1.

(1). $\mathrm{sign}(w(t) \cdot x_*) \neq y_*$, then $\mathrm{sign}(w^T(t) \cdot x_*) \neq y_*$

$y_* w^T(t) x_* < 0$

(2). $w(t+1) = w(t) + y_* x_*$, then $w^T(t+1) = w^T(t) + y_* x_*$

$y_* w^T(t+1) x_* = y_* (w^T(t) + y_* x_*) x_*$

$= y_* w^T(t) x_* + y_* y_* x_* x_*$

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

δ = 0.05
c = 0.1
size = math.log(δ / 2) / (-2 * c ** 2)
size = math.ceil(size)
```

(b) 4612

```
import math

δ = 0.05
c = 0.02
size = math.log(δ / 2) / (-2 * c ** 2)
size = math.ceil(size)
```

(c) 204938

```
import math

δ = 0.05
c = 0.003
size = math.log(δ / 2) / (-2 * c ** 2)
size = math.ceil(size)
```

3 Probability = 5.62237059758089e-13

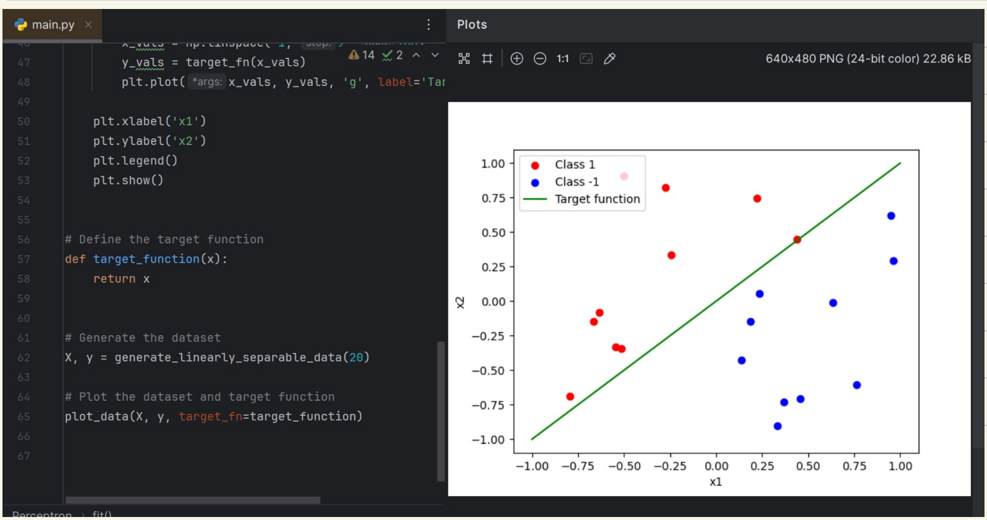
```
import math

δ = 0.85
N = 20
prob = 2 * math.exp(-2 * δ ** 2* N)
```

4.

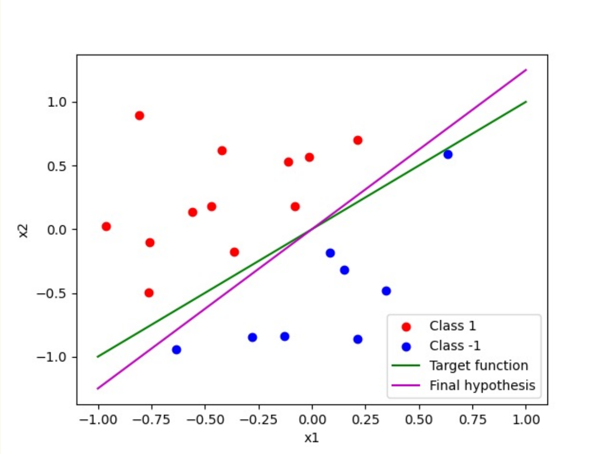
(a).

```
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 = []
            # 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) >= 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)
```



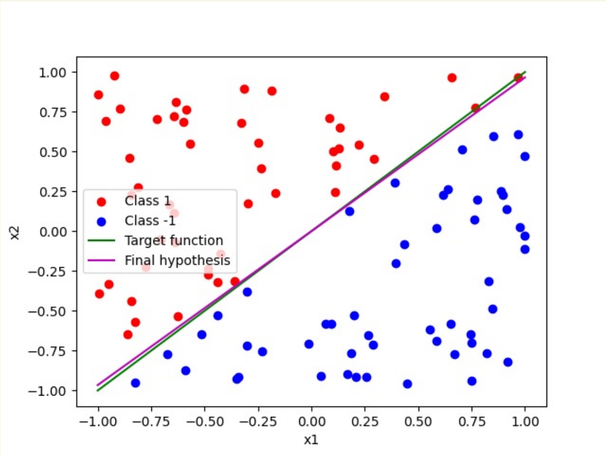
(b).
```
import numpy as np
import matplotlib.pyplot as plt

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.weights = np.zeros(X.shape[1] + 1)  # Initialize weights to zero
        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
```




```
# 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(20)

# 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.weights))

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

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

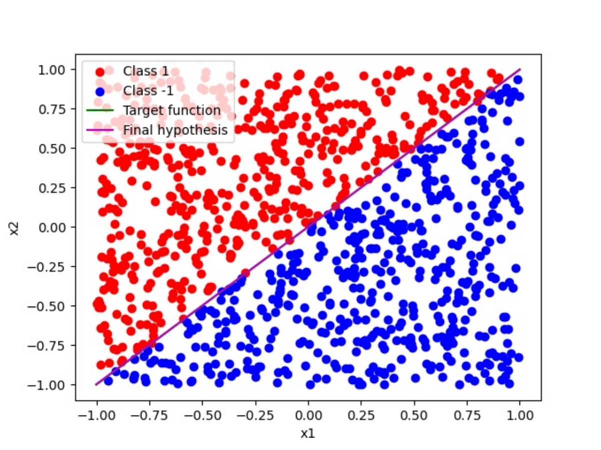
(c).
```
import numpy as np
import matplotlib.pyplot as plt

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 = []
            # 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 closer to g 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, y):
        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)
```

(e).

```
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 = []
            # 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 in
R^10
def generate_linearly_separable_data(n):
    X = np.random.uniform(low=-1, high=1,
size=(n, 10))
    y = np.where(X[:, 9] > X[:, 0], 1, -1)
    return X, y
```

Number of updates = 9

```
# Plot the dataset, target function, and
hypothesis
def plot_data(X, y, target_fn=None,
hypothesis_fn=None):
    # Plot only the first two dimensions
    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)
```

(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 d is, the less accurate the prediction is, and the longer running time it will take.

5.

(a).

```
import math

def probability(n, μ):
    v = 0
    p = math.comb(n, v) * (μ ** v) * ((1 - μ) ** (n - v))
    return p

# μ = 0.05
n = 10
μ = 0.05
probability_1 = probability(n, μ)

# μ = 0.6
μ = 0.6
probability_2 = probability(n, μ)

# μ = 0.9
μ = 0.9
probability_3 = probability(n, μ)

print(str(probability_1) + '\n' + str(probability_2) + '\n' + str(probability_3))
```

Output:

```
0.5987369392383787
0.00010485760000000006
9.999999999999978e-11
```

(b).
```
import math

def probability(n, μ):
    v = 0
    p = math.comb(n, v) * (μ ** v) * ((1 - μ) ** (n - v))
    return p

n = 10
sample = 1000

# μ = 0.05
μ = 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)

# μ = 0.6
μ = 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)

# μ = 0.9
μ = 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) * (μ ** v) * ((1 - μ) ** (n - v))
    return p

n = 10
sample = 1000000

# μ = 0.05
μ = 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)

# μ = 0.6
μ = 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)

# μ = 0.9
μ = 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
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)

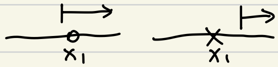
In conclusion, the bigger μ is, the smaller the result will be;

the more samples included, the bigger the result will be.

6.

(1). $m_{\mathcal{H}}(N) = N+1$

(2). $m_{\mathcal{H}}(N) = 2$ if $N=1$



$m_{\mathcal{H}}(N) = (1+N)\cdot N/2+1 = \frac{1}{2}N+\frac{1}{2}N^2+1$ if $N>1$