n, µ)
     probability_at_least_one_zero = 1 - probability_not_zero ** sample
     print(probability at least one zero)
     \# \mu = 0.6
     \mu = 0.6
     probability_not_zero = 1 - probability(n, µ)
     probability at least one zero = 1 - probability not zero ** sample
     print(probability at least one zero)
     \# u = 0.9
     \mu = 0.9
     probability_not_zero = 1 - probability(n, µ)
     probability_at_least_one_zero = 1 - probability_not_zero ** sample
     print(probability at least one zero)
Output: 1.0
           9.999500844481979e-05
(d). For \mu = 0.05, the result increases from (a) to (b) and keeps 1.0 from (b) to (c);
     for \mu = 0.6 and \mu = 0.9, the result increases continuously from (a) to (c)
In conclusion, the bigger u is, the smaller the result will be:
                the more samples included, the bigger the result will be.
```

