



UNIVERSIDAD DE GRANADA

PROGRAMACIÓN TÉCNICA Y CIENTÍFICA
GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 2

ROBÓTICA

Autor

Vladislav Nikolov Vasilev

Rama

Computación y Sistemas Inteligentes



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2019-2020

Índice

1. Objetivo de la práctica	2
2. Captura de los datos	2
3. Agrupación de los puntos en clusters	4
4. Extracción de características	6
5. Entrenamiento de un clasificador SVM	7
6. Predicción y corrección de los datos	12
7. Apartado opcional: detección de objetos	21

1. Objetivo de la práctica

El principal objetivo de esta práctica es hacer que un robot sea capaz de diferenciar las piernas de las personas de objetos que no son piernas, como podrían ser por ejemplo cilindros. Para ello, se ha propuesto utilizar un simulador donde se dispondrá de un robot con un sensor láser y una cámara. Se tomarán primeramente datos y después de procesarlos se entrenará un modelo de *machine learning* que clasifique la información como “pierna” y “no pierna”. El modelo que se propone utilizar es un **SVM**, y hay que escoger la mejor variante de dicho modelo comparando una serie de *kernels* e hiperparámetros. Una vez que se tenga el modelo definitivo, se utilizará una escena de test para ver qué tan bien lo hace con datos nuevos. Finalmente se creará un archivo en formato **HTML** que contendrá una tabla donde se mostrarán fotos de los objetos detectados, además de la distancia a la que están del robot y de la clase real y de la predicha.

En esta memoria se expondrá que *scripts* se han utilizado, qué funcionalidad tiene cada uno y se describirá brevemente lo que se ha ido haciendo.

2. Captura de los datos

Lo primero que se ha hecho ha sido capturar los datos. Para hacerlo, se ha creado un *script* llamado **capturar.py**, cuyo funcionamiento puede resumirse en que pide al usuario el nombre del archivo de salida, el directorio donde quiere que se guarde el fichero, el número de ciclos que se quiera leer y cuánto tiempo se tiene que esperar entre lectura y lectura. Una vez hecho esto, se establece conexión con el servidor de V-REP activo y se crea el directorio de salida y se cambia el entorno de trabajo a este directorio. Una vez hecho esto, se toman los datos durante tantos ciclos como se ha especificado anteriormente, haciendo una pausa entre cada captura en la cantidad de segundos especificada. En cada lectura se van guardando los datos en el fichero de salida. Además, se toma una captura de la cámara en la primera y en la última toma de datos. Finalmente, se finaliza la conexión.

Para modularizar el código, el *script* se ha estructurado en funciones. Estas funciones se describen a continuación:

- **establecerConexion**: Establece la conexión con el servidor de V-REP activo y devuelve un ID del cliente. En caso de que no se pueda conectar, se finaliza la ejecución del *script*.
- **obtenerCamaraHandler**: Obtiene un *handler* de la cámara e inicializa la cámara y el sensor láser. Devuelve la referencia al *handler* de la cámara para

usos futuros.

- **init_entorno:** Combina las dos funciones anteriores en una y devuelve tanto el ID del cliente como el *handler* de la cámara.
- **procesar_ciclo:** Esta función hace una captura del sensor laser y obtiene información sobre los puntos detectados en los ejes X e Y. Una vez hecho esto, crea un diccionario que contiene información sobre la iteración actual y los puntos detectados en los dos ejes anteriores, con el mismo formato que el que se puede ver en los ficheros de ejemplo.
- **capturar_guardar_imagen:** Esta función hace una captura de lo que ve la cámara de la misma forma que se hacía en el código de ejemplo. Guarda el resultado en un fichero con un nombre específico.
- **stop_simulacionConexion:** Con esta función se detiene la simulación que se está haciendo y se cierra la conexión con el servidor.

Para capturar los datos, se han creado 4 escenas: una con una persona de pie, una con una persona sentada, una con un par de cilindros más pequeños que las piernas de una persona y otra con un par de cilindros más grandes que las piernas de una persona. Para cada caso se han tomado capturas de cerca, a distancia media y de lejos. Este proceso se puede ver a continuación:

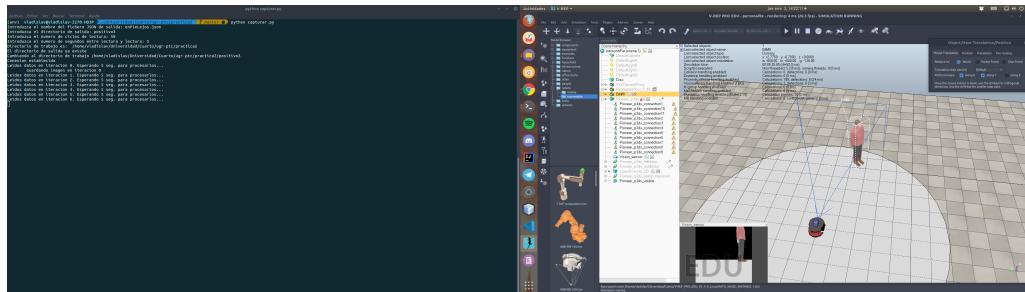


Figura 1: Captura de datos con persona de pie.

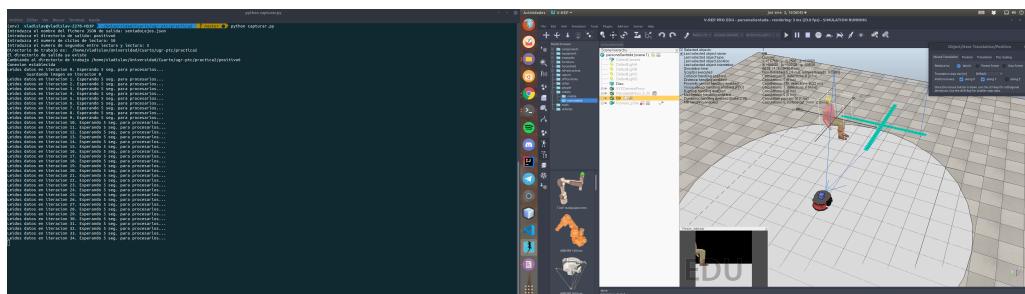


Figura 2: Captura de datos con persona sentada.

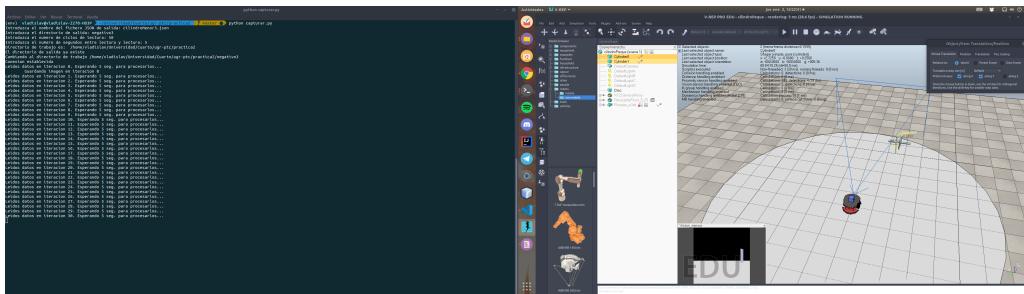


Figura 3: Captura de datos con par de cilindros pequeños.

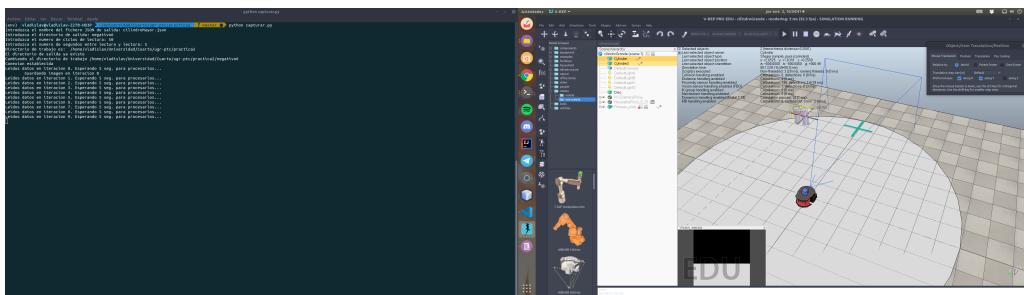


Figura 4: Captura de datos con par de cilindros grandes.

Como se puede ver, también se ha puesto un círculo en el suelo de radio 3.5, el cuál sirve de guía a la hora de tomar las capturas de los datos, para no salirse del radio especificado.

3. Agrupación de los puntos en clusters

Una vez que se han tomado los datos, hace falta agrupar los puntos en clusters. Para ello se ha creado un *script* llamado **agrupar.py** el cuál se encarga precisamente de eso.

El funcionamiento general se resumen en que toma todos los directorios de ejemplos positivos y de ejemplos negativos y crea clusters a partir de la información de cada tipo de directorio, según el formato que se ha especificado. Finalmente, guarda los ejemplos negativos en un fichero en formato **JSON** y los positivos en otro, teniendo por tanto la información separada.

Para crear los clusters es necesario establecer el número mínimo de puntos que lo pueden formar, el máximo y la distancia que puede haber, como máximo, entre dos puntos consecutivos. En un primer instante había establecido que el

número mínimo de puntos fuese 3, el **número máximo de puntos** fuese 25 y la **distancia máxima entre dos puntos consecutivos** fuese 0.03. Sin embargo, estos valores se modificarán más adelante, aunque se comentará en la sección correspondiente.

De nuevo, tal y como se hizo ante, se ha estructurado el fichero en funciones, las cuáles se pueden ver a continuación:

- **obtener_cluster_datos**: Esta función se encarga de extraer un único cluster a partir de un conjunto de puntos. Recorre el *array* de puntos proporcionado, el cuál está compuesto por coordenadas (x, y) , y va añadiendo los puntos consecutivos a una lista, siempre y cuando la distancia de un punto al siguiente no supere el umbral, haya menos puntos que el máximo permitido en la lista y no se supere la longitud del *array* de entrada (ya que se está recorriendo con índices).
- **procesar_clusters_muestra**: Esta función procesa una muestra de datos, generando los clusters que se pueden encontrar en el conjunto de puntos. Utiliza la función anterior para determinar los clusters uno por uno y al final los junta en una única lista. Se acepta un cluster siempre y cuando el número de puntos supere al mínimo requerido. La función está pensada para que pueda recibir la información como un diccionario de Python, de forma que se pueda reutilizar más adelante. Por tanto, si se le quiere pasar información que está en formato JSON, se tiene que cargar antes.
- **procesar_clusters_fichero**: Con esta función se pueden procesar los clusters de un fichero determinado. Hace uso de la función anterior, y junta los clusters de cada muestra en una única lista.
- **procesar_clusters_directorios**: Con esta función se pueden obtener los clusters de todos los archivos de un conjunto de directorios del mismo tipo. Para ello, se utiliza la función anterior, la cuál procesa cada fichero de forma individual. Se devuelve una lista con los clusters encontrados para todos los directorios del mismo tipo.
- **generar_informacion_cluster**: Esta función permite generar la información de salida de los clusters, en el formato especificado. Recibe la lista de clusters y construye una lista con diccionarios con el mismo formato.
- **guardar_clusters**: Esta función permite guardar los clusters en un archivo con un nombre determinado. Para generar la información, utiliza la función anterior. Después, la transforma a formato JSON y la escribe en el fichero de salida.

4. Extracción de características

Una vez que se han agrupado los puntos en clusters, se pueden extraer características, las cuáles serán útiles a la hora de entrenar el clasificador. Dichas características son el **perímetro**, la **anchura** y la **profundidad** del cluster. Para conseguir dicha tarea se ha creado un *script* llamado **características.py**. Este programa lee los ficheros creados anteriormente y genera las características para cada uno, guardando la información de forma separada tal y como se hacia antes. Los datos tienen el formato de salida especificado en el enunciado. Una vez hecho esto, carga los dos ficheros que acaba de generar y crea un archivo en formato CSV donde se juntan los datos de las dos clases, escribiendo primero los ejemplos de “no pierna” y luego los de “pierna”.

El fichero se ha estructurado en las siguientes funciones:

- **calcular_perímetro**: Función con la que se calcula el perímetro para un cluster determinado.
- **calcular_anchura**: Función con la que se calcula la anchura para un cluster determinado.
- **calcular_profundidad**: Función que calcula la profundidad de un cluster determinado. Para hacerlo, se utiliza el producto vectorial, ya que simplifica bastante los cálculos. Se hace dicha operación para cada punto del cluster excepto para el primero y el último y se escoge el resultado más grande (es decir, la mayor distancia).
- **calcular_características**: Función con la que se calculan las tres características anteriormente mencionadas. Utiliza las tres funciones anteriores.
- **generar_características_clusters_muestra**: Esta función se utiliza para generar las características de un conjunto de clusters de una muestra determinada. No se usa directamente en el programa, pero se deja la función para usos futuros, los cuales se comentarán más adelante. Utiliza la función anterior para calcular las características.
- **generar_características_clusters**: Esta función se utiliza para generar las características de un conjunto de clusters a partir de la información leída de un fichero. Utiliza la función para calcular características anteriormente mencionada. Guarda los resultados en un archivo con formato JSON.
- **escribir_clase**: Esta función permite escribir los datos de uno de los ficheros generados con la función anterior a un fichero con formato CSV.

- **generar_dataset**: La función permite generar el conjunto de datos que se utilizará para entrenar el clasificador. Utiliza la función anterior para escribir en el mismo fichero los ejemplos negativos primero y después los positivos.

5. Entrenamiento de un clasificador SVM

Una vez que se han generado las características y se han guardado en un fichero, podemos entrenar un modelo de *machine learning* para poder detectar piernas. Tal y como se dijo anteriormente, se va a entrenar un **SVM**, y se van a probar una serie de *kernels* distintos para ver cuál se adapta mejor. Cuando se conozca la mejor opción, se ajustarán los hiperparámetros para ver cuáles son los mejores valores. Los *kernels* que vamos a probar son uno lineal, uno polinómico de grado 2, uno de grado 3, uno de grado 4 y un *kernel* de base radial, también conocido como **rbf**.

El proceso general para determinar el mejor modelo y entrenarlo va a ser el siguiente:

1. **Dividir el dataset anteriormente generado en dos conjuntos: uno de de entrenamiento y uno de validación.** Entrenaremos nuestro modelo con los datos de entrenamiento y estudiaremos su comportamiento con los datos de validación.
2. **Hacer un primer estudio del comportamiento de los distintos modelos utilizando validación cruzada.** De esta forma, se dividirá el conjunto de entrenamiento en n particiones (muy importante que se conserve la proporcionalidad de las clases). Se entrenará el modelo con $n - 1$ particiones y la restante se utilizará para validar. Este proceso se repite n veces, utilizando todas las particiones para validar. De esta forma, obtendremos n resultados de *accuracy*, y al promediar tendremos una primera aproximación de los resultados que se podrían obtener con el conjunto de validación real, o lo que es lo mismo, al intentar predecir datos que nunca antes había visto. En este caso, se utilizarán 5 particiones, ya que se estima que es una cantidad razonable para la cantidad de datos de la que se dispone.
3. **Ajustar los modelos con el conjunto de entrenamiento y validar los resultados con el conjunto de validación.** De esta forma veremos qué resultados se obtienen con el conjunto de validación que tenemos, para ver cómo de bien funcionan y si los resultados se aproximan a lo que habíamos obtenido con la validación cruzada. Además, de esta forma podremos obtener otras métricas aparte de la *accuracy*, como por ejemplo la matriz de confusión, la cual nos permitirá ver cómo se comporta el modelo a la hora de predecir los datos de cada clase (cuántos predice bien y cuántos no), la precisión (la

capacidad de no clasificar ejemplos positivos como negativos) y el *recall* (la capacidad para encontrar los ejemplos verdaderos de cada clase).

4. **Escoger el mejor modelo y ajustar los hiperparámetros.** Para escoger dicho modelo, vamos a fijarnos en los resultados que se han obtenido en los dos pasos anteriores. Aquel modelo que tenga una mejor *accuracy* y unos mejores resultados en el resto de métricas podrá ser considerado como el mejor. En cuanto al ajuste de hiperparámetros, en este caso vamos a ajustar el parámetro C del modelo, el cuál mide cuánto se penalizan los puntos que caen dentro del margen del modelo (recordemos que SVM intenta encontrar el mejor separador, dejando la mayor cantidad de margen posible a cada lado). En un principio probaremos con valores pequeños, como por ejemplo 0.01, 0.1, 1, 2, 3, 4 y 5, hasta valores algo más grande, como por ejemplo 10 (es decir, de menos penalización a más).
5. **Ver cómo se comporta el modelo ajustado utilizando para ello el conjunto de validación.** Una vez que ya hayamos determinado el mejor modelo con los mejores hiperparámetros, vamos a utilizar el conjunto de validación para ver qué resultados obtenemos en esta ocasión. Utilizaremos las mismas métricas, y veremos si ha habido una mejora o no respecto a lo obtenido inicialmente.
6. **Reentrenar el modelo anterior con todos los datos que dispongamos y guardarlo para usos futuros, en caso de que sea mejor.** De esta forma, utilizamos todos los datos de los que disponemos para intentar conseguir un modelo aun mejor que el que teníamos anteriormente. Cuantos más datos de entrenamiento tengamos, en general, mejores van a ser los resultados obtenidos posteriormente.

Para hacer esto, se ha utilizado un *script* llamado **clasificarSVM.py**. En este fichero se pueden ver reflejados todos los pasos anteriores. Además de eso, se pueden visualizar los datos, de forma que se tenga algo más de información sobre el problema.

Antes de ver y estudiar los resultados, vamos a ver qué funciones tiene el archivo:

- **visualizar_datos:** Con esta función se puede crear un gráfico 3D que permita visualizar los datos del problema.
- **evaluar_modelo:** Esta función permite evaluar un modelo determinado utilizando para ello validación cruzada.
- **imprimir_informe:** La función permite imprimir por pantalla información sobre la *accuracy*, la matriz de confusión, la precisión y el *recall*.

- **mostrar_informe_validacion:** Función que entrena el modelo con los datos de entrenamiento y valida con los validación. Muestra los resultados utilizando la función anterior. Es importante destacar que, antes de entrenar el modelo, se hace una copia de este, de manera que el modelo original no se vea modificado por si se utiliza en el futuro.

Una vez que hemos explicado las funciones, vamos a visualizar los datos que tenemos:

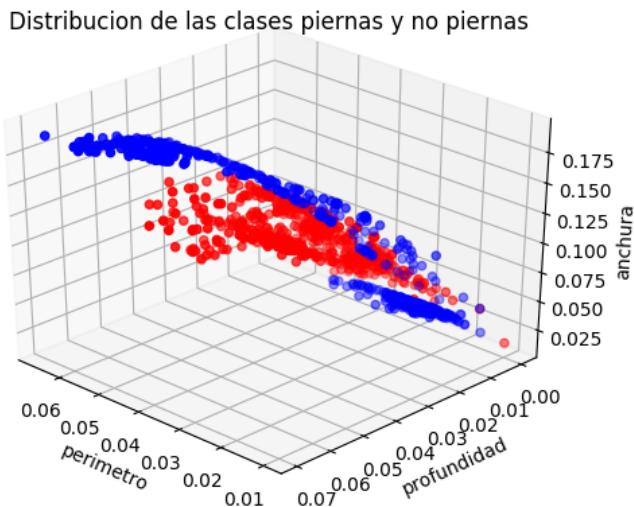


Figura 5: Distribución de las clases. En azul ejemplos de la clase “no pierna”, en rojo de la clase “pierna”.

Podemos ver que, en general, las regiones que separan a los puntos están bastante delimitadas. Vemos que también hay algunos puntos que son de la clase “pierna” que están mezclados con los de la otra clase. Por tanto, estos puntos se podrían considerar como *outliers*, y es muy posible que el clasificador no sea capaz de clasificarlos correctamente, ya que son ruido.

Una vez que hemos comentado esto, vamos a ver qué resultados hemos obtenido al ejecutar el programa. También comentaremos qué decisiones hemos tomado y el porqué de ellas:

```
----- Validacion de los modelos -----
Modelo evaluado: SVM Kernel Lineal Accuracy media en 5-fold CV: 0.5108
```

Modelo evaluado: SVM Kernel Polinomico Grad=2 Accuracy media en 5-fold CV: 0.6538
 Modelo evaluado: SVM Kernel Polinomico Grad=3 Accuracy media en 5-fold CV: 0.6924
 Modelo evaluado: SVM Kernel Polinomico Grad=4 Accuracy media en 5-fold CV: 0.6913
 Modelo evaluado: SVM Kernel RBF Accuracy media en 5-fold CV: 0.8911

```
----- Resultados de entrenamiento de los modelos -----
----- SVM Kernel Lineal -----
Acc_val: (TP+TN)/(T+P) 0.4751
Matriz de confusión Filas: verdad Columnas: predicción
[[ 0 116]
 [ 0 105]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score  support
          0         0.00     0.00     0.00      116
          1         0.48     1.00     0.64      105

accuracy           0.48      221
macro avg         0.24     0.50     0.32      221
weighted avg      0.23     0.48     0.31      221

----- SVM Kernel Polinomico Grad=2 -----
Acc_val: (TP+TN)/(T+P) 0.5837
Matriz de confusión Filas: verdad Columnas: predicción
[[43 73]
 [19 86]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score  support
          0         0.69     0.37     0.48      116
          1         0.54     0.82     0.65      105

accuracy           0.58      221
macro avg         0.62     0.59     0.57      221
weighted avg      0.62     0.58     0.56      221

----- SVM Kernel Polinomico Grad=3 -----
Acc_val: (TP+TN)/(T+P) 0.6154
Matriz de confusión Filas: verdad Columnas: predicción
[[48 68]
 [17 88]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score  support
          0         0.74     0.41     0.53      116
          1         0.56     0.84     0.67      105

accuracy           0.62      221
macro avg         0.65     0.63     0.60      221
weighted avg      0.66     0.62     0.60      221

----- SVM Kernel Polinomico Grad=4 -----
Acc_val: (TP+TN)/(T+P) 0.6290
Matriz de confusión Filas: verdad Columnas: predicción
[[48 68]
 [14 91]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score  support
```

```

          0      0.77      0.41      0.54      116
          1      0.57      0.87      0.69      105

accuracy           0.63      221
macro avg         0.67      0.64      0.61      221
weighted avg      0.68      0.63      0.61      221

----- SVM Kernel RBF -----
Acc_val: (TP+TN)/(T+P)  0.9005
Matriz de confusión Filas: verdad Columnas: predicción
[[ 97  19]
 [  3 102]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score   support
          0       0.97     0.84     0.90      116
          1       0.84     0.97     0.90      105

accuracy           0.90      221
macro avg         0.91     0.90     0.90      221
weighted avg      0.91     0.90     0.90      221

----- Mejor estimador -----
SVC(C=10, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
     decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
     max_iter=-1, probability=False, random_state=1, shrinking=True, tol=0.001,
     verbose=False)

----- Resultados finales -----
Acc_val: (TP+TN)/(T+P)  0.9683
Matriz de confusión Filas: verdad Columnas: predicción
[[110   6]
 [  1 104]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score   support
          0       0.99     0.95     0.97      116
          1       0.95     0.99     0.97      105

accuracy           0.97      221
macro avg         0.97     0.97     0.97      221
weighted avg      0.97     0.97     0.97      221

```

Lo primero que vemos son los resultados de validación. Ahí vemos que el peor modelo es el que utiliza el *kernel* lineal. Tiene una *accuracy* a muy próxima al 0.5, por tanto se podría casi considerar que clasifica de forma aleatoria. Los *kernels* polinómicos sí que son algo mejores. Parece que entre el grado 3 y el 4 no hay mucha diferencia, ya que los resultados obtenidos son muy parecidos. El de grado 2 se queda bastante por detrás, siendo por tanto el peor de los polinómicos. Para sorpresa, el *kernel* de base radial es el que obtiene los mejores resultados, ya que obtiene una *accuracy* de casi 0.9. Por tanto, en esta primera comparación vemos que el *kernel* radial es el mejor de entre los probados.

Si ahora observamos los resultados para el conjunto de validación, vemos que en general son parecidos a los obtenidos por validación, un poco por encima o por debajo de ellos, en general. Vemos que de nuevo el *kernel* lineal es el peor de todos ellos. Observando su matriz de confusión vemos que clasifica bien todos los ejemplos positivos pero mal todos los negativos. Por tanto, se puede directamente descartar este modelo, ya que lo hace bastante mal en general. Si observamos los polinómicos, vemos que el mejor es el de grado 4, ya que es el que clasifica mejor, tiene una mayor *accuracy* y unos mejores valores de precisión y *recall* en general. No obstante, parece que todos los polinómicos tienen problemas serios con los casos negativos (cuando no es pierna), ya que se equivocan demasiado (tienen un *recall* muy bajo para los negativos). Finalmente, si observamos los resultados ofrecidos por el *kernel* de base radial, vemos que los resultados son los mejores obtenidos hasta ahora en todas las métricas. Es el que mejor clasifica tanto los ejemplos positivos como los negativos. Por tanto, tal y como pasaba en el caso anterior, podemos afirmar que el *kernel* de base radial es el mejor modelo, y es el claro candidato a ser mejorado.

Una vez que se ha determinado el mejor modelo, se ha realizado una `GridSearchCV` para obtener los mejores hiperparámetros. Se ha encontrado que el mejor valor de C es 10. Por tanto, una vez que se ha determinado este valor, como el clasificador ya estaba entrenado para la mejor combinación de hiperparámetros, podemos estudiar su comportamiento con el conjunto de validación.

Vemos que ha habido bastante mejora respecto al modelo base, ya que clasifica casi todos los ejemplos positivos como piernas a excepción de uno, y falla solo unos pocos negativos. Vemos además que todos los otros valores han mejorado, desde el *accuracy* hasta la precisión y el *recall*. Por tanto, podemos afirmar que sí que ha merecido la pena buscar los mejores hiperparámetros, ya que nos han permitido mejorar bastante los resultados obtenidos.

Con todo esto hecho, ahora podemos entrenar el modelo con todos los datos de los que disponemos. De esta forma, los resultados obtenidos en el futuro muy posiblemente serán mejores que los que podríamos obtener actualmente, ya que al tener más datos con los que entrenar, el clasificador será capaz de generalizar mejor, ya que ha visto más situaciones diferentes. Una vez hecho el último ajuste, guardamos los datos en un fichero utilizando el módulo `joblib` de `sklearn`.

6. Predicción y corrección de los datos

Ahora que hemos entrenado el modelo, es hora de ponerlo a prueba con la escena de test. Para ello, se ha creado el *script* `predecir.py`, el cuál se conecta al servidor de V-REP, lee los datos del laser, procesa la información y obtiene los

puntos, extraer los clusters, obtiene las características y predice las clases de los objetos. Aparte, con lo que ha predicho, se muestra un gráfico, donde se pueden ver los clusters obtenidos, a qué clase pertenecen y el punto medio de la línea que une los centroides de dos clusters cercanos de la misma clase, representando la posición del objeto detectado. Finalmente se cierra la conexión y se detiene la simulación.

Para hacer todo esto, se han reutilizado muchas de las funciones anteriores, como la de iniciar y parar la conexión, la de procesar la lectura láser, la de obtener los clústeres y la de obtener las características. Adicionalmente, se han implementado las siguientes funciones:

- **calcular_centroide**: Función que calcula el centroide de un cluster.
- **calcular_centroides_clusters**: Función que calcula el centroide de un grupo de clusters, juntándolos todos luego en un único *array*.
- **calcular_punto_medio_centroides**: Función que calcula el punto medio de los centroides de dos clusters próximos. Para hacerlo, se utiliza un **KDTree**, es decir, un árbol multidimensional. Esta estructura de datos permite obtener los k puntos más próximos a uno. Como vamos a buscar los puntos más próximos a los mismos que forman el árbol, tendremos que buscar para $k = 2$ puntos, ya que el primer punto más cercano será él mismo. Una vez que tenemos los índices de los vecinos más cercanos, procesamos cada centroide. Se comprueba si el centroide más próximo es de la misma clase y, en caso de serlo, se calcula el punto medio entre los dos centroides y se añade a la lista. Además, al compartir la misma clase, se guarda la etiqueta del punto medio. Si se da el caso de que no son de la misma clase, se guarda el centroide actual en una lista aparte, además de su etiqueta. Esto será más útil más adelante, ya que esta información no se pintará de momento.
- **plot_prediccion**: Función que pinta el gráfico mencionado anteriormente.
- **predecir**: Función que hace todo el proceso anteriormente descrito (detectar, obtener puntos, clusters, características, predecir y cálculo de los puntos medios).

Una vez explicada la funcionalidad básica, vamos a poner en funcionamiento el programa. Vamos a coger la escena de test disponible y vamos a predecir las clases de los objetos disponibles. A continuación podemos ver dicha escena:

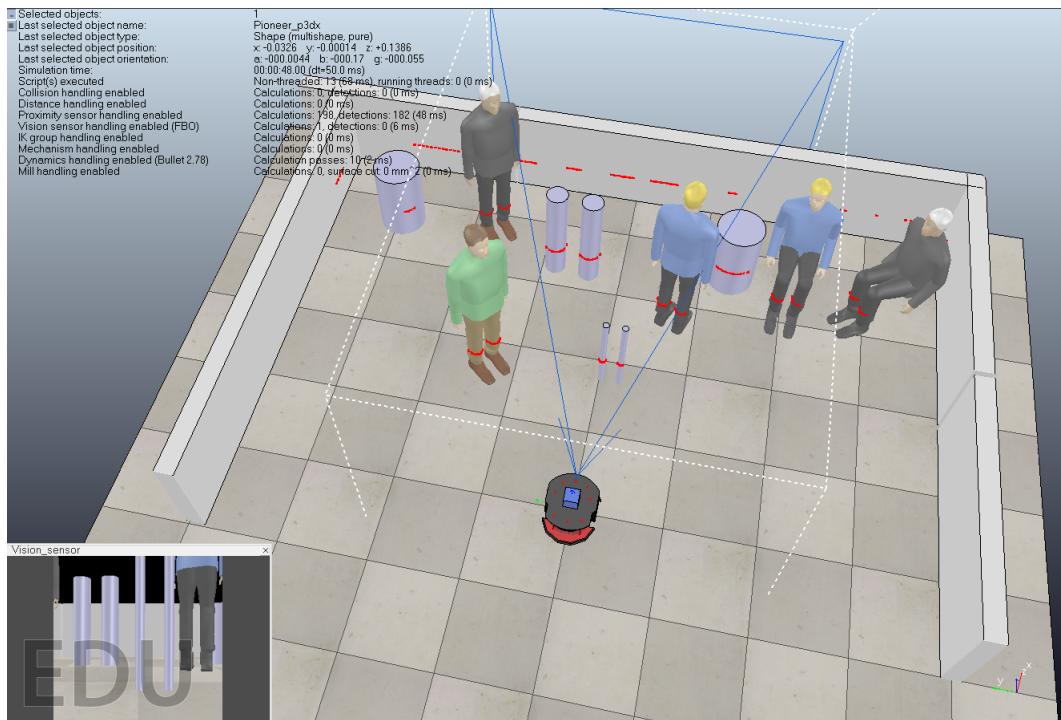


Figura 6: Escena inicial de test.

Ejecutamos el programa y obtenemos los siguientes resultados:

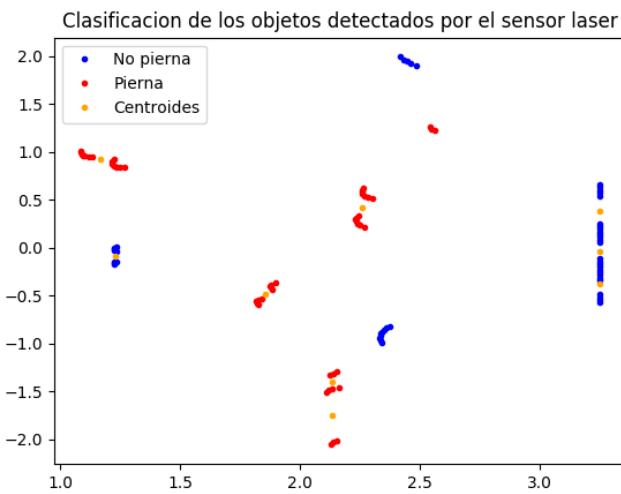


Figura 7: Resultados de la predicción de la escena.

Como podemos ver en la figura 7, predice bien casi todos los objetos menos el par de cilindros que están al fondo, entre la persona con el jersey gris y la persona con el jersey azul. Podemos ver que esos dos los clasifica como “pierna” en vez de como “no pierna”. Además, parece que no detecta la pared de la izquierda, y para la persona de la derecha del todo solo detecta una pierna, haciendo que el punto medio del objeto esté entre las dos personas sentadas. Sorprendentemente clasifica bien la pared, teniendo en cuenta que hasta ahora nunca ha visto algo así.

A la vista de los resultados, podemos afirmar que estos no son del todo buenos, ya que se esperaba que lo hiciese perfecto en la escena de test. Para intentar mejorar los resultados, podemos hacer toda una serie de cosas, como tomar más muestras, cambiar la forma en la que se generan los clusters, o intentar mejorar más el clasificador, buscando por ejemplo una mejor combinación de hiperparámetros que la que se tiene actualmente.

La solución que yo he propuesto consta de dos partes. Por una parte, se modifica la forma de generar clusters. Ahora el número **máximo de puntos por cluster** es de 13, y el **umbral de la distancia entre dos puntos consecutivos** es 0.05. El número mínimo de puntos se queda igual. Estos valores se han obtenido tras realizar bastantes pruebas, buscando cuál podría ser la mejor combinación de parámetros para la generación de clusters. Por otra parte, he aumentado el número de valores de C que probar a la hora de buscar los mejores hiperparámetros. He añadido valores más grandes, como 50 y 100, con el objetivo de penalizar más los puntos que caen dentro del margen. Por tanto, he tenido que volver a agrupar los puntos en clusters, generar de nuevo características y entrenar de nuevo. A la hora de entrenar, si se visualizan los datos, se obtiene el siguiente gráfico:

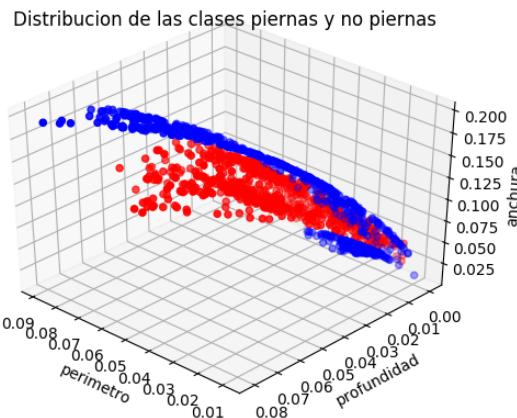


Figura 8: Distribución de las clases. En azul “no piernas”, en rojo “piernas”.

Vemos que los puntos siguen más o menos la misma organización que la que se puede ver en la figura 5, solo que esta vez parece que hay algo menos de puntos que se puede dar que no se clasifiquen correctamente.

Si analizamos los resultados del entrenamiento, vemos lo siguiente:

```
----- Validacion de los modelos -----
Modelo evaluado: SVM Kernel Lineal Accuracy media en 5-fold CV: 0.5237
Modelo evaluado: SVM Kernel Polinomico Grad=2 Accuracy media en 5-fold CV: 0.6858
Modelo evaluado: SVM Kernel Polinomico Grad=3 Accuracy media en 5-fold CV: 0.6995
Modelo evaluado: SVM Kernel Polinomico Grad=4 Accuracy media en 5-fold CV: 0.6995
Modelo evaluado: SVM Kernel RBF Accuracy media en 5-fold CV: 0.8834

----- Resultados de entrenamiento de los modelos -----
----- SVM Kernel Lineal -----
Acc_val: (TP+TN)/(T+P) 0.5055
Matriz de confusión Filas: verdad Columnas: predicción
[[ 0 136]
 [ 0 139]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score   support
          0         0.00     0.00     0.00       136
          1         0.51     1.00     0.67       139

      accuracy           0.51      275
      macro avg        0.25     0.50     0.34      275
  weighted avg        0.26     0.51     0.34      275

----- SVM Kernel Polinomico Grad=2 -----
Acc_val: (TP+TN)/(T+P) 0.6836
Matriz de confusión Filas: verdad Columnas: predicción
[[ 67  69]
 [ 18 121]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score   support
          0         0.79     0.49     0.61       136
          1         0.64     0.87     0.74       139

      accuracy           0.68      275
      macro avg        0.71     0.68     0.67      275
  weighted avg        0.71     0.68     0.67      275

----- SVM Kernel Polinomico Grad=3 -----
Acc_val: (TP+TN)/(T+P) 0.6873
Matriz de confusión Filas: verdad Columnas: predicción
[[ 65  71]
 [ 15 124]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score   support
          0         0.81     0.48     0.60       136
          1         0.64     0.89     0.74       139

      accuracy           0.69      275
```

```

macro avg      0.72      0.69      0.67      275
weighted avg   0.72      0.69      0.67      275

----- SVM Kernel Polinomico Grad=4 -----
Acc_val: (TP+TN)/(T+P)  0.6873
Matriz de confusión Filas: verdad Columnas: predicción
[[ 63  73]
 [ 13 126]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score  support
          0         0.83     0.46     0.59      136
          1         0.63     0.91     0.75      139

      accuracy           0.69      275
macro avg      0.73      0.68      0.67      275
weighted avg   0.73      0.69      0.67      275

----- SVM Kernel RBF -----
Acc_val: (TP+TN)/(T+P)  0.8800
Matriz de confusión Filas: verdad Columnas: predicción
[[130   6]
 [ 27 112]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score  support
          0         0.83     0.96     0.89      136
          1         0.95     0.81     0.87      139

      accuracy           0.88      275
macro avg      0.89      0.88      0.88      275
weighted avg   0.89      0.88      0.88      275

----- Mejor estimador -----
SVC(C=100, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=1, shrinking=True, tol=0.001,
    verbose=False)

----- Resultados finales -----
Acc_val: (TP+TN)/(T+P)  0.9673
Matriz de confusión Filas: verdad Columnas: predicción
[[133   3]
 [ 6 133]]
Precision= TP / (TP + FP), Recall= TP / (TP + FN)
f1-score es la media entre precisión y recall
      precision    recall   f1-score  support
          0         0.96     0.98     0.97      136
          1         0.98     0.96     0.97      139

      accuracy           0.97      275
macro avg      0.97      0.97      0.97      275
weighted avg   0.97      0.97      0.97      275

```

Los resultados son en general muy parecidos a los obtenidos antes, solo que

ahora el modelo con *kernel* de base radial base (sin mejora de hiperparámetros) es algo peor que en el caso anterior. A la hora de ajustar los hiperparámetros, vemos que el mejor valor de C es 100, y que la *accuracy* obtenida es igual que en el caso anterior. La matriz de confusión indica que ahora se equivoca en menos casos cuando no se trata de una pierna y un poco más cuando sí lo es, pero hay que considerar que hay un mayor número de datos en el conjunto de validación. Observando los valores de precisión y *recall* vemos que son muy parecidos a los obtenidos anteriormente, solo que están cambiados (para la clase para la que antes la precisión era mejor, ahora es peor, y lo mismo pasa con el *recall*). Sin embargo, su media sigue siendo la misma, y tampoco es que los valores obtenidos disten mucho de los anteriores. Por tanto, en un principio, el modelo ofrecerá unos resultados igual de buenos que el anterior.

Si ahora lo llevamos a la escena de test y repetimos el proceso, se obtiene la siguiente predicción:

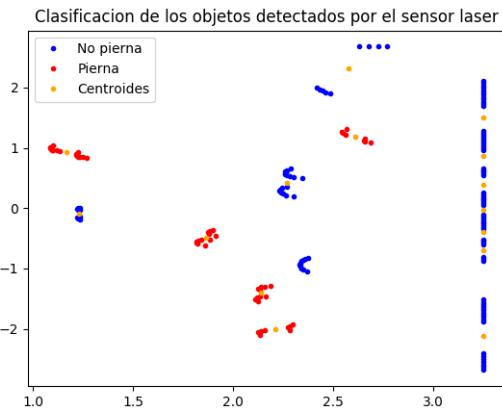


Figura 9: Resultados de la predicción de la escena.

Vemos que ahora sí que clasifica los cilindros bien como “no pierna”. Además, ahora detecta también la pared izquierda, además de más puntos en la pared del fondo (las cuales sigue clasificando bien). Adicionalmente, detecta la pierna de la otra persona sentada a la derecha del todo, estimando mejor por tanto el centro del objeto.

Por tanto, vemos que estos cambios, en un principio, han mejorado el modelo que teníamos anteriormente.

Para comprobar si es bueno, vamos a cambiar la orientación del robot y vamos a predecir, para ver qué tal se comporta. A continuación se pueden ver los resultados, junto con la modificación a la escena realizada:



Figura 10: Escena de test con rotación del robot a la izquierda.

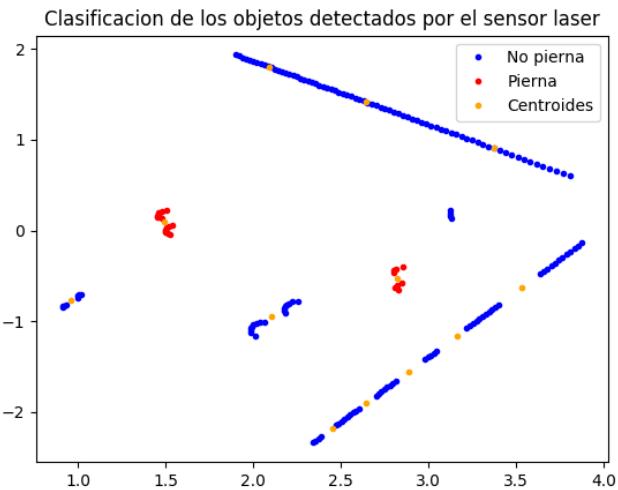


Figura 11: Resultados de la predicción de la escena anterior.



Figura 12: Escena de test con rotación del robot a la derecha.

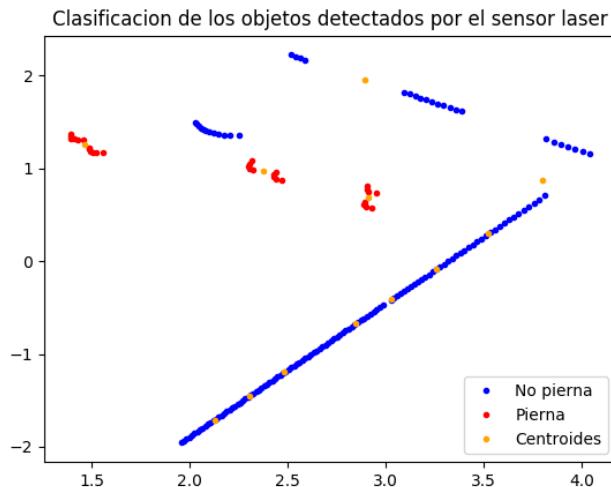


Figura 13: Resultados de la predicción de la escena anterior.

Vemos que en los dos casos el modelo ha predicho todos los objetos correctamente. Por tanto podemos afirmar con cierta certeza que nuestro modelo es bastante bueno, ya que es capaz de clasificar correctamente los elementos de la escena de test, independientemente de la orientación del robot. Para tener una certeza del 100% de que lo hace bien, habría que probar todos los posibles casos y, en caso de ver en algún momento algo no se detectase correctamente, se podría probar a modificar la forma de generar los clusters o tomar más muestras.

7. Apartado opcional: detección de objetos

Una vez hecho todo lo anterior, se ha realizado la detección de los objetos de la escena. El objetivo es detectar todos los objetos y crear una tabla en formato HTML que visualice la clase real del objeto, la clase predicha por el clasificador, la distancia al objeto y una imagen del objeto detectado. Los objetos son tanto los puntos medios de la linea que une dos clusters próximos como los centroides de los objetos no emparejados (recordemos la función para calcular los puntos medios de la sección anterior, que también devolvía esa información).

Para hacer esta tarea, se ha creado el archivo **detectar.py**. Este archivo, igual que el anterior, reutiliza ciertas funciones anteriores. En concreto, utiliza la función de **predecir** y la de **capturar_guardar_imagen**, entre otras, como la de iniciar la conexión y detener la simulación y la conexión.

El funcionamiento principal del *script* es conectarse con el servidor, obtener *handlers* para el robot y los motores derecho e izquierdo (aparte del de la cámara que se tenía anteriormente), predecir, juntar los puntos medios obtenidos con los centroides no emparejados y hacer lo mismo para las etiquetas (recordemos que tenemos que detectar todos los objetos, incluso aquellos cuyos centroides no han podido ser emparejados con otros), obtener las orientaciones y los valores reales de los objetos (estos valores están insertados a mano en el *script*, con lo cuál cualquier modificación a la escena va a hacer que los valores reales dejen de ser válidos), ordenar las orientaciones y el resto de información en función de los valores de las orientaciones, crear el directorio de salida de las imágenes, rotar el robot a cada orientación y capturar la imagen de la cámara y generar el fichero HTML de salida. Finalmente, se cierra la conexión y la simulación.

De lo explicado anteriormente, hay algunos puntos a destacar:

1. Para hacer las rotaciones se ha utilizado la función **simxSetObjectOrientation**. Esta función modifica directamente la orientación de los objetos referenciados por los *handlers* a la orientación especificada. Para especificar la orientación, se tiene que utilizar el formato de coordenadas de Euler, las cuáles son α

(rotación en el eje X), β (rotación en el eje Y) y γ (rotación en el eje Z). Como nos interesa rotar el robot en el eje Z , solo especificaremos valores para γ , dejando el resto a 0.

2. Todas las rotaciones están especificadas en radianes, ya que es el formato que acepta V-REP.
3. El motivo de obtener los *handlers* de los motores es que también se rotarán estos, ya que si solo se rota el robot, la cámara se inclina un poco hacia un lado, haciendo que las capturas que se tomen posteriormente estén algo inclinadas.
4. El motivo por el que se ordenan las orientaciones y el resto de datos no tiene mucha importancia. Simplemente se hace para que todas las capturas de la cámara se puedan tomar en una sola pasada, sin tener que volver hacia atrás en ningún momento.

Una vez dicho esto, vamos a comentar brevemente las funciones implementadas en este archivo, las cuáles son utilizadas para completar la tarea de generación de resultados:

- **rotar_robot**: Función que rota el robot y los motores hasta situarlos en γ grados. Al final, espera un segundo para evitar fallos de comunicación con el servidor (si se hace demasiado rápido la actualización no da tiempo a que se haga correctamente).
- **calcular_distancias_objetos**: Función que calcula la distancia del robot a un conjunto de objetos.
- **calcular_orientaciones**: Función que calcula las orientaciones. Para calcular las orientaciones, primero calcula las distancias a los objetos. A continuación, se llevan dichos puntos al eje X (haciendo que su coordenada y valga 0) y se calcula la distancia hasta cada uno de ellos. De esta forma, se tienen dos lados de un triángulo rectángulo para cada objeto detectado, de forma que la hipotenusa está formada por la línea que va del robot al objeto en cuestión, y uno de los catetos es el que va del robot al punto trasladado sobre el eje X del objeto en cuestión. Utilizando las dos distancias anteriores se puede calcular el coseno, y para sacar el valor de γ , la orientación, solo hace falta aplicar la función arco coseno. Todas las orientaciones calculadas son positivas; es decir, están en el primer cuadrante. No obstante, en algunos casos tienen que ser negativas, ya que hay objetos en el cuarto cuadrante. Por tanto, hay que cambiar los valores de dichas orientaciones a negativo en los casos en los que el objeto detectado quede a la derecha del robot, es decir, donde la coordenada y del objeto sea negativa.

- `escribir_comienzo_fichero`: Función que escribe el comienzo del fichero.
- `escribir_fila`: Función para escribir una fila de la tabla con el formato antes especificado.
- `escribir_final_fichero`: Función para escribir el final del archivo.

Los resultados se pueden ver en el archivo **resultados.html**, donde se puede ver la tabla con el formato anteriormente mencionado para cada uno de los objetos detectados de la escena.