Facultad de Ingeniería
Universidad de Concepción

Tarea 2: Machine Learning*

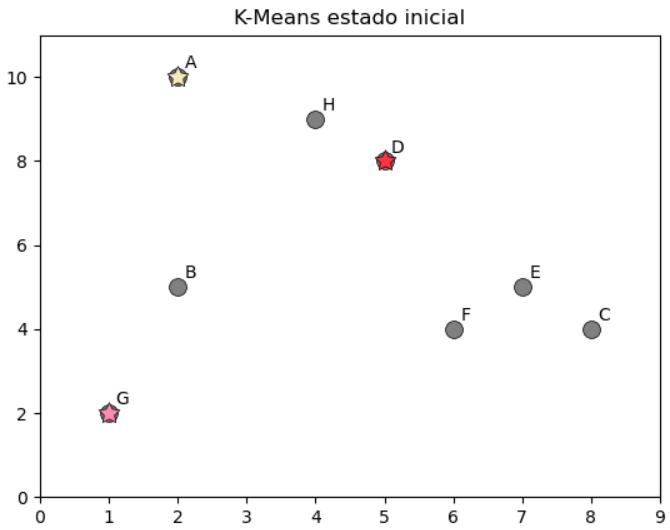
Claudia Cser †

June 22, 2025

1 K-means

Para resolver el problema, se implementó un programa en Python que ejecuta *K-means* con distancia Euclidiana. Este programa agrupa los 8 puntos dados en 3 clusters, utilizando como centroides iniciales los puntos A = $(2, 10)$, D = $(5, 8)$ y G = $(1, 2)$, como se indica en el enunciado.

En el estado inicial, aún no hay asignación de puntos a clusters. Solo se ubican los tres centroides iniciales, representados como estrellas $\star$ del color correspondiente a su futuro cluster: $(2, 10)$ en amarillo, $(5, 8)$ en rojo y $(1, 2)$ en rosado.



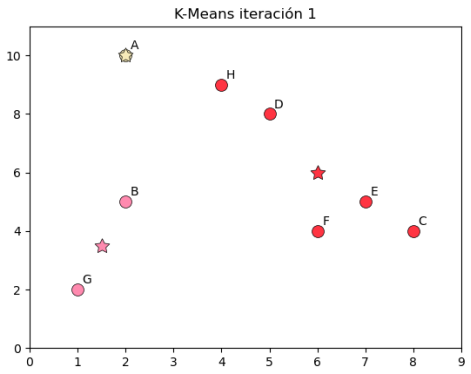
*Universidad de Concepción; Inteligencia Artificial. Docente: Julio Godoy.
†Matrícula: 2021456848.

1.1 Iteración 1:

Cada punto fue asignado al centroide más cercano según la distancia euclidiana y cada centroide recalculado de acuerdo a la asignación. El resultado fue:

- `Clusters:`
  - Amarillo: A
  - Rojo: C, D, E, F, H
  - Rosado: B, G

- `Centroides:`
  - Amarillo: $(2, 10)$
  - Rojo: $(6, 6)$
  - Rosado: $(1.5, 3.5)$

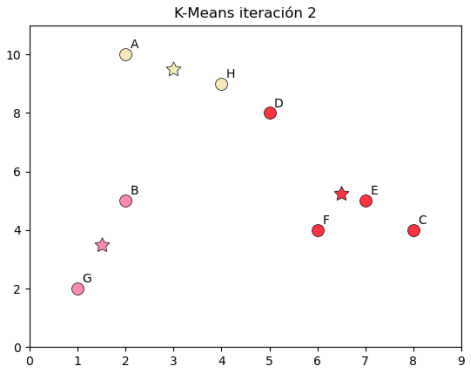


1.2 Iteración 2:

En esta iteración, solo el punto H cambió de cluster, pasando del rojo al amarillo y ajustando sus centroides. El centroide rosado se mantuvo igual, indicando que posiblemente está estabilizado.

- `Clusters:`
  - Amarillo: A, H
  - Rojo: C, D, E, F
  - Rosado: B, G

- `Centroides:`
  Amarillo: $(3, 9.5)$
  Rojo: $(6.5, 5.25)$
  Rosado: $(1.5, 3.5)$

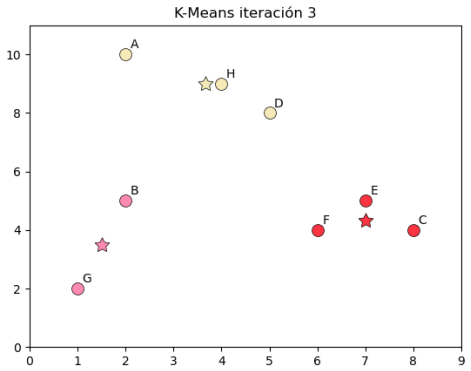


1.3 Iteración 3:

En esta última iteración, el único cambio fue que el punto D pasó del cluster rojo al amarillo y sus centroides se reacomodaron. Este cambio ya refleja una distribución más natural y clara de los clusters.

- `Clusters:`
  - Amarillo: A, D, H
  - Rojo: C, E, F
  - Rosado: B, G

- `Centroides:`
  - Amarillo: $(3.67, 9)$
  - Rojo: $(7, 4.33)$
  - Rosado: $(1.5, 3.5)$



2 DBSCAN

Para desarrollar este item, se implementó el algoritmo *DBSCAN* en Python. Este algoritmo permite identificar clusters basados en densidad, sin necesidad de fijar la cantidad de agrupaciones como en K-means.

2.1 Caso 1: $eps = 2$, $minPoints = 2$

En este caso, el valor de eps define un radio bastante pequeño, lo que significa que solo los puntos con vecinos muy cercanos serán considerados *core points*. Se obtuvieron los siguientes resultados:

`Clusters:`
- Amarillo: C, E, F
- Rojo: D, H

`Outliers:`
- A, B, G

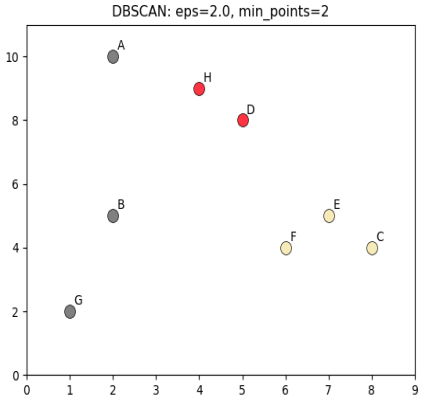
Esto se debe a que los puntos A, B y G no tienen suficientes vecinos dentro del radio eps = 2 como para ser considerados puntos centrales o frontera. Solo los subconjuntos (D, H) y (C, E, F) tienen suficiente densidad para quedar agrupados.

2.2 Caso 2: $eps = \sqrt{10}$, $minPoints = 2$

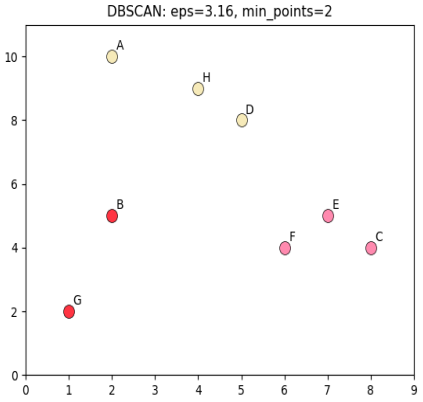
Al aumentar el valor de eps, el algoritmo encuentra más vecinos dentro del radio, lo cual todos los puntos tienen suficientes vecinos para ser considerados core points y cambia significativamente la asignación:

`Clusters:`
- Amarillo: A, D, H
- Rojo: B, G
- Rosado: C, E, F

Así se muestra cómo el parámetro eps influye directamente en la sensibilidad del algoritmo: un valor pequeño detecta agrupaciones más densas y puede dejar puntos como ruido, mientras que un valor mayor tiende a agrupar más puntos y eliminar outliers.



(a) Caso 1



(b) Caso 2

3 Aprendizaje por refuerzo

Para esta sección se trabajó sobre el archivo `tutorial.txt` entregado como base. Cabe mencionar que todo el código desarrollado para este item y los anteriores se encuentra junto con las imágenes y resultados en el archivo `.zip` entregado y en GitHub.

Se imprimen pares `[episodio, recompensa acumulada]` al finalizar cada episodio dentro de la función `main`, en el archivo `Rewards.txt`, que permite visualizar la evolución del agente y se utilizará luego para graficar las curvas de aprendizaje.

```
int main(int argc, char* argv[]) {
    srand(time(NULL));

    reward_output.open("Rewards.txt");
    ...
    for(int i = 0; i < num_episodes; i++) {
        reward_output << "\n\n Episode " << i;
        ...
        finalrw[i] = cum_reward;
        reward_output << " Total reward obtained: " << finalrw[i] << "\n";
    }
    ...
```

Respecto a las estrategias de selección de acciones, se implementaron las politicas *greedy* y su extensión ε-*greedy*, utilizando un valor inicial de $\varepsilon = 0.99$, con el objetivo de favorecer la exploración en las primeras etapas de aprendizaje, cada vez que el agente toma una acción aleatoria el valor de `exp_rate` (que representa ε) se reduce en $\delta = 1^{-4}$. Esto permite que el agente comience explorando intensamente, pero que con el tiempo y la experiencia acumulada, disminuya progresivamente la exploración, favoreciendo la explotación de lo aprendido.

Esta política se encuentra definida de la siguiente forma en la función `action_selection`:

```
int action_selection() {
    pair<float, int> highest = {Qvalues[x_pos][y_pos][0], 0};
    for(int i = 1; i < 4; ++i) {
        highest = max(highest, {Qvalues[x_pos][y_pos][i], i});
    }

    if(action_sel == 2) { // epsilon-greedy
        float ran = (rand()%100)/100.0;
        if(ran < 1-exp_rate) {
            return highest.second;
        }
        else {
            cout << exp_rate << "\n";
            exp_rate -= DELTA;
            return rand()%4;
        }
```

```
17      }
18      else { // greedy
19          return highest.second;
20      }
21  }
```

En cada episodio el agente identifica primero la acción con el mayor valor de Q en el estado actual. Luego, si la política es *greedy*, se escoge la acción con este valor. Si es ε-*greedy*, se genera un número flotante entre 0 y 1: `ran`, si es menor que $1 - \varepsilon$, el agente escoge la mejor acción conocida. En caso contrario, se selecciona una acción aleatoria.

La implementación de acciones estocásticas se hizo en la función `move`. Estas permiten que el agente no siempre se mueva en la dirección seleccionada, con un 80% de probabilidad ejecuta la acción tal como fue elegida. Sin embargo, existe un 10% de probabilidad de que el movimiento sea desviado hacia la derecha (equivalente a rotar la acción en +1), y otro 10% de que se desvíe a la izquierda (rotación -1).

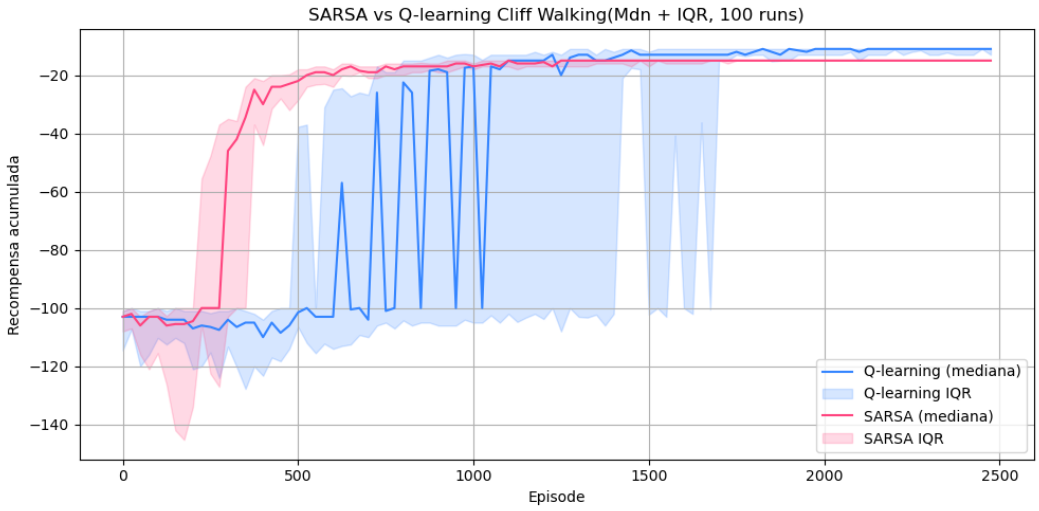
Esto se simula generando un número aleatorio entre 0 y 9: si el resultado es 0, se aplica el desvío hacia la derecha, si es 1, hacia la izquierda, y en cualquier otro caso, se mantiene la acción original.

```
1   void move(int action) {
2       ...
3       if(stochastic_actions) {
4           int choice = rand()%10;
5           if(choice == 0) { // Moverse a la derecha
6               cout << ">\n";
7               action = (action+1)%4;
8           }
9           if(choice == 1) { // Moverse a la izquierda
10              cout << "<\n";
11              action = (action - 1 + 4)%4;
12          }
13      }
14
15      if(action == 0) { // Up
16      ...
```

3.1 Resultados

Para esta sección, se evaluaron los algoritmos *Q-learning* y *SARSA* en el ambiente 1(Small Grid) y 2 (Cliff Walking) utilizando acciones deterministas y estocásticas. Para Cliff Walking determinista, el entrenamiento se realizó a lo largo de 2500 episodios, con parámetros fijos: factor de descuento (`disc_factor`) de 0.9, tasa de aprendizaje (`learn_rate`) de 0.1 y una política ε-`greedy` con exploración inicial de $\varepsilon = 0.99$ que disminuye en 1^{-4} cada vez que el agente realiza una acción aleatoria, tal como se describió anteriormente.

Para el análisis, se escribió un script en Python que ejecuta cada algoritmo 100 veces independientes. Para cada episodio, se utilizó la *mediana* como medida central y el *rango intercuartil* (IQR) como medida de dispersión, lo que permite observar de forma más clara el comportamiento típico del agente, evitando que valores atípicos distorsionen la interpretación. Con estos datos, se generó un gráfico de la curva de aprendizaje, actualizando la curva cada 25 episodios.



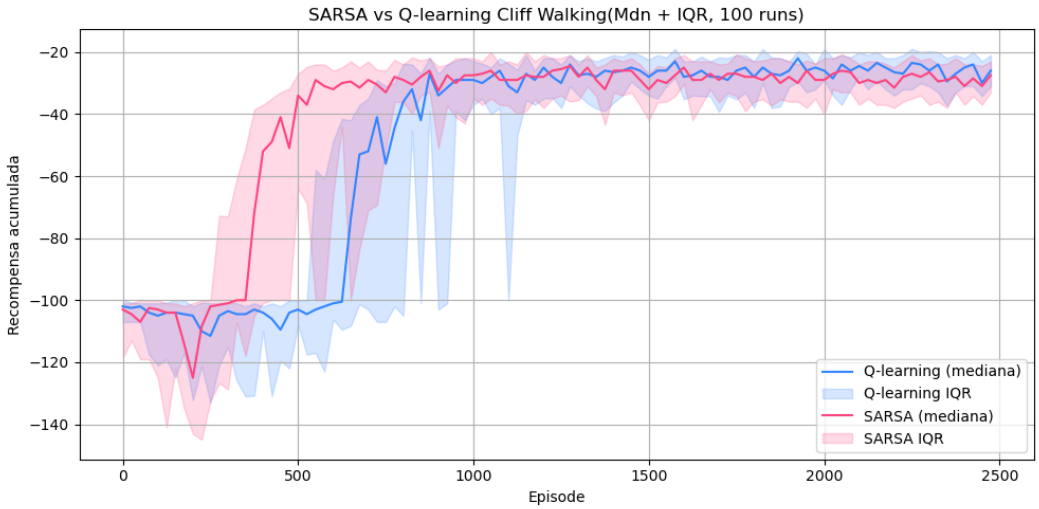
Clif Walking en ambiente determinista.

Los resultados coinciden con lo visto en clases y en la literatura. Por un lado, SARSA converge más rápidamente hacia una política relativamente segura y estable, evitando el borde del acantilado. Esto se nota en su menor variabilidad desde etapas tempranas del entrenamiento. Por otro lado, Q-learning alcanza "mejores" resultados en el largo plazo, encontrando caminos más cortos pero también más riesgosos, lo que se traduce en mayor variabilidad en los primeros episodios. De hecho, la mediana de Q-learning es mucho más alta que su promedio, lo que se debe principalmente a un número reducido de episodios

fallidos con grandes castigos, más que a un mal desempeño generalizado.

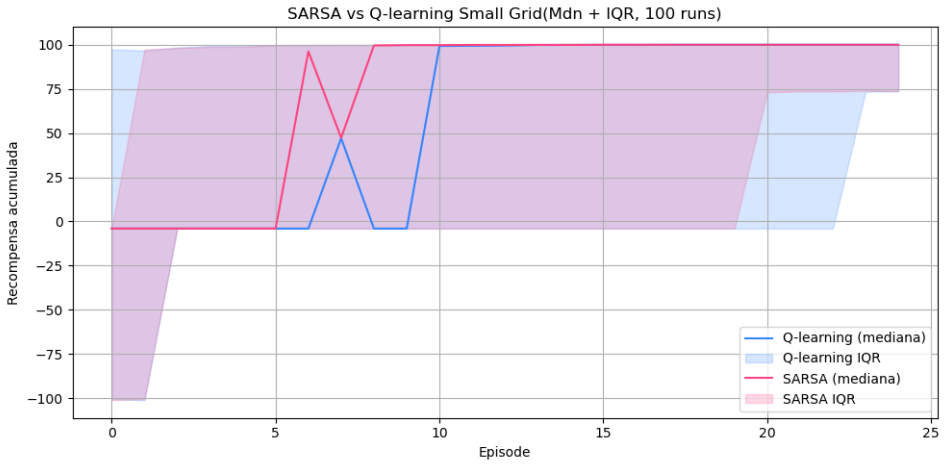
También se aprecia cómo la variabilidad disminuye significativamente con el paso de los episodios, especialmente en Q-learning. Esto es por la reducción gradual del valor de ε, que limita la exploración aleatoria provocando que el comportamiento del agente se estabilice. Al principio, es muy sensible a decisiones aleatorias que pueden hacer que caiga por el borde del acantilado, pero una vez que la exploración disminuye, esas desviaciones dejan de ocurrir con frecuencia, reflejando mejor el conocimiento adquirido.

Cuando se aplican decisiones estocásticas en este caso, se puede apreciar como el aprendizaje de Q-learning presenta una curva más estable en los primeros episodios. Mientras que SARSA requiere algo mas de tiempo para adaptarse que en un entorno determinista. Sin embargo, a medida que avanzan los episodios, cuando las curvas se estabilizaban en el ambiente determinista, aquí se mantienen turbulentas. Aún así se mantiene el resultado de que Q-learning obtiene finalmente menor penalización que SARSA.



Clif Walking en ambiente estocástico.

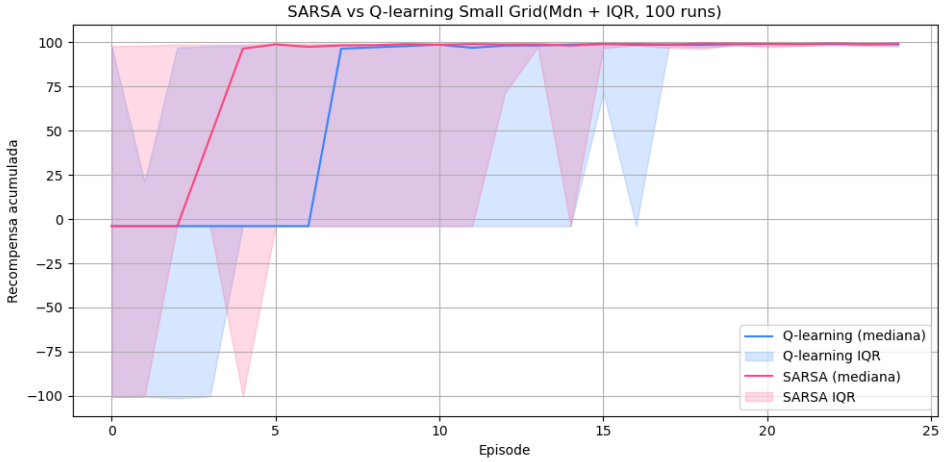
Para el primer ambiente, Small Grid, al ser un entorno más compacto y simple que Cliff Walking, se ajustaron los parámetros para reflejar la escala del problema. Se utilizó un total de 25 episodios, tasa de exploración de 0.5, delta de 1^{-1} y a lo más 100 pasos por episodio. Al igual que en el caso anterior, se realizaron 100 ejecuciones independientes por algoritmo, calculando la mediana y rangos intercuartil.



Small Grid en ambiente determinista.

En su versión determinista, los resultados muestran una dinámica similar al caso anterior, SARSA aprende más rápidamente, pero esta vez ambos convergen mucho más rápidamente, y dado que en este ambiente el camino seguro tiene distancia mínima, ambos algoritmos llegan al mismo valor.

A continuación se muestran los resultados para el mismo caso, pero en la versión estocástica, donde se utilizaron las mismas especificaciones de parámetros:



Small Grid en ambiente estocástico.

Al ser un ambiente pequeño, los resultados son muy similares. Algunas variaciones son que el agente aprende ligeramente más rápido y se aprecian algunas turbulencias en la curva estabilizada.

4 Referencias

- Russel, S.; Norvig, P. (2003) *Artificial Intelligence: A Modern Approach*, 2nd edition. Prentice-Hall.
- Callan, R. (2003) *Artificial Intelligence*. Palgrave Macmillan.
- Alpaydin, E. (2010) *Introduction to Machine Learning*, 2nd edition. MIT Press.