UNIVERSIDAD ADOLFO IBÁÑEZ

Tarea 2:
Multi-Armed Bandits and Reinforcement Learning.

(Acceso al GitHub MIA_ReinforcementLearning del autor de este informe en siguiente enlace)

Acceso al notebook Python de la tarea Nº 1 en siguiente enlace.
Acceso al notebook Python de la tarea Nº 2 en siguiente enlace.

Estudiante:
Eduardo Carrasco

Profesor:
Jorge Vásquez

Curso:
Reinforcement Learning. MIA, 2022

Tabla de contenido

I. Descripción de la tarea y requerimientos:

A. Requerimientos:

1. Tarea individual, no trabajar en grupo.
2. Entrega para el 28 de junio.
3. Entregable es un PDF del resumen de la tarea, parte del su código y el gráfico solicitado.

B. Objetivo:

Implementar Multi-Armed Bandits.

C. Descripción:

1. Vamos a usar “10-arm testbed” descrito en el libro del curso (Sutton, Section 2.3). Donde hay 10 palancas (*K=10 arms*), la recompensa promedio para cada palanca es *N(0,1)* y cuando el agente tira la palanca *A*, se observa una recompensa con ruido *r ~ N(r(a),1)*, esto significa *r(a)+ N(0,1)*.
2. Implementa una exploración Є-greedy. Plotear la recompensa esperada en el eje de las Y, y timesteps en el eje X para los siguientes valores Є:[0, 0.001, 0.01, 0.1, 1.0].
3. Los resultados para cada valor de Є deben ser representados en diferentes líneas del mismo gráfico.
4. Haz correr el algoritmo 20 veces para cada valor Є y toma el promedio de esos experimentos.
5. Se computan las recompensas esperadas de la forma: $\sigma_a\, r(a)\pi(a)$.

II. Desarrollo de la tarea:

El problema Multi-Armed Bandits consiste en un número determinado *K* de máquinas tragamonedas (*row of slot machines: "Bandits"*). En cada tiempo *t* el jugador debe elegir una de las *K* máquinas, la cual le devolverá una recompensa.
Como objetivo principal del sistema, se plantea maximizar la recompensa acumulada, encontrando la mejor máquina en el menor tiempo posible.

Dentro de los escenarios que el jugador puede obtener, se encuentran:

1. Que una de las *K* Máquinas esté dando muchos premios hacíendonos pensar como que es la mejor, pero no siendo la mejor.
2. Por otra parte, una de las K máquinas no dé premios justo en el instante que nosotros jugamos.

Todo lo anterior, crea un ambiente de inseguridad que trata de responder la pregunta ¿Es esta la mejor máquina para jugar? o ¿existe uno mejor?.

Lo anterior, se observa en el **dilema Explorar o Explotar**, lo cual se profundizará en las secciones precedentes.

III. Setup del problema:

Habiendo descrito el problema en el punto anterior, objeto definir técnicamente lo que se requiere, se tiene la siguiente ecuación:

$$Qt(a) = E[Rn|An = a]$$

En esta ecuación $Qt(a)$ es la recompensa estimada de (Rn) cuando se elige la acción An del paso n.

Para desarrollar lo anterior, se desarrollará un modelo que cubra el valor verdadero (*media - promedio*) de cada acción usando distribución normal (Gaussiana).

A. Explotación:

De acuerdo a lo requerido en el enunciado, se utilizará un **Greedy Action**, de manera de tomar la acción que se piensa traerá el mayor beneficio en cada paso (*timestep*). Como se describe en la siguiente ecuación:

$$A\_n = MAX\_a(Q\_n(a))$$

Se puede observar que la expectación máxima de greedy es A_n^* , que simboliza la explotación del **dilema explorar-explotar**, como primera aproximación, hacer repetidamente permite maximizar nuestra recompensa.

B. Exploración:

Como fue presentado en el punto anterior, se tiene un algoritmo que permita obtener la explotación. Pero, de igual manera se requiere uno que efectúe la exploración del espacio de búsqueda para ejecutar las mejores acciones.

Para ejecutar el cálculo, necesitamos sólo 2 valores conocidos, la media y el número de pasos dados.

$$m\_n = m\_\{n - 1\} + \frac{\{R\_n + m\_\{n-1\}}{n}$$

Como se observa en la ecuación, si se requiere calcular la media en el tiempo n, m_n, se puede efectuar con la media del tiempo anterior, m_{n-1}.

IV. Implementación del Método ϵ-Greedy:

A diferencia del método greedy puro (explicado anteriormente), este introduce un ϵ que permite efectuar la exploración a modo de probabilidad.

Para efectuar la modelación, se definen las siguientes variables:

1) `eps_bandit`: es la clase del experimento.
2) `k` : es la cantdad de máquinas.
3) `eps` : es la probabilidad (ruído) epsilon del modelo.
4) `iters` : es el número de iteraciones.
5) `mu` : es la recompensa que se ajusta para cada máquina, por default es N (0,1).

Código se observa en Anexo A de este documento.

A. Método de ϵ-Greedy - 1er modelo:

Se efectuará un modelado utilizando el método de ϵ-Greedy, con las siguientes características:

1) `k` = 10 máquinas.
2) `iters` = 3000 iteraciones.
3) `episodes` = 20 experimentos.
4) `eps_0_rewards` = recompensas con epsilon = 0 (greedy).
5) `eps_001_rewards` = recompensas con epsilon = 0.001.
6) `eps_01_rewards` = recompensas con epsilon = 0.01.
7) `eps_1_rewards` = recompensas con epsilon = 0.1.
8) `eps_10_rewards` = recompensas con epsilon = 1.0.

Código se observa en Anexo A de este documento.

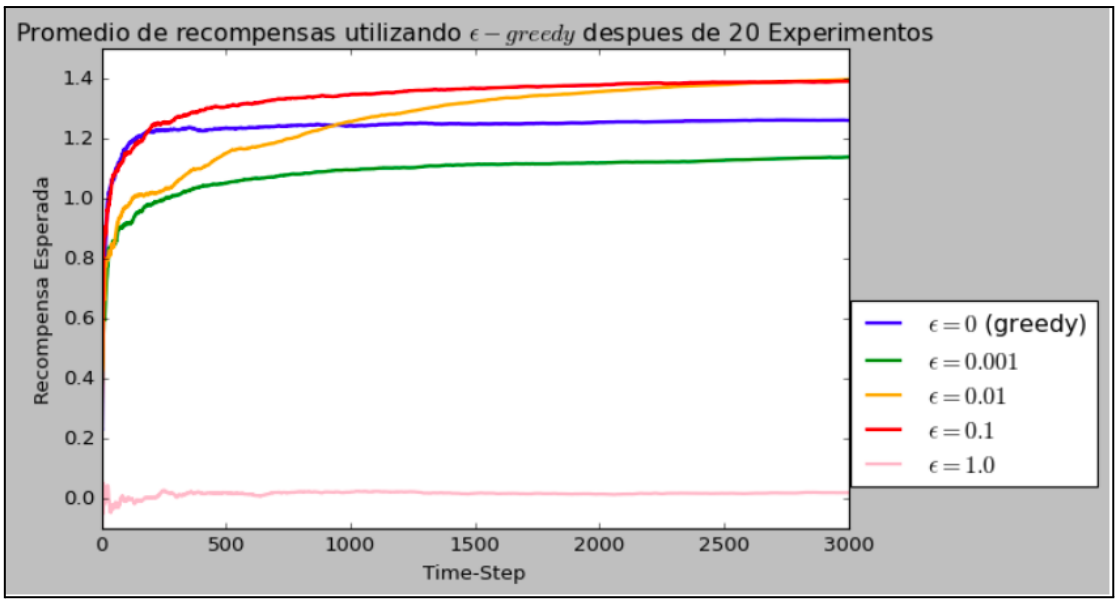

Figura Nº 1: Gráfico promedio de recompensas utilizando e-greedy.

Como se observa en el gráfico, la función `eps_1_rewards`, que involucra un **epsilon de 0.1** es que la logra maximizar la recompensa.

B. Método de ϵ-Greedy - 2do modelo:

Con el objeto graficar, de mejor manera cuales acciones proveen la mejor recompensa, se puede setear _mu=sequence_, de manera de sequenciar las recompensas y permitiendo observar de mejor manera, cual es la función que genera el mayor beneficio.

Código se observa en Anexo A de este documento.

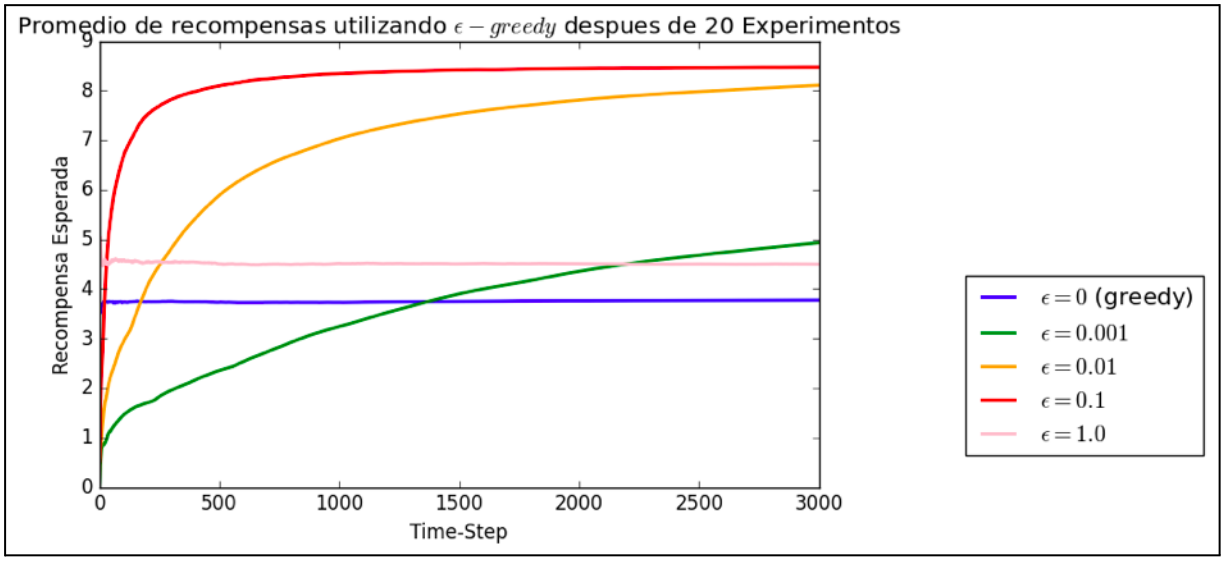

Figura Nº 2: Gráfico de recompensas utilizando e-greedy.

Como se observa, de igual manera se obtuvo el `eps_1_rewards`, de epsilon = 0.1, para lo cual, se mostrará un cuadro que represente el número de acciones efectuadas por cada acción.

Código se observa en Anexo A de este documento.

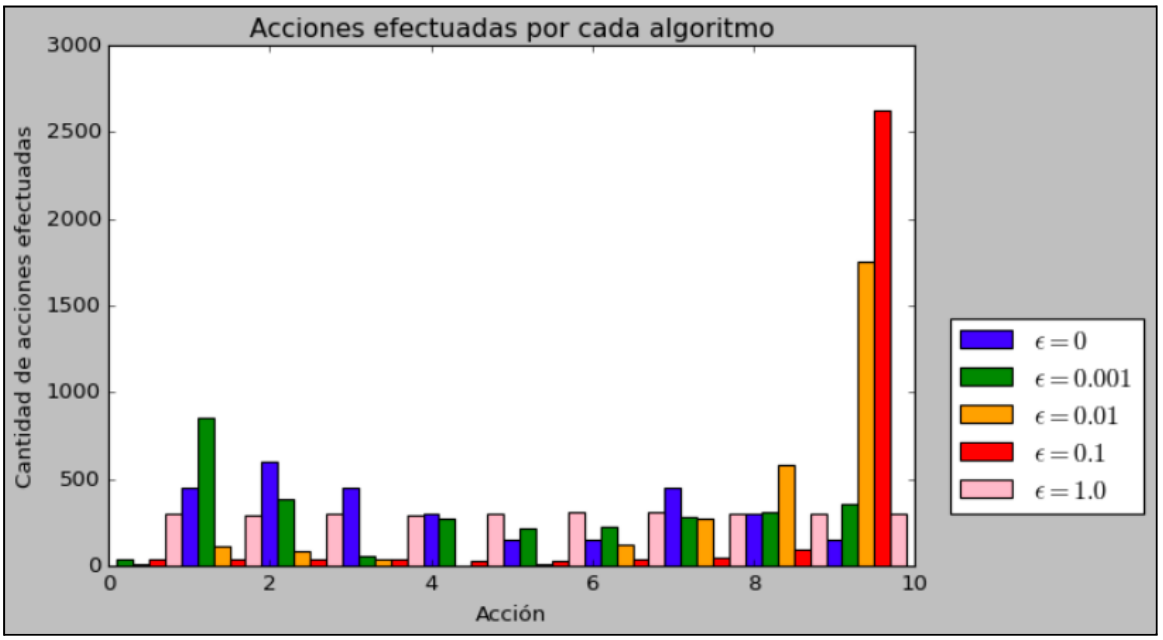

Figura Nº 3: Gráfico de cantidad de acciones utilizando e-greedy.

Porcentaje de acciones seleccionadas:

	a = 0	a = 1	a = 2	a = 3	a = 4	a = 5	a = 6	a = 7	a = 8	a = 9
$\epsilon = 0$	0.013333	15.001667	19.985000	15.000000	10.000000	5.000000	5.000000	15.000000	10.000000	5.000000
$\epsilon = 0.001$	1.358333	28.523333	12.948333	1.731667	8.996667	7.273333	7.515000	9.318333	10.320000	12.015000
$\epsilon = 0.01$	0.306667	3.773333	2.778333	1.363333	0.110000	0.330000	4.071667	9.188333	19.528333	58.550000
$\epsilon = 0.1$	1.110000	1.240000	1.098333	1.120000	1.025000	1.066667	1.196667	1.501667	3.108333	87.533333
$\epsilon = 1.0$	10.090000	9.821667	10.011667	9.750000	9.898333	10.270000	10.191667	10.070000	9.851667	10.045000

Figura Nº 4: Tabla comparativa de cantidad de acciones utilizando e-greedy.

Si observamos la selección promedio de los algoritmos, podemos ver por qué el mayor valor de epsilon = 0.1 lo hace mejor, obteniendo el valor optimo el 87% de las veces.

V. Implementación del Método ϵ-Decay:

La estrategia ϵ-greedy presentado en el punto 3, tiene una debilidad obvia al incluir continuamente un ruido (probabilidad), sin importar cuantos ejemplos se observen.
Como se señala, existe una forma de determinar la solución optima a explotar y reducir la probabilidad de exploración en cada paso.
Para ello se utiliza *ϵ* como una función del número de steps *n*.

$$\epsilon(n) = \frac{1}{1 + n\text{ß}}$$

Donde ß representa un factor escalar (tasa) y permitir al algoritmo un tiempo de exploración, 1 + nß permite que no sea definido como infinito.
Dado esto, se pueden definir las siguientes variables:

1) `eps_decay_bandi`: es la nueva clase del experimento, esta vez con ϵ-Decay.
2) `k` : es la cantdad de máquinas.
3) `eps` : es la probabilidad (ruído) epsilon del modelo.
4) `iters` : es el número de iteraciones.
5) `mu` : es la recompensa que se ajusta para cada máquina, por default es N (0,1).

Código se observa en Anexo A de este documento.

A. Método de ϵ-Decay:

Se efectuará un modelado utilizando el método de ϵ-Decay, se seleccionó el epsilon que obtuvo mayores recompensas en el modelado anterior, con las siguientes características:

1) `k` = 10 máquinas.
2) `iters` = 3000 iteraciones.
3) `episodes` = 20 experimentos.
4) `eps_1_rewards` = recompensas con epsilon = 0.1.
5) `eps_decay_rewards` = recompensas con modelo ϵ-Decay.

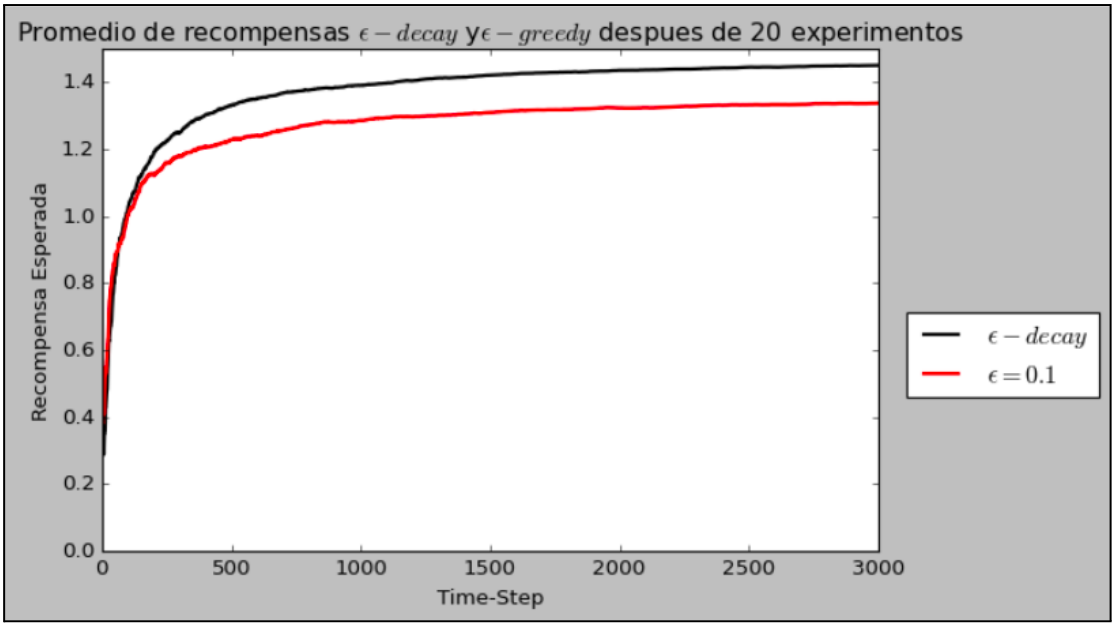

Figura Nº 5: Gráfico comaración recompensas e-greedy / e-decay.

Se puede observar que utilizar el algoritmo ϵ-Decay, permite optimizar aún el experimento.

VI. Conclusiones y comentarios finales:

En base a los resultados del experimento, en el primer modelo **ϵ-Greedy** podemos señalar que esta estrategia permite inicialmente maximizar la recompensa ya que jugará preferentemente con la máquina que ha ofrecido la mayor recompensa hasta ese momento.

Efectuado el modelamiento del método **ϵ-Decay**, podemos concluir que es más eficiente que el **método greedy** puesto que al seleccionar una máquina inicialmente al azar y, a medida que avanzan los experimentos, cada vez de una forma más ambiciosa permite obtener mayores recompensas. Esto debido a que, una vez que se conoce cual es la mejor máquina ya no es necesario **explorar** en busca del mejor.

Con lo anterior, hemos visto el impacto de balancear correctamente la **exploración y la explotación** de las acciones con un mayor valor estimado y el uso del conocimiento a priori para obtener unas estimaciones mejores y más rápidas de Q_a^* de cada acción.

Por último, cabe señalar que existe una diversidad de modelos para resolver el problema del Multi-Armed Bandit, entre los cuales existen:

1) UCB1.
2) Softmax.
3) Optimistic Inicial Values.
4) Test A/B.
5) Pursuit Algorithm.
6) CP-UCB.
7) UCB-V.
8) MOSS.
9) EXP3.

Los cuales presentan diferentes métodos para encontrar la optimalidad.

VII. Referencias:

[A]. Data Incubator (2016) Multi-Armed Bandits. [En línea]. Disponible en: https://www.thedataincubator.com/blog/2016/07/25/multi-armed-bandits-2/ [Acceso el 27 de junio de 2022].

[B]. Guinea, Álvaro [P.G. González Ana] (2016) Introducción al Aprendizaje de Refuerzo, Problema del Bandido multibrazo. Tesis para obtener el grado de Ingeniero Informático. Universidad Autónoma de Madrid [En línea]. Disponible en: https://repositorio.uam.es/bitstream/handle/10486/675554/Guinea_Julia_Alvaro_tfg.pdf?sequence=1&isAllowed=y [Acceso el 27 de junio de 2022].

[C]. Hubbs, Christian (2019) Multi-Armed Bandits and Reinforcement Learning. Towards Data Science. [En línea]. Disponible en: https://towardsdatascience.com/multi-armed-bandits-and-reinforcement-learning-dc9001dcb8da [Acceso el 27 de junio de 2022].

[D]. Rodríguez, Daniel (2021) Epsilon-Greedy para el Bandido Multibrazo (Multi-Armed Bandit). Analytics Lane. [En línea]. Disponible en: https://www.analyticslane.com/2021/02/26/epsilon-gready-para-el-bandido-multibrazo-multi-armed-bandit/ [Acceso el 27 de junio de 2022].

[E]. Rodríguez, Daniel (2021) Epsilon-Greedy con decaimiento para un problema Bandido Multibrazo (Multi-Armed Bandit). Analytics Lane. [En línea]. Disponible en: https://www.analyticslane.com/2021/03/05/epsilon-greedy-con-decaimiento-para-un-problema-bandido-multibrazo-multi-armed-bandit/ [Acceso el 27 de junio de 2022].

[F]. Wong, Anson (2017) Solving the Multi-Armed Bandit Problem. Towards Data Science. [En línea]. Disponible en: https://towardsdatascience.com/solving-the-multi-armed-bandit-problem-b72de40db97c [Acceso el 27 de junio de 2022].

Anexo “A”

Código de Python 3

IDE: Jupyterlab versión 3.1.7. (2021)
web site https://jupyter.org/install

```
# pip install modules
```

```
import modules
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
```

```
class eps_bandit:
    '''
    epsilon-greedy k-bandit problem

    Inputs
    =====================================================
    k: number of arms (int)
    eps: probability of random action 0 < eps < 1 (float)
    iters: number of steps (int)
    mu: set the average rewards for each of the k-arms.
        Set to "random" for the rewards to be selected from
        a normal distribution with mean = 0.
        Set to "sequence" for the means to be ordered from
        0 to k-1.
        Pass a list or array of length = k for user-defined
        values.
    '''

    def __init__(self, k, eps, iters, mu='random'):
        # Number of arms
        self.k = k
        # Search probability
        self.eps = eps
        # Number of iterations
        self.iters = iters
        # Step count
        self.n = 0
        # Step count for each arm
        self.k_n = np.zeros(k)
```

```
        # Total mean reward
        self.mean_reward = 0
        self.reward = np.zeros(iters)
        # Mean reward for each arm
        self.k_reward = np.zeros(k)

        if type(mu) == list or type(mu).__module__ == np.__name__:
            # User-defined averages
            self.mu = np.array(mu)
        elif mu == 'random':
            # Draw means from probability distribution
            self.mu = np.random.normal(0, 1, k)
        elif mu == 'sequence':
            # Increase the mean for each arm by one
            self.mu = np.linspace(0, k-1, k)

    def pull(self):
        # Generate random number
        p = np.random.rand()
        if self.eps == 0 and self.n == 0:
            a = np.random.choice(self.k)
        elif p < self.eps:
            # Randomly select an action
            a = np.random.choice(self.k)
        else:
            # Take greedy action
            a = np.argmax(self.k_reward)

        reward = np.random.normal(self.mu[a], 1)

        # Update counts
        self.n += 1
        self.k_n[a] += 1

        # Update total
        self.mean_reward = self.mean_reward + (
            reward - self.mean_reward) / self.n

        # Update results for a_k
        self.k_reward[a] = self.k_reward[a] + (
            reward - self.k_reward[a]) / self.k_n[a]

    def run(self):
        for i in range(self.iters):
            self.pull()
            self.reward[i] = self.mean_reward

    def reset(self):
        # Resets results while keeping settings
```

```
        self.n = 0
        self.k_n = np.zeros(k)
        self.mean_reward = 0
        self.reward = np.zeros(iters)
        self.k_reward = np.zeros(k)
```

```
k = 10
iters = 3000

eps_0_rewards = np.zeros(iters)
eps_001_rewards = np.zeros(iters)
eps_01_rewards = np.zeros(iters)
eps_1_rewards = np.zeros(iters)
eps_10_rewards = np.zeros(iters)

eps_0_selection = np.zeros(k)
eps_001_selection = np.zeros(k)
eps_01_selection = np.zeros(k)
eps_1_selection = np.zeros(k)
eps_10_selection = np.zeros(k)

episodes = 20
for i in range(episodes):
# Inicialización de las máquinas (bandits)
    eps_0 = eps_bandit(k, 0, iters)
    eps_001 = eps_bandit(k, 0.001, iters, eps_0.mu.copy())
    eps_01 = eps_bandit(k, 0.01, iters, eps_0.mu.copy())
    eps_1 = eps_bandit(k, 0.1, iters, eps_0.mu.copy())
    eps_10 = eps_bandit(k, 1.0, iters, eps_0.mu.copy())

# Comenzar Experimentos
    eps_0.run()
    eps_001.run()
    eps_01.run()
    eps_1.run()
    eps_10.run()

# Actualizar recompensas:
    eps_0_rewards = eps_0_rewards + (
        eps_0.reward - eps_0_rewards) / (i + 1)
    eps_001_rewards = eps_001_rewards + (
        eps_001.reward - eps_001_rewards) / (i + 1)
    eps_01_rewards = eps_01_rewards + (
        eps_01.reward - eps_01_rewards) / (i + 1)
    eps_1_rewards = eps_1_rewards + (
        eps_1.reward - eps_1_rewards) / (i + 1)
    eps_10_rewards = eps_10_rewards + (
        eps_10.reward - eps_10_rewards) / (i + 1)
```

```
# Promedio de acciones efectuadas por experimento
    eps_0_selection = eps_0_selection + (
        eps_0.k_n - eps_0_selection) / (i + 1)
    eps_001_selection = eps_001_selection + (
        eps_001.k_n - eps_001_selection) / (i + 1)
    eps_01_selection = eps_01_selection + (
        eps_01.k_n - eps_01_selection) / (i + 1)
    eps_1_selection = eps_1_selection + (
        eps_1.k_n - eps_1_selection) / (i + 1)
    eps_10_selection = eps_10_selection + (
        eps_10.k_n - eps_10_selection) / (i + 1)

plt.figure(figsize=(8,5))
plt.style.use("classic")
plt.plot(eps_0_rewards, label="$\epsilon=0$ (greedy)", linewidth=2.0, color='blue')
plt.plot(eps_001_rewards, label="$\epsilon=0.001$", linewidth=2.0, color='green')
plt.plot(eps_01_rewards, label="$\epsilon=0.01$", linewidth=2.0, color='orange')
plt.plot(eps_1_rewards, label="$\epsilon=0.1$", linewidth=2.0, color='red')
plt.plot(eps_10_rewards, label="$\epsilon=1.0$", linewidth=2.0, color='pink')
plt.ylim(-0.1,1.50)
plt.legend(bbox_to_anchor=(1.35, 0.5))
plt.xlabel("Time-Step")
plt.ylabel("Recompensa Esperada")
plt.title("Promedio de recompensas utilizando $\epsilon-greedy$ despues de " +
str(episodes)
    + " Experimentos")
plt.show()
```

```
k = 10
iters = 3000

eps_0_rewards = np.zeros(iters)
eps_001_rewards = np.zeros(iters)
eps_01_rewards = np.zeros(iters)
eps_1_rewards = np.zeros(iters)
eps_10_rewards = np.zeros(iters)

eps_0_selection = np.zeros(k)
eps_001_selection = np.zeros(k)
eps_01_selection = np.zeros(k)
eps_1_selection = np.zeros(k)
eps_10_selection = np.zeros(k)

episodes = 20
for i in range(episodes):
# Inicialización de las máquinas (bandits)
    eps_0 = eps_bandit(k, 0, iters, mu='sequence')
```

```
    eps_001 = eps_bandit(k, 0.001, iters, eps_0.mu.copy())
    eps_01 = eps_bandit(k, 0.01, iters, eps_0.mu.copy())
    eps_1 = eps_bandit(k, 0.1, iters, eps_0.mu.copy())
    eps_10 = eps_bandit(k, 1.0, iters, eps_0.mu.copy())

# Comenzar Experimentos
    eps_0.run()
    eps_001.run()
    eps_01.run()
    eps_1.run()
    eps_10.run()

# Actualizar recompensas:
    eps_0_rewards = eps_0_rewards + (
        eps_0.reward - eps_0_rewards) / (i + 1)
    eps_001_rewards = eps_001_rewards + (
        eps_001.reward - eps_001_rewards) / (i + 1)
    eps_01_rewards = eps_01_rewards + (
        eps_01.reward - eps_01_rewards) / (i + 1)
    eps_1_rewards = eps_1_rewards + (
        eps_1.reward - eps_1_rewards) / (i + 1)
    eps_10_rewards = eps_10_rewards + (
        eps_10.reward - eps_10_rewards) / (i + 1)

# Promedio de acciones efectuadas por experimento
    eps_0_selection = eps_0_selection + (
        eps_0.k_n - eps_0_selection) / (i + 1)
    eps_001_selection = eps_001_selection + (
        eps_001.k_n - eps_001_selection) / (i + 1)
    eps_01_selection = eps_01_selection + (
        eps_01.k_n - eps_01_selection) / (i + 1)
    eps_1_selection = eps_1_selection + (
        eps_1.k_n - eps_1_selection) / (i + 1)
    eps_10_selection = eps_10_selection + (
        eps_10.k_n - eps_10_selection) / (i + 1)

plt.figure(figsize=(8,5))
plt.style.use("classic")
plt.plot(eps_0_rewards, label="$\epsilon=0$ (greedy)", linewidth=2.0, color='blue')
plt.plot(eps_001_rewards, label="$\epsilon=0.001$", linewidth=2.0, color='green')
plt.plot(eps_01_rewards, label="$\epsilon=0.01$", linewidth=2.0, color='orange')
plt.plot(eps_1_rewards, label="$\epsilon=0.1$", linewidth=2.0, color='red')
plt.plot(eps_10_rewards, label="$\epsilon=1.0$", linewidth=2.0, color='pink')
plt.ylim(0,9)
plt.legend(bbox_to_anchor=(1.55, 0.5))
plt.xlabel("Time-Step")
plt.ylabel("Recompensa Esperada")
plt.title("Promedio de recompensas utilizando $\epsilon-greedy$ despues de " +
str(episodes)
```

```
    + " Experimentos")
plt.show()
```

```
bins = np.linspace(0, k-1, k)

plt.figure(figsize=(8,5))
plt.bar(bins, eps_0_selection,
        width = 0.20, color='blue',
        label="$\epsilon=0$")
plt.bar(bins+0.20, eps_001_selection,
        width=0.20, color='green',
        label="$\epsilon=0.001$")
plt.bar(bins+0.40, eps_01_selection,
        width=0.20, color='orange',
        label="$\epsilon=0.01$")
plt.bar(bins+0.60, eps_1_selection,
        width=0.20, color='red',
        label="$\epsilon=0.1$")
plt.bar(bins+0.80, eps_10_selection,
        width=0.20, color='pink',
        label="$\epsilon=1.0$")
plt.legend(bbox_to_anchor=(1.3, 0.5))
plt.xlim([0,k])
plt.title("Acciones efectuadas por cada algoritmo")
plt.xlabel("Acción")
plt.ylabel("Cantidad de acciones efectuadas")
plt.show()

opt_per = np.array([eps_0_selection, eps_001_selection, eps_01_selection,
                    eps_1_selection, eps_10_selection]) / iters * 100
df = pd.DataFrame(opt_per, index=['$\epsilon=0$', '$\epsilon=0.001$',
    '$\epsilon=0.01$', '$\epsilon=0.1$', '$\epsilon=1.0$'],
                  columns=["a = " + str(x) for x in range(0, k)])
print("Porcentaje de acciones seleccionadas:")
df
```

```
class eps_decay_bandit:
    '''
    epsilon-decay k-bandit problem

    Inputs
    =====================================================
    k: number of arms (int)
    iters: number of steps (int)
    mu: set the average rewards for each of the k-arms.
        Set to "random" for the rewards to be selected from
        a normal distribution with mean = 0.
```

```
    Set to "sequence" for the means to be ordered from
    0 to k-1.
    Pass a list or array of length = k for user-defined
    values.
'''

def __init__(self, k, iters, mu='random'):
    # Number of arms
    self.k = k
    # Number of iterations
    self.iters = iters
    # Step count
    self.n = 0
    # Step count for each arm
    self.k_n = np.zeros(k)
    # Total mean reward
    self.mean_reward = 0
    self.reward = np.zeros(iters)
    # Mean reward for each arm
    self.k_reward = np.zeros(k)

    if type(mu) == list or type(mu).__module__ == np.__name__:
        # User-defined averages
        self.mu = np.array(mu)
    elif mu == 'random':
        # Draw means from probability distribution
        self.mu = np.random.normal(0, 1, k)
    elif mu == 'sequence':
        # Increase the mean for each arm by one
        self.mu = np.linspace(0, k-1, k)

def pull(self):
    # Generate random number
    p = np.random.rand()
    if p < 1 / (1 + self.n / self.k):
        # Randomly select an action
        a = np.random.choice(self.k)
    else:
        # Take greedy action
        a = np.argmax(self.k_reward)

    reward = np.random.normal(self.mu[a], 1)

    # Update counts
    self.n += 1
    self.k_n[a] += 1

    # Update total
    self.mean_reward = self.mean_reward + (
```

```
            reward - self.mean_reward) / self.n

        # Update results for a_k
        self.k_reward[a] = self.k_reward[a] + (
            reward - self.k_reward[a]) / self.k_n[a]

    def run(self):
        for i in range(self.iters):
            self.pull()
            self.reward[i] = self.mean_reward

    def reset(self):
        # Resets results while keeping settings
        self.n = 0
        self.k_n = np.zeros(k)
        self.mean_reward = 0
        self.reward = np.zeros(iters)
        self.k_reward = np.zeros(k)
```

```
k = 10
iters = 3000
eps_decay_rewards = np.zeros(iters)
eps_1_rewards = np.zeros(iters)
episodes = 20
# Run experiments
for i in range(episodes):
    # Initialize bandits
    eps_decay = eps_decay_bandit(k, iters)
    eps_1 = eps_bandit(k, 0.1, iters, eps_decay.mu.copy())

    # Run experiments
    eps_decay.run()
    eps_1.run()

    # Update long-term averages
    eps_decay_rewards = eps_decay_rewards + (
        eps_decay.reward - eps_decay_rewards) / (i + 1)
    eps_1_rewards = eps_1_rewards + (
        eps_1.reward - eps_1_rewards) / (i + 1)

plt.figure(figsize=(8,5))
plt.plot(eps_decay_rewards, label="$\epsilon-decay$", linewidth=2.0, color='black')
plt.plot(eps_1_rewards, label="$\epsilon=0.1$", linewidth=2.0, color='red')
plt.legend(bbox_to_anchor=(1.3, 0.5))
plt.ylim(0,1.5)
plt.xlabel("Time-Step")
plt.ylabel("Recompensa Esperada")
```

```
plt.title("Promedio de recompensas $\epsilon-decay$ y" +
    "$\epsilon-greedy$ despues de "
    + str(episodes) + " experimentos")
plt.show()
```