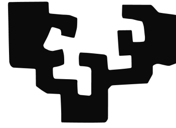
eman ta zabal zazu



Universidad del País Vasco
Euskal Herriko Unibertsitatea

Grado en Ingeniería Informática
Computación

Proyecto de Fin de Grado

Reconocimiento de posturas mediante Kinect en ROS

Autor
Asier Aguado

informatika fakultatea
facultad de informática

2015

Resumen

Proyecto de Fin de Grado, especialidad en Computación. Se ha desarrollado un software en ROS para detectar posturas y movimientos de personas. Para ello, se utiliza la información del esqueleto proporcionada por el sensor Kinect y la biblioteca *OpenNI*. Se ha realizado un enfoque basado en técnicas de aprendizaje supervisado para generar modelos que clasifiquen posturas estáticas. En el caso de los movimientos, el enfoque se ha basado en *clustering*. Estos modelos, una vez generados, se incluyen como parte del software, que reacciona ante las posturas y gestos que realice un usuario. La detección automática de posturas resulta interesante para diferentes aplicaciones, como aplicaciones médicas o interacción inteligente mediante visión por computador.

Laburpena

Gradu Amaierako Proiektua, Konputazio espezialitatea. Proiektu honetan pertsonen mugimenduak eta posturak detektatzeko software bat garatu da ROS-en. Horretarako, *OpenNI* liburutegiak emandako eskeletoaren informazioa erabili da. Planteamendua gainbegiratutako ikasketa automatikoaren inguruan oinarritu da, postura estatikoak sailkatzen dituzten modeloak eratzeko. Mugimenduen kasuan, planteamendua *clustering* teknikan oinarritu da. Modelo hauek, eratu eta gero, softwarearen parte bilakatu ditugu. Software honek erabiltzaile batek egindako postura eta keinuen ondorioz erantzungo du. Posturen detekzio automatikoa hainbat aplikaziorako interesgarria izan daiteke, adibidez, aplikazio medikoak edo interakzio adimenduna konputagalu bidezko ikusmenaren bidez.

Abstract

Bachelor Thesis, speciality in Computer Science. A software in ROS has been developed to detect postures and movements of people, using the skeleton information provided by the *OpenNI* library. A focus in supervised learning has been used for generating static posture classifier models. In the case of movements, the focus has been done in clustering techniques. These models are included as part of the software once generated, which reacts to postures and gestures made by any user. The automatic detection of postures is interesting for many applications, such as medical applications or intelligent interaction based on computer vision.

Índice general

Resumen I

Índice general III

Índice de figuras VII

Indice de tablas IX

1. Introducción 1

1.1. Kinect . . . 1

1.2. Estado del arte . . . 2

2. Documento de Objetivos del Proyecto 3

2.1. Objetivos . . . 3

2.1.1. Interés . . . 4

2.2. Alcance . . . 4

2.3. Herramientas a utilizar . . . 4

2.3.1. Dispositivos físicos . . . 4

2.3.2. Software . . . 5

2.4. Fases del proyecto . . . 6

2.5. Tareas del proyecto . . . 6

2.5.1. Fase 1: Auto-aprendizaje 6
2.5.2. Fase 2: Preparación del entorno 7
2.5.3. Fase 3: Desarrollo 7
2.5.4. Fase 4: Documentación 8
2.6. Calendario del proyecto 8
2.6.1. Estimación de duración del proyecto 9
2.6.2. Estimación de la distribución en fases 9
2.7. Hitos 9
2.8. Riesgos 11
2.8.1. Periodo de intercambio 11
2.8.2. Pérdida de información 11
2.8.3. Averías 11

3. Conceptos básicos **13**
3.1. Robot Operating System (ROS) 13
3.2. ROS Topics 14
3.3. Servicios y parámetros 15

4. Configuración inicial **19**
4.1. Instalación de Ubuntu, ROS y Kinect 19
4.1.1. Instalación de ROS y OpenNI para Kinect en Ubuntu 19
4.2. Preparación de ROS 21
4.2.1. Configuración del workspace 21
4.2.2. Compilar un *publisher* y *subscriber* 22
4.3. Kinect en ROS con OpenNI tracker 26
4.3.1. Lectura de la información en C++ 28

5. Reconocimiento de posturas **29**

5.1. Un detector simple 30

5.2. Aprendizaje supervisado 30

5.2.1. Conjunto de clases 31

5.2.2. Vector de características 33

5.3. Obtención de datos 34

5.4. Preprocesado 35

5.4.1. Selección de atributos 36

5.4.2. Detección de *outliers* 36

5.5. Clasificadores 37

5.5.1. J48 38

5.5.2. J48graft 38

5.5.3. BFTree 39

5.5.4. Clasificadores bayesianos 40

5.5.5. Nearest neighbour 41

5.5.6. Nearest neigbour generalizado 42

5.5.7. Random Forest 42

5.6. Experimentos con posturas generales 46

5.6.1. Evaluación de clasificadores 46

5.7. Clasificación en directo 47

5.8. Conclusiones 47

6. Reconocimiento de movimientos **49**

6.1. Detección de movimiento 50

6.2. Estados de movimiento 51

6.3. Precisión del movimiento 52

6.3.1. Postura final 52
6.3.2. Posturas intermedias 53
6.3.3. Clustering en Weka 53
6.3.4. Precisión de posturas intermedias 54

7. Implementación **57**
7.1. Árboles de decisión 57
7.2. Módulos y comunicación con ROS 59
7.3. Interfaz gráfica del usuario 60

8. Conclusiones **63**
8.1. Posibles mejoras 64
8.2. Posibles aplicaciones 65
8.3. Comentarios adicionales 65

Anexos

A. Implementación de árboles de clasificación **69**
A.1. Clase ‘modeltree’ 69
A.2. Clase ‘randomforest’ 71
A.3. Generación de *random forests* desde fichero 71

Bibliografía **73**

Índice de figuras

3.1. Ventana gráfica de turtlesim. 14

3.2. Representación gráfica de los nodos, el *topic*, y la comunicación entre nodos con rqt_graph. 15

3.3. Visualización del cambio de color al modificar un parámetro de *turtlesim*. 18

4.1. Un *publisher* y un *subscriber* simple en ejecución. 25

4.2. Visualización de 'tf' con openni_tracker y rviz. 27

5.1. Ejemplos de posturas *stand*, *sit*, *lie* y *bend*. 32

5.2. Proyección de los ángulos obtenidos a partir de la posición del torso T y la posición de un punto P del esqueleto. 33

5.3. Selección de atributos en Weka. 36

5.4. Distribución de *outliers*. 37

5.5. Conjunto de datos una vez eliminados los *outliers*. 38

5.6. Árbol de decisión generado con J48. 40

6.1. Esquema básico de los estados y transiciones. 51

6.2. Clustering en Weka con el algoritmo K-Means. 54

7.1. Representación en grafo de los nodos y *topics* de ROS durante una ejecución de todos los componentes. 61

7.2. Ventana de la interfaz gráfica del usuario. 62

Indice de tablas

5.1. Resultados de la validación y matriz de confusión del clasificador J48 . . 39

5.2. Resultados de la validación y matriz de confusión del clasificador C4.5-graft 41

5.3. Resultados de la validación y matriz de confusión del clasificador BFTree 42

5.4. Resultados de la validación y matriz de confusión del clasificador Naive Bayes 43

5.5. Resultados de la validación y matriz de confusión del clasificador BayesNet 44

5.6. Resultados de la validación y matriz de confusión del clasificador Nearest Neighbour 44

5.7. Resultados de la validación y matriz de confusión del clasificador Nearest Neighbour generalizado 45

5.8. Resultados de la validación y matriz de confusión del clasificador Random Forest 45

5.9. Tasas de acierto clasificando posturas generales 46

5.10. Matriz de confusión del clasificador Random Forest con posturas generales 47

1. CAPÍTULO

Introducción

El objetivo de este proyecto es crear un software de reconocimiento de posturas, y también de reconocimiento de movimientos. Este problema de reconocimiento de movimientos suele ser tratado de forma similar como *reconocimiento de gestos*, un reto ya conocido en el ámbito de visión por computador. Podríamos considerar también que las posturas son *gestos estáticos*. El reconocimiento de gestos aspira a mejorar la interacción entre personas y máquinas, haciéndola más natural y preferible para las personas. Se desarrollará un software para reconocimiento de posturas y movimientos en Robot Operating System (ROS) [http://www.ros.org], lo que permitirá que sea utilizado en aplicaciones de robótica.

1.1. Kinect

En este proyecto utilizaremos el sensor Kinect de Xbox 360 [http://www.xbox.com/en-US/xbox-360/accessories/kinect], conectado a un equipo portátil mediante USB.

Kinect tiene tres sensores diferentes: una cámara de infrarrojos, una cámara RGB, y un array de micrófonos. También incluye un emisor de infrarrojos, que permite al sensor funcionar en cualquier condición de luz ambiental. Esta combinación de sensores lo hace capaz de visualizar en tres dimensiones (3D).

La cámara y el sensor de profundidad del Kinect no son sólo compatibles con Xbox 360. Existen varios controladores de software libre para Kinect, entre ellos, el desarrollado

por el propio fabricante de los sensores (PrimeSense): OpenNI. Además, también han desarrollado un *middleware* para seguimiento del movimiento: NITE. Estos controladores son libres y pueden ser compilados e instalados en Linux.

1.2. Estado del arte

El problema del reconocimiento de posturas y gestos ha sido tratado en muchas ocasiones. Normalmente implica dos áreas: visión por computador y aprendizaje automático (*machine learning*). Inicialmente, lo más habitual era utilizar imágenes 2D proporcionadas por una cámara digital. La principal limitación de utilizar imágenes 2D es que no se puede identificar la profundidad de forma precisa. Para evitar esto, es posible utilizar sistemas de varias cámaras (dos o tres cámaras digitales), que proporcionan visión en estéreo, de la cual sí se puede obtener la profundidad de imagen. El lanzamiento del sensor Kinect supuso un cambio en el reconocimiento de gestos, y en la visión por computador en general. Kinect es un sensor barato y efectivo para reconocer los gestos de una persona en 3D. Además, el propio fabricante del sensor proporciona un controlador de interacción natural que utiliza un modelo de esqueleto y articulaciones.

Existen varios ejemplos de usos del Kinect para el reconocimiento de posturas. T. Le, M. Nguyen y T. Nguyen [Le et al., 2013] proponen un método para el reconocimiento de posturas, realizando experimentos con las posturas *de pie*, *sentado*, *tumbado* e *inclinado*. Para ello, utilizan el esqueleto proporcionado por Kinect, y un modelo de aprendizaje supervisado con máquinas de vectores de soporte (SVM: *support vector machines*). En otro trabajo con un enfoque similar [Xiao et al., 2012], tratan de reconocer una base de datos de muchas más posturas, utilizando selvas de árboles de decisión aleatorios (*random forests*).

En el campo del reconocimiento de gestos (o movimientos) el enfoque del *machine learning* también es habitual. Una biblioteca open-source en C++ para reconocimiento de gestos [`http://www.nickgillian.com/software/grt`] ofrece compatibilidad con diferentes sensores (incluyendo Kinect), y varios algoritmos de *machine learning* que pueden ser útiles para el reconocimiento de posturas y gestos. Cabe destacar que los algoritmos de *clustering* están entre los recomendados en la biblioteca para el reconocimiento de gestos con movimiento.

2. CAPÍTULO

Documento de Objetivos del Proyecto

2.1. Objetivos

El objetivo principal de este proyecto es crear un programa en ROS que detecte posturas y movimientos. Este programa estará diseñado de forma que pueda utilizarse para un proyecto mayor, por lo que se sus salidas de datos deberían poder leerse fácilmente desde otro programa de ROS siguiendo un esquema típico. Para hacer un programa más modular, el detector de posturas y el detector de movimientos se ejecutarán por separado.

1. Detector de posturas. Debe ser capaz de detectar y clasificar entre diferentes posturas de cualquier usuario que se encuentre frente al Kinect. Estas posturas formarán parte de un grupo de posturas previamente definido, es decir, habrá que clasificar la postura actual entre todas las posibles posturas disponibles.

2. Detector de movimientos. Detectará un movimiento y lo comparará con otro previamente definido. Debe ser capaz de indicar en qué medida se ha realizado el movimiento correctamente, considerando que el movimiento previamente definido es el correcto.

Como objetivo adicional, también se podrá desarrollar una interfaz gráfica sencilla, que permitirá visualizar la información proporcionada por los detectores de posturas y movimientos.

2.1.1. Interés

Además de obtener el resultado a final –el programa a desarrollar– también hay especial interés en que el desarrollo de este proyecto sirva como una primera toma de contacto con ROS, un conjunto de software y bibliotecas cada vez más utilizado para proyectos de robótica.

2.2. Alcance

Como resultado del proyecto, se obtendrá lo siguiente:

1. Un programa que detecte y clasifique posturas de cualquier usuario, utilizando los datos del sensor Kinect. Al menos diferenciará entre tres posturas generales: sentado, de pie y tumbado.

2. Un programa que sea capaz de seguir los movimientos de un usuario, detectando cuándo realiza correctamente un movimiento previamente definido, y en qué medida lo ha realizado correctamente.

2.3. Herramientas a utilizar

Este proyecto requiere diferentes herramientas para su desarrollo: tanto dispositivos físicos como software.

2.3.1. Dispositivos físicos

- Equipo personal con Ubuntu 12.04. Se utilizará un Macbook Air 6.2 13' con dos particiones de arranque (Mac OSX / Ubuntu).

- Sensor Kinect. Se utilizará un Kinect para Xbox 360 proporcionado por la Facultad de Informática.

- Disco duro externo para copias de seguridad.

2.3.2. Software

El software utilizado para este proyecto no tendrá ningún coste económico. Preferiblemente se optará por software libre.

Software para desarrollo

- Ubuntu 12.04.
- ROS Hydro.
  http://www.ros.org
- Weka: software para *machine learning*.
  http://www.cs.waikato.ac.nz/ml/weka
- OpenNI, NiTE: drivers libres de PrimeSense para Kinect.
- openni_tracker para ROS Hydro.
- NetBeans IDE: entorno de desarrollo integrado.
- Qt, Qt Creator: entorno de desarrollo integrado para interfaces gráficas.
- gedit, emacs: editores de texto.
- Terminator: emulador de terminal con utilidades añadidas.

Software para documentación

- LaTeX, para la memoria del proyecto.
- TexMaker como editor de LaTeX.
- LibreOffice Calc: para seguimiento y control de las horas dedicadas al proyecto.
- LibreOffice Writer: para borradores de la memoria y apuntes.

2.4. Fases del proyecto

Para llevar a cabo este proyecto, lo dividiremos en cuatro fases, que a su vez contendrán todas las tareas a realizar.

1. **Auto-aprendizaje**. En esta primera fase se aprenderán los conceptos básicos de ROS para poder desarrollar un programa en esta plataforma, y también otros conceptos específicos para el proyecto a desarrollar.
2. **Preparación del entorno de trabajo**. Instalación de Ubuntu, ROS, OpenNI y el resto de utilidades para trabajar con Kinect.
3. **Desarrollo**. Dentro de esta fase se realizarán todas las tareas de programación y otras tareas necesarias para alcanzar los objetivos del proyecto.
4. **Documentación (memoria)**. Fase a realizar a lo largo de todo el proyecto, cuya importancia será mayor una vez alcanzados los objetivos.

La fase 3, a su vez, será la más larga del proyecto y estará distribuida en varias subfases:

3.1. Desarrollo del detector de posturas

3.2. Desarrollo del detector de movimientos

3.3. Desarrollo final

2.5. Tareas del proyecto

Cada una de las fases del proyecto contiene un número de tareas individuales. Una vez finalizadas todas las tareas, puede darse la fase del proyecto como completada.

2.5.1. Fase 1: Auto-aprendizaje

T 1.1 Instalar ROS y preparar un primer entorno de trabajo para aprendizaje.

T 1.2 Completar el tutorial básico de wiki.ros.org, hasta el “talker” y el “listener”.

T 1.3 Escribir un primer programa en ROS (C++).

T 1.4 Estudiar la documentación de bibliotecas específicas para este proyecto: 'tf', 'openni_tracker' y 'std_msgs'.

2.5.2. Fase 2: Preparación del entorno

T 2.1 Preparar el entorno de trabajo para trabajar en el programa.

T 2.2 Instalar todos los drivers y bibliotecas para el *skeleton tracker* de Kinect en ROS.

T 2.3 Instalar herramientas adecuadas para trabajar con ROS y programar en C++.

T 2.4 Crear un programa simple con el que se pueda comprobar la lectura de datos del *skeleton tracker*.

T 2.5 Preparar el control de versiones.

2.5.3. Fase 3: Desarrollo

T 3.1.1 Crear un programa que guarde los ángulos entre el torso y las articulaciones en un fichero de formato similar al de Weka.

T 3.1.2 Concretar las posturas que el programa detectará. Definir los atributos y clases que se procesarán durante el aprendizaje supervisado.

T 3.1.3 Obtener un fichero de datos con posturas clasificadas, con el programa que guarda los ángulos y una persona colocada frente al Kinect.

T 3.1.4 Importar los datos en Weka. Realizar diferentes pruebas y experimentos con modelos de clasificación. Seleccionar un modelo adecuado para detectar posturas de forma instantánea.

T 3.1.5 Desarrollar un programa que clasifique, a partir de las lecturas de ángulos entre el torso y las articulaciones, la postura en la que se encuentra el usuario. Este programa utilizará el modelo de clasificación elegido en base a los experimentos con Weka.

T 3.2.1 Desarrollar un programa que detecte si el usuario está en movimiento o parado.

T 3.2.2 Diseñar una solución para hacer el seguimiento de un movimiento concreto y detectar si se completa correctamente.

T 3.2.3 Implementar la solución para hacer seguimiento de un movimiento y detectar si se completa correctamente.

T 3.3.1 Implementar la comunicación de salida de los nodos “detector de posturas” y “detector de movimientos”.

T 3.3.2 Proporcionar un *launcher* para el programa.

T 3.3.3 Revisar el código de los programas. Solucionar errores detectados.

T 3.3.4 Empaquetar el resultado final.

2.5.4. Fase 4: Documentación

T 4.1 Desarrollo de una memoria borrador a lo largo de todo el proyecto.

T 4.2 Establecer un control de versiones para la memoria.

T 4.3 Definición del alcance, capítulos y secciones de la memoria final.

T 4.4 Desarrollo de la memoria del proyecto.

2.6. Calendario del proyecto

Este es un proyecto con plazos de inicio y fin muy extensos, pero con una dedicación distribuida de forma no uniforme. Esta falta de uniformidad se debe a una estancia de intercambio Erasmus en Friburgo (Alemania) de abril a finales de agosto, que condiciona las fechas más idóneas para una mayor dedicación al proyecto.

Fechas de estancia en Alemania:

Salida estimada	01 / 04 / 2015
Regreso estimado	23 / 08 / 2015

No va a ser posible realizar la defensa del proyecto durante la estancia en Alemania, por lo que la defensa del proyecto ha sido planificada para después del regreso (en septiembre).

2.6.1. Estimación de duración del proyecto

Inicio del proyecto	13 / 10 / 2014
Fin del proyecto	02 / 07 / 2015
Entrega del proyecto	02 / 09 / 2015

La fecha de fin del proyecto se ha situado dos meses antes de la fecha de entrega, debido a que en los meses de julio y agosto puede haber una mayor carga de trabajo en la universidad de Friburgo. Además, habría que añadir que sería más difícil trabajar en el proyecto desde otro país, donde las posibilidades de comunicación con la dirección del proyecto son más limitadas (no es posible hacer reuniones presenciales).

2.6.2. Estimación de la distribución en fases

Fase	Inicio	Fin
Fase 1	13 / 10 / 2014	30 / 11 / 2014
Fase 2	01 / 12 / 2014	14 / 01 / 2015
Fase 3	15 / 01 / 2015	18 / 03 / 2015
Fase 4.1 (borrador memoria)	01 / 12 / 2014	18 / 03 / 2015
Fase 4.2 (dedicación exclusiva)	19 / 03 / 2015	02 / 07 / 2015

La última fase, que corresponde a la elaboración de la memoria del proyecto, se ha dividido en dos. En la primera parte, se irá completando un borrador de la memoria a lo largo del avance del resto del proyecto. Como es necesario que se realicen simultáneamente, la fase 4.1 se solapa con las fases 2 y 3. La fase 4.2 se ubica en un periodo de tiempo donde se supone que el resto del proyecto está terminado, y está previsto que todo el tiempo de dedicación sea exclusivamente para elaborar la memoria final del proyecto. Conviene destacar que todas las fases excepto la 4.2 deben estar terminadas para la fecha de salida a Alemania (01/04/15). También que en el periodo comprendido entre el 15/01/15 y el 15/03/15 (planificado para las fases 3 y 4.1) las horas de dedicación pueden ser mayores, ya que en mi caso personal no tengo clases ni exámenes en ese tiempo.

2.7. Hitos

Los principales hitos del proyecto se pueden deducir de las fechas críticas visibles en la distribución de las fases en el calendario.

H1 - Fin Fase 1 (30/11/14)

En esta fecha finaliza la fase 1 del proyecto, por lo que será necesario acordar una reunión para revisar lo realizado hasta el momento y revisar el plan para la siguiente fase.

H2 - Fin fase 2 (14/01/15)

Esta es la fecha estimada como frontera entre las fases 2 y 3. A partir de este momento empezará la parte más *dura* del proyecto, donde hay que trabajar con el Kinect y programar los detectores de posturas y movimientos. Al mismo tiempo, también habrá que documentar el progreso del proyecto. Será necesaria una reunión para resolver dudas y concretar la especificación del programa.

H3 - Terminar detector de posturas (20/02/15)

Esta es una fecha aproximada a la mitad de la Fase 2, donde debería estar terminada la parte de reconocimiento de posturas. A partir de aquí se debería empezar con el detector de movimientos. Es probable que sea necesario acordar una reunión hacia esta fecha.

H4 - Previo a salida a Friburgo (18/03/15)

La finalización de la fase 3 ha sido elegida dos semanas antes de la salida a Friburgo, porque esta fecha afecta directamente al proyecto. Para este momento la Fase 3 debería estar terminada y debería haber un borrador de la memoria que recoja más del 70 % del trabajo realizado hasta el momento. Debería haberse producido una reunión con los directores del proyecto, donde se haya revisado el estado actual del mismo. El único trabajo que debería quedar por finalizar es la memoria definitiva.

H5 - Proyecto terminado (02/07/15)

En esta fecha debería estar terminado todo el proyecto, incluyendo la memoria. Se enviará a los directores de proyecto una versión candidata a memoria definitiva, que, si fuera necesario, debería ser revisada y corregida antes del 02/09/15.

H6 - Solicitud de defensa del proyecto (24/07/15)

Matricular el proyecto en GAUR y solicitar la defensa antes de esta fecha.

H7 - Entrega del proyecto (02/09/15)

Subir la memoria definitiva a la plataforma ADDI antes de esta fecha.

H8 - Defensa del proyecto (16-18/09/15)

Rango de fecha donde podría realizarse la defensa del proyecto.

2.8. Riesgos

A continuación se detallan los diferentes riesgos que pueden comprometer el desarrollo y el éxito del proyecto.

2.8.1. Periodo de intercambio

El primer riesgo, y más evidente, es que la propia estancia de Erasmus en Friburgo dificulte terminar el proyecto. Por ello, se han ajustado los plazos de forma que estén adaptados a dedicar el menor tiempo posible desde Alemania, y poder centrar allí más esfuerzos en las clases del cuatrimestre. Este cuatrimestre tiene una dedicación estimada de 750 horas presenciales y no presenciales (en base a 30 créditos ECTS). El plazo estimado para la finalización del proyecto también es dos meses antes del plazo de entrega, para evitar que coincida con la fecha de regreso.

2.8.2. Pérdida de información

Para evitar la pérdida de información (en caso de avería del disco duro, borrado involuntario, etc.) se realizan copias de seguridad con el siguiente plan:

- Semanal: Copia de seguridad del código fuente y los documentos (incluyendo memoria del proyecto) en un fichero .tar.gz sincronizado en la nube. Las copias de seguridad se duplicarán ocasionalmente a una memoria externa.
- Mensual o según cambios de configuración: Copia de la imagen de disco del equipo donde se trabaja, incluyendo: particionado, sistema operativo, programas, documentos, código fuente.

Además de las copias de seguridad, se establecerá un sistema de control de versiones con Git para el código fuente y la memoria del proyecto. El control de versiones será local, y opcionalmente se podrá sincronizar en un repositorio remoto.

2.8.3. Averías

En todo momento se trabaja con un ordenador portátil, y en la mayor parte del proyecto también se utiliza un dispositivo externo (el Kinect) proporcionado por la facultad. Estos

dispositivos son herramientas de trabajo imprescindibles y pueden fallar. Si esto ocurre, deberán ser reemplazados por otros, lo cual puede suponer un tiempo añadido y retrasar el proyecto. Durante la estancia en Alemania, una avería puede suponer más problemas, debido a que la facultad no estará allí para prestar nuevo material que sustituya al averiado. Por eso, la mayor parte del proyecto y sobre todo la que más depende del correcto funcionamiento del dispositivo Kinect prestado por la facultad se desarrollará antes del periodo de intercambio.

3. CAPÍTULO

Conceptos básicos

Antes de empezar a trabajar en el reconocimiento de posturas, se dedicó un tiempo a aprender lo básico sobre ROS. El objetivo de este capítulo es documentar, como resumen, lo realizado durante esta etapa de aprendizaje y también servir como una guía rápida para futuros proyectos basados en ROS.

3.1. Robot Operating System (ROS)

ROS es un conjunto de software y bibliotecas para desarrollar aplicaciones para robots. En su sitio web [http://wiki.ros.org (ago. 2015)] ofrecen una descripción más detallada:

ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license.

Las herramientas de ROS ayudarán en la tarea de programar un detector de posturas de forma modular. También será útil para obtener lecturas del sensor Kinect ya procesadas como un esqueleto.

Los programas de ROS se estructuran en paquetes (*packages*) y nodos (*nodes*). Los paquetes contienen los nodos, que son programas ejecutables, y también los ficheros que estos nodos necesiten para ejecutarse. Los nodos se comunican entre ellos mediante mensajes, a través de dos canales diferentes: *topics* y *services*.

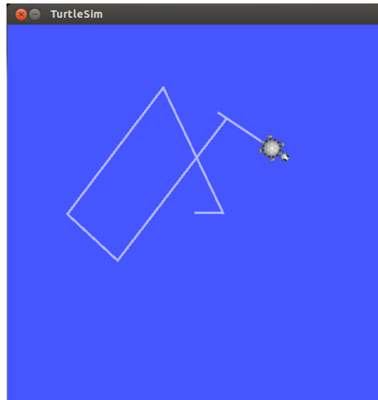


Figura 3.1: Ventana gráfica de turtlesim.

3.2. ROS Topics

Los temas (*topics*) son una forma que tienen los nodos de ROS de comunicarse entre ellos. En un tema puede haber un nodo que publica información (“habla"), otro nodo que se suscribe (“escucha"). Estas acciones son *publish* y *subscribe*. Esta forma de comunicación no hace que los nodos se comuniquen directamente, ni siquiera tienen por qué saber que el nodo que publica o recibe información está activo. La interacción se realiza siempre mediante un *topic*.

El paquete de ejemplo ‘turtlesim’ ayuda a entender cómo funcionan los *topics*. El nodo ‘turtlesim_node’ muestra una ventana gráfica (Fig. 3.1) con una tortuga en un espacio abierto, que puede actuar como un simulador muy simple de un robot en movimiento. El mismo paquete también contiene el nodo ‘turtlesim_teleop_key’, que nos permite controlar la tortuga con el teclado. Para ejecutar los dos nodos al mismo tiempo, ejecutaremos los siguientes comandos, cada uno en un terminal diferente.

```
$ roscore
$ rosrun turtlesim turtlesim_node
$ rosrun turtlesim turtle_teleop_key
```

Después, seleccionando el terminal donde se ha ejecutado ‘turtle_teleop_key’, podremos comprobar cómo la tortuga de la ventana que muestra ‘turtlesim_node’ se mueve cuando pulsamos una tecla. Esto se consigue gracias a los mensajes enviados entre un nodo y otro. ‘turtle_teleop_key’ publica los mensajes en un *topic*, mientras que ‘turtlesim_node’ se suscribe a este *topic*. Esto se puede visualizar gráficamente con la utilidad rqt_graph (Fig. 3.2).

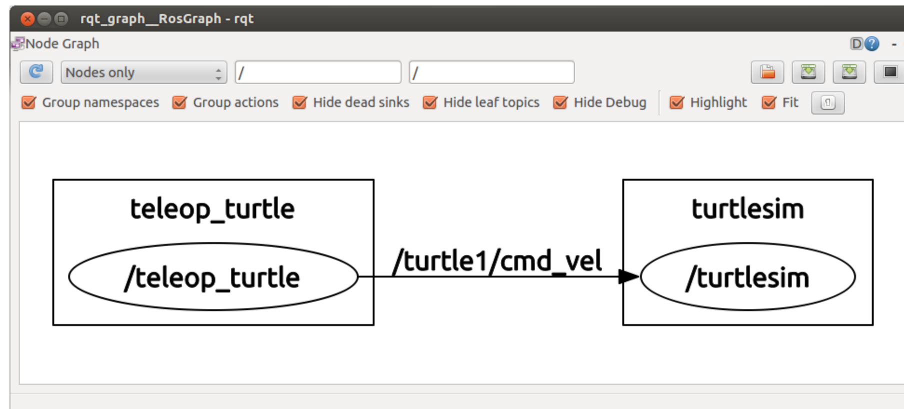


Figura 3.2: Representación gráfica de los nodos, el *topic*, y la comunicación entre nodos con rqt_graph.

```
$ rosrun rqt_graph rqt_graph
```

Un comando útil relacionado con los *topics* es 'rostopic'. Sirve para obtener información sobre los *topics*, y también para manipularlos, publicando datos en ellos. A continuación se muestran todas sus funciones:

```
$ rostopic -h
rostopic bw     display bandwidth used by topic
rostopic echo   print messages to screen
rostopic hz     display publishing rate of topic
rostopic list   print information about active topics
rostopic pub    publish data to topic
rostopic type   print topic type
```

En definitiva, 'rostopic' proporciona todas las utilidades necesarias para poder consultar y manipular la información en los *topics* de ROS.

3.3. Servicios y parámetros

Volviendo al nodo 'turtlesim_node' visto en el apartado anterior, se puede comprobar también el uso que este hace de servicios y parámetros de ROS. Al igual que tenemos el comando 'rostopic' para *topics*, el comando 'rosservice' proporciona utilidades para consultar y manipular servicios.

```
rosservice list         print information about active services
rosservice call         call the service with the provided args
rosservice type         print service type
rosservice find         find services by service type
rosservice uri          print service ROSRPC uri
```

Si ejecutamos el comando ‘rosservice list’ veremos que hay varios servicios activos. Nueve de ellos los proporciona turtlesim: reset, clear, spawn, kill, turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/get_loggers, y turtlesim/set_logger_level.

Veamos cómo llamar a estos servicios. En primer lugar, es necesario saber que las llamadas a un servicio pueden necesitar diferentes argumentos, dependiendo del tipo de servicio. Esto se puede consultar con el comando ‘rosservice type’.

Ejemplo 1:

```
$ rosservice type clear
std_srvs/Empty
$ rosservice type /clear | rossrv show
---
```

Ejemplo 2:

```
$ rosservice type /spawn
turtlesim/Spawn
$ rosservice type /spawn | rossrv show
float32 x
float32 y
float32 theta
string name
---
string name
```

En el primer ejemplo, el servicio clear es de tipo std_srvs/Empty, lo que significa que no tiene ningún argumento. En el segundo ejemplo, es de un tipo concreto, por lo que hay que comprobar cuales son los argumentos que pide. La forma de hacer esto es con el último comando (con un pipe a ‘rossrv show’). Lo que se muestra tras ejecutar el comando es

cuáles son los argumentos de spawn, y que el argumento ‘name’ es opcional. Por ejemplo, se puede llamar a ‘spawn’ de la siguiente forma:

```
$ rosservice call spawn 2 2 0.2 ""
```

En turtlesim aparecerá una nueva tortuga en la posición indicada. En el terminal de turtlesim, podremos observar:

```
[ INFO] [1423137719.509802004]: Spawning turtle [turtle2]
at x=[2.000000], y=[2.000000], theta=[0.200000]
```

En cuanto a los parámetros, estos se almacenan en el servidor de parámetros de ROS. Este servidor puede almacenar números enteros, números en coma flotante, booleanos, diccionarios y listas. Utiliza en lenguaje de marcas YAML como sintaxis. De forma similar a lo visto anteriormente, también está disponible el comando ‘rosparam’ para consultar y modificar parámetros:

```
$ rosparam list
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_macbux__38513
/rosversion
/run_id
```

Los parámetros /background_b, /background_g y /background_r definen el color de fondo de la ventana de turtlesim. Como ejemplo, vamos a probar a modificar uno de estos parámetros y ver lo que ocurre.

```
$ rosparam set background_r 150
$ rosservice call clear      # este comando es necesario para actualizar
                             la ventana
```

El color de fondo de la ventana pasará a ser morado (Fig. 3.3).

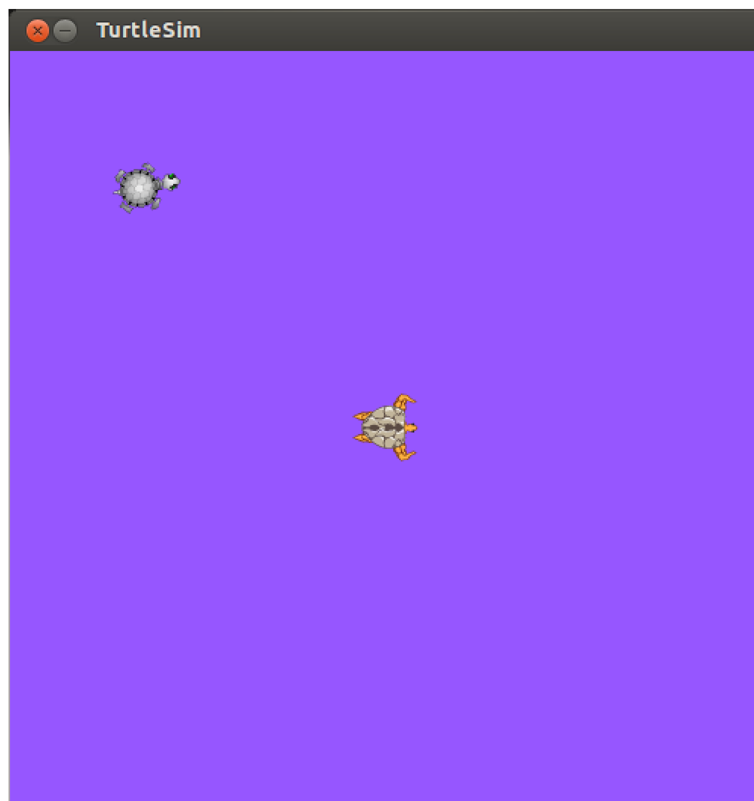


Figura 3.3: Visualización del cambio de color al modificar un parámetro de *turtlesim*.

4. CAPÍTULO

Configuración inicial

La instalación y configuración inicial del entorno de trabajo ha sido una parte necesaria para este proyecto. Para poder trabajar, es necesario que Kinect, NiTE, OpenNI, ROS y todas las dependencias necesarias estén instaladas, y que no haya problemas de compatibilidad.

4.1. Instalación de Ubuntu, ROS y Kinect

Se tuvo que realizar una instalación limpia de Ubuntu, en la que después se instaló ROS y OpenNI. La instalación de Ubuntu un proceso intuitivo, por lo que los detalles de la misma están fuera del alcance de esta memoria. Tan solo conviene indicar que el equipo utilizado ha sido un Macbook Air 6.2 de Apple, lo que ha provocado pequeñas diferencias respecto a una instalación en un PC estándar. En la instalación de ROS y OpenNI con Kinect sí que ocurrieron problemas que retrasaron el proyecto, los cuales se indican a continuación.

4.1.1. Instalación de ROS y OpenNI para Kinect en Ubuntu

En primer lugar, se intentó instalar Ubuntu 14.04, con ROS Indigo. Para instalar ROS y OpenNI (el driver que nos permite utilizar Kinect), se instalaron los siguientes paquetes:

- ros-indigo-desktop-full

- ros-indigo-openni-*

El siguiente comando ejecuta el controlador OpenNI en ROS:

```
$ roslaunch openni_launch openni.launch
```

Dicho comando debería ejecutarse sin problemas una vez instalados los paquetes, pero no ocurre así. Es un error conocido, en Ubuntu 14.04 y ROS Indigo no funciona correctamente con los paquetes de los repositorios.

Una solución encontrada para este error es instalar avin2-sensorkinect y NITE 1.5.2.23. En este caso, se probó a descargar los ficheros fuente, compilarlos e instalarlos manualmente. El error anterior desaparecía, pero aparecía un nuevo error, por el que OpenNI seguía sin funcionar correctamente:

```
Xiron OS failed to wait on event!
```

Este error no se pudo evitar en la instalación realizada en el Macbook Air 6.2. Se probó a realizar la misma configuración en otros dos equipos diferentes: en uno de ellos funcionaba correctamente, no aparecía ese mensaje de error. En otro de los equipos probados ocurría exactamente el mismo error, con el mismo mensaje. No fue posible averiguar el motivo de ese error, ni cómo solucionarlo, así que se decidió regresar a la versión anterior de ROS.

La versión anterior de ROS sólo está disponible para Ubuntu 12.04. Finalmente, la configuración elegida fue: Ubuntu 12.04, ROS Hydro y OpenNI. Como en el caso anterior, es necesario instalar los paquetes de ROS y OpenNI:

- ros-hydro-desktop-full
- ros-hydro-openni-*

Después, hay que quitar los módulos gspca:

```
$ sudo modprobe -r gspca_kinect
$ sudo modprobe -r gspca_main
```

Una vez hecho esto, ya debería funcionar OpenNI:

```
$ roslaunch openni_launch openni.launch
```

Esta vez sí que funciona correctamente. Para nuestro proyecto también necesitamos utilizar *skeleton traking*. Esto lo proporciona el nodo openni_tracker, el cual ya está instalado junto con todos los paquetes que empiezan por 'ros-hydro-openni-'. Al intentar ejecutarlo por primera vez, ocurre el siguiente error:

```
$ rosrun openni_tracker openni_tracker
[ERROR] [1415303771.788331222]: Find user generator failed:
This operation is invalid!
```

En este caso sí que ha sido posible solucionar el error. Para que openni_tracker funcione correctamente, hay que instalar NITE 1.5.2.21 o NITE 1.5.2.23 (se han probado las dos versiones, y ambas solucionan el problema). Es necesario descargar los ficheros fuente, compilarlos e instalarlos manualmente. Una vez hecho esto, los errores desaparecerán y el *skeleton tracker* será capaz de detectar a los usuarios frente al Kinect. La consola mostrará un texto cada vez que openni_tracker detecte nuevos usuarios.

4.2. Preparación de ROS

Antes de comenzar a trabajar en nuestro programa, debemos configurar correctamente el entorno de trabajo de ROS. A continuación se muestra el proceso de configuración y pruebas del entorno de ROS realizado tras la instalación.

4.2.1. Configuración del workspace

Los comandos de ROS se ejecutan en un terminal de Linux. ROS requiere que el entorno de ejecución del intérprete de comandos esté configurado correctamente. Si no lo está, no se reconocerá ningún comando ni se podrá ejecutar ningún programa compilado en el *workspace*. En primer lugar, es necesario importar el entorno de ROS (ejecutar el comando 'source' en terminal o añadirlo al fichero .bashrc).

```
source /opt/ros/hydro/setup.bash
```

En cuanto al entorno de trabajo (*workspace*), ROS ofrece dos opciones: *catkin* y *rosbuild*. *Catkin* es la opción recomendada ya que mejora la portabilidad respecto a *rosbuild* y está soportada por más paquetes. Por este motivo, elegimos trabajar con *catkin*. En primer lugar, iniciamos un entorno de trabajo.

```
$ mkdir -p ~/robotak/kinect/ros/src
$ cd ~/robotak/kinect/ros/src
$ catkin_init_workspace
```

Esto habrá creado el espacio de trabajo, sin ningún paquete pero con un fichero CMakeLists.txt. Este fichero permite compilar el contenido del espacio de trabajo:

```
$ cd ~/robotak/kinect/ros
$ catkin_make
```

No hay ningún paquete que compilar, pero se habrán creado los directorios ‘build’ y ‘devel’. Aquí se encuentran los ficheros necesarios para configurar el entorno de la consola.

```
source ~/robotak/kinect/ros/devel/setup.bash
```

Una vez introducidos ambos comandos ‘source’, o tras añadirlos al fichero .bashrc y reiniciar el intérprete de comandos, se puede comprobar si las variables de entorno de ROS están configuradas correctamente.

```
$ echo $ROS_PACKAGE_PATH
```

4.2.2. Compilar un *publisher* y *subscriber*

Para añadir nuestro propio código, en primer lugar es necesario crear un paquete dentro del *workspace* de Catkin. Creamos un paquete de nombre ‘pruebas’.

```
$ cd ~/robotak/kinect/ros/src
$ catkin_create_pkg pruebas std_msgs rospy roscpp
```

Una vez creado el paquete, en él vamos a escribir el código de un simple *publisher* y un *subscriber*. Estos efectuarán, de la forma más simple posible, las acciones *publish* y *subscribe* descritas en el capítulo anterior. El código de los siguientes programas está disponible en la documentación de ROS.

Código del *publisher*:

```
1   ros::init(argc, argv, "talker");
2   ros::NodeHandle n;
3   ros::Publisher chatter_pub = n.advertise<std_msgs::String>
4           ("chatter", 1000);
5   ros::Rate loop_rate(10);
6   int count = 0;
7   while (ros::ok())
8   {
9     std_msgs::String msg;
10    std::stringstream ss;
11    ss << "hello world " << count;
12    msg.data = ss.str();
13    ROS_INFO("%s", msg.data.c_str());
14    chatter_pub.publish(msg);
15    ros::spinOnce();
16    loop_rate.sleep();
17    ++count;
18  }
```

En primer lugar, se crea una instancia de NodeHandle. NodeHandle es el punto de acceso de las comunicaciones para cualquier nodo de ROS. En la siguiente línea, con la función advertise(), le estamos diciendo a ROS que queremos publicar en el *topic* ‘chatter’. El segundo argumento es el tamaño de la cola que vamos a utilizar. Esta cola se acabará llenando si se publican mensajes más rápido que la velocidad a la que puede recibirlos el receptor. En ese caso, los mensajes más antiguos serán descartados.

Dentro del bucle, que itera siempre que no haya ningún error, se genera un mensaje “ hello world ‘k’ ", donde k es el número de iteración. Este mensaje se publica a través del *topic* en ‘chatter_pub.publish(msg)’. La función spinOnce() que le sigue no es necesaria en este programa, pero es una buena práctica añadirla al final de cada nodo de ROS porque es la encargada de gestionar las llamadas a funciones *callback*.

Código del *subscriber*:

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

Lo primero que podemos ver en el código es la función *callback*. Esta es la función que hemos definido para ser llamada cuando el nodo reciba un mensaje.

En el programa principal, tenemos, al igual que en el *publisher*, el elemento NodeHandle, y la función subscribe() –en lugar de advertise()– para suscribirnos al *topic* ‘chatter’. El segundo argumento es el tamaño de la cola. Si los mensajes llegan tan rápido que la función *callback* no puede procesarlos, se empezarán a descartar mensajes viejos a partir de ese número. El último argumento es la función *callback*, encargada de procesar los mensajes cuando lleguen.

Por último, la función ros::spin() entrará en un bucle, donde estará esperando a recibir mensajes. Cuando llegue un mensaje, llamará a la función *callback* que hemos definido, y esta función lo procesará. El bucle volverá a iterar iterando para esperar otro nuevo mensaje, y seguirá iterando hasta que ros::ok() devuelva false.

Para compilar y ejecutar estos programas, creamos los ficheros listener.cpp y talker.cpp, con este código, en /robotak/kinect/ros/src/pruebas/src.

Añadimos las siguientes líneas al final del fichero CMakeLists.txt:

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener beginner_tutorials_generate_messages_cpp)
```

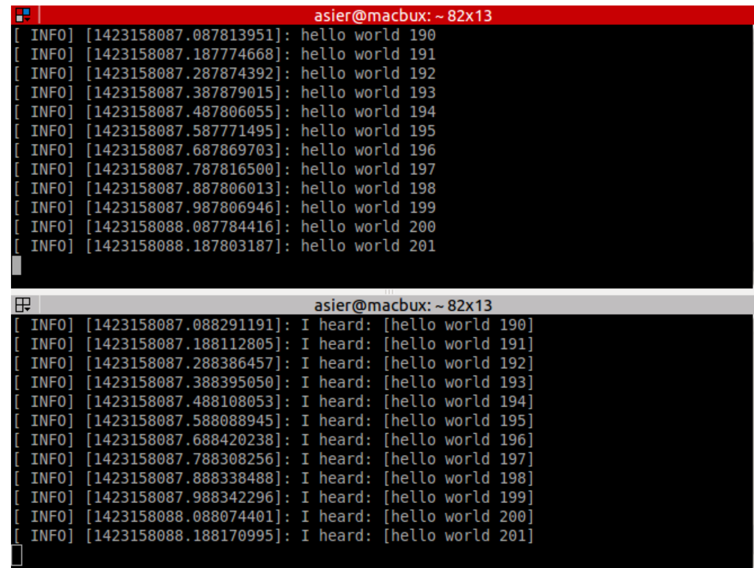


Figura 4.1: Un *publisher* y un *subscriber* simple en ejecución.

El comando catkin_make compilará todos los ficheros fuente del *workspace*:

```
# Compilar los nodos
$ cd ~/robotak/kinect/ros
$ catkin_make
```

Para ejecutar el *publisher* y el *subscriber*, ejecutar cada nodo en un terminal diferente:

```
# Terminal 1
$ roscore

# Terminal 2
$ rosrun beginner_tutorials talker

# Terminal 3
$ rosrun beginner_tutorials listener
```

El resultado de este proceso es una salida similar a la que se muestra en la Figura 4.1.

4.3. Kinect en ROS con OpenNI tracker

La función del nodo openni_tracker que hemos instalado anteriormente es publicar información sobre los puntos del esqueleto de una persona que se sitúe frente al sensor Kinect. Para ello utiliza 'tf'. 'tf' es el sistema que utiliza ROS para gestionar transformaciones en 3D.

Para acceder a la información de openni_tracker, basta con acceder al conjunto de transformaciones (en /tf) que este publica. Estas transformaciones son las siguientes:

- /head
- /neck
- /torso
- /left_shoulder
- /left_elbow
- /left_hand
- /right_shoulder
- /right_elbow
- /right_hand
- /left_hip
- /left_knee
- /left_foot
- /right_hip
- /right_knee
- /right_foot

Además, también diferencia a qué usuario pertenece cada transformación. Así, las transformaciones que publica en 'tf' son: /head_1, /torso_1, (...), para el usuario uno; /head_2, /torso_2, etc. para el usuario dos. Toda esta información se puede consultar sin que sea necesario empezar a programar. La herramienta rviz (Fig. 4.2) permite visualizar gráficamente la información proporcionada por el *tracker* de Kinect.

```
$ rosrun rviz rviz
```

En rviz, elegimos 'openni_depth_frame' en 'Pixed Frame', y después hacemos clic en 'add' y seleccionamos 'tf'. Ahora, siempre que haya un usuario frente al Kinect, podremos observar todas las posiciones de las que hace seguimiento openni_tracker, tomando como punto de referencia el marco de profundidad de OpenNI. Se pueden utilizar otros marcos de referencia que no varían al acercarse o alejarse del Kinect, por ejemplo, el torso del usuario.

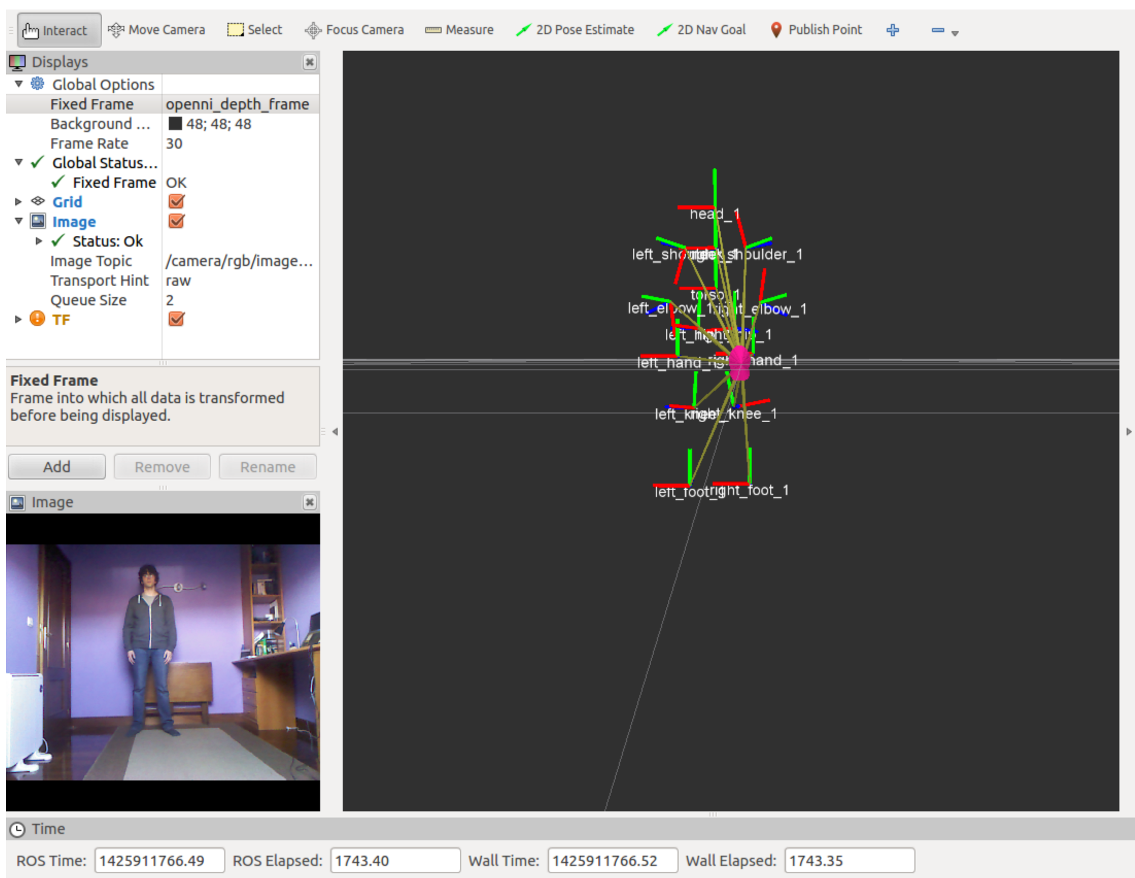


Figura 4.2: Visualización de 'tf' con openni_tracker y rviz.

4.3.1. Lectura de la información en C++

ROS proporciona la clase TransformListener para obtener la información de ‘tf’ desde un programa en C++. ‘tf’ es bastante potente, y permite obtener información de las transformaciones en cualquier instante de tiempo, hasta 10 segundos atrás desde el momento actual con precisión de milisegundos. Por ahora, nos interesa más obtener la información de las transformaciones en el instante 0, es decir, en el momento actual. Podemos hacer esto porque openni_tracker publica de forma ininterrumpida la información de un usuario que esté siguiendo. En otros casos, esto no sería posible y habría que buscar cuál es la transformación más reciente. Por tanto, nuestro caso es el más simple de todos: en todo momento, sólo tenemos que leer la información disponible en ‘tf’, en el instante 0.

Como ejemplo, el código a continuación lee la transformación de la cabeza respecto al torso e imprime las coordenadas x,y,z:

```
1  // Declarar una instancia de TransformListener
2    tf::TransformListener listener;
3
4  // Almacenar la lectura en un StampedTransform
5    tf::StampedTransform head_torso;
6
7  // Lectura de la transformacion en el momento actual
8    listener.lookupTransform("/torso_1", "/head_1", ros::Time(0),
9         head_torso);
10
11 // Mostrar las coordenadas x,y,z en salida estandar
12   std::cout << head_torso.getOrigin().x() << ","
13             << head_torso.getOrigin().y() << ","
14             << head_torso.getOrigin().z();
```

De la misma forma, se puede leer e imprimir cualquier otra transformación, o imprimir las coordenadas a fichero en lugar de por salida estándar.

5. CAPÍTULO

Reconocimiento de posturas

Una vez familiarizados con ROS y finalizada la configuración del entorno de trabajo, abordamos el objetivo principal de este proyecto: la detección y clasificación de posturas. Openni_tracker proporciona una gran ventaja para resolver esto: ya disponemos de la posición de todos los puntos del esqueleto al realizar lecturas desde el sensor Kinect. Teniendo en cuenta esto, y que las posturas a diferenciar estarán ya definidas, es posible plantear dos estrategias diferentes:

1. Estrategia Ad hoc. Algunas posturas son muy fáciles de reconocer y se pueden abordar de forma simple y directa. Por ejemplo: queremos diferenciar si un brazo está levantado o no (verdadero / falso). En ese caso, basta con comprobar la posición de un punto del brazo en el eje vertical respecto a la posición del torso o de los hombros. De esta forma, conseguimos diferenciar entre dos posturas (brazo levantado / bajado), pero cada vez que queramos diferenciar otra nueva postura habrá que diseñar una nueva solución. Además, esta solución debe ser compatible con todas las diseñadas anteriormente para el resto de posturas.

2. Machine Learning. Cuantas más posturas queramos añadir, más compleja es su clasificación. Desarrollar de forma manual un modelo que clasifique posturas en base a la información obtenida se vuelve una tarea inviable. Para poder tratar esta información, utilizaremos técnicas de aprendizaje supervisado. Dado un conjunto de datos representativo, dispondremos de un modelo de clasificación que obtenga una postura (clase) a partir de estos datos. Definir el conjunto de posturas previamente nos permitirá usar la estrategia de aprendizaje supervisado.

Es evidente que la segunda estrategia tiene más ventajas y es más flexible que la primera. Aún así, debido a la facilidad de su implementación, se ha empezado por diseñar un "detector" que hace un seguimiento de las dos manos para diferenciar si están levantadas o bajadas.

5.1. Un detector simple

El primer programa diseñado para la detección de posturas diferencia entre "mano levantada" y "mano bajada", de forma booleana, para la mano izquierda y la mano derecha. Esto lo hace capaz de diferenciar $2^2 = 4$ posturas diferentes.

Esto se ha hecho comprobando la posición relativa del marco de referencia de las manos respecto al marco de referencia del torso. Se ha definido el cero (mano a la altura del torso) como valor frontera entre "levantada" y "bajada". Se guarda el estado anterior de cada mano para poder detectar el momento en el que cambia. En otras palabras, en ese momento ocurre la acción de levantar o bajar la mano. En concreto:

$izquierda_levantada \longleftarrow false$
$derecha_levantada \longleftarrow false$
while $true$ **do**
  if $mano_izquierda_y - torso_y > 0$ **then**
    $izquierda_levantada \longleftarrow true$
  else
    $izquierda_levantada \longleftarrow false$
  if $mano_derecha_y - torso_y > 0$ **then**
    $derecha_levantada \longleftarrow true$
  else
    $derecha_levantada \longleftarrow false$
end

Algorithm 1: Detector de posturas simple

5.2. Aprendizaje supervisado

En esta ocasión el objetivo es clasificar un conjunto de posturas predefinido, pero cuyo clasificador no se puede elaborar de forma tan intuitiva como en el caso de la mano izquierda y derecha. A falta de un conjunto de posturas, en primer lugar, se han definido las posturas a clasificar.

5.2.1. Conjunto de clases

El conjunto de posturas a clasificar será el conjunto de clases o las salidas de nuestro clasificador. Se han propuesto las siguientes posturas para este proyecto, debido a que recoge posturas de diferentes características que podrían adaptarse a varios usos prácticos del clasificador. En la Figura 5.1 se pueden observar algunos ejemplos de estas posturas.

1. psi: Posición de “rendirse", necesaria para calibrar el *skeleton tracker*.
2. arms-wide: Los dos brazos abiertos, a lo ancho.
3. arms-front: Los dos brazos estirados hacia adelante.
4. arms-up: Los dos brazos levantados hacia arriba.
5. left-arm-left: Brazo izquierdo levantado hacia la izquierda.
6. left-arm-front: Brazo izquierdo levantado hacia adelante.
7. left-arm-up: Brazo izquierdo levantado hacia arriba.
8. right-arm-right: Brazo derecho levantado hacia la derecha.
9. right-arm-front: Brazo derecho levantado hacia adelante.
10. right-arm-up: Brazo derecho levantado hacia arriba.
11. hello: Levantando la mano derecha en señal de saludo.
12. left-leg-up: Pierna izquierda levantada.
13. right-leg-up: Pierna derecha levantada.
14. stand: De pie, frente al Kinect, con los brazos y las piernas relajados.
15. sit: Sentado.
16. bend: Inclinado.
17. lie: Tumbado.
18. none: Ninguna de las anteriores.

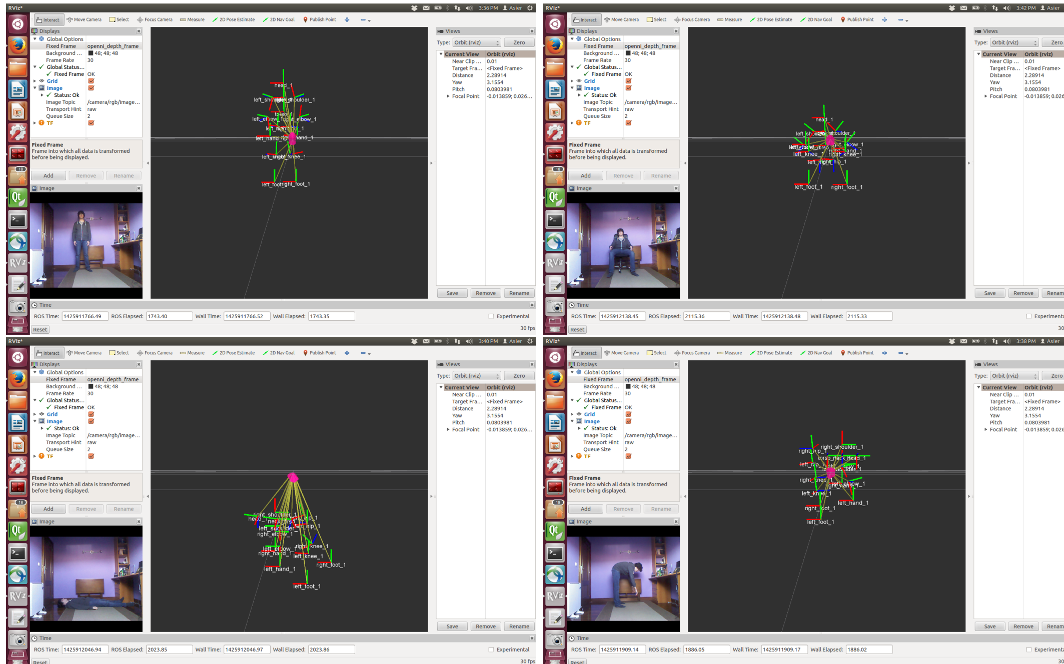


Figura 5.1: Ejemplos de posturas *stand*, *sit*, *lie* y *bend*.

Ya habíamos planteado la idea de que añadir más posturas a clasificar podría dificultar más la clasificación. Analizando las diferentes posturas que hemos definido, vemos que es posible clasificarlas en varios grupos independientes, lo que reduciría el número de posturas a clasificar.

1. Brazos: psi, arms-wide, arms-front, arms-up, left-arm-left, left-arm-front, left-arm-up, right-arm-right, right-arm-front, right-arm-up, hello.

2. Piernas: left-leg-up, right-leg-up.

3. Postura general: stand, sit, bend, lie.

Esta división en tres grupos también ofrece una nueva posibilidad: clasificar varias posturas al mismo tiempo, y que todas las clasificaciones sean correctas. Por ejemplo, una persona puede estar sentada y con los dos brazos en alto (Brazos: *arms-up*; Postura general: *sit*).

Obtenemos, por tanto, tres posibles clasificadores (uno por cada grupo). El más complejo, en el sentido de que tiene un mayor número de clases a diferenciar, parece ser el de 'brazos', por lo que este será el que utilicemos para la mayor parte de las pruebas. De todas formas, en experimentos posteriores, se ha descubierto que clasificar posturas más

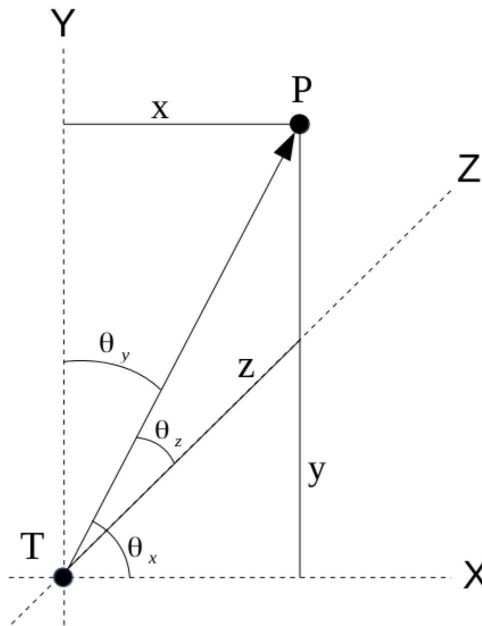


Figura 5.2: Proyección de los ángulos obtenidos a partir de la posición del torso T y la posición de un punto P del esqueleto.

generales como “sentado” o “tumbado” requiere de un vector de características de más dimensiones. Esto hace que sea más difícil clasificarlas correctamente, y clasificadores que funcionaban para posturas de brazos no son tan exactos con este otro caso.

5.2.2. Vector de características

La característica que se tiene en cuenta es el ángulo entre la posición del torso y la posición del resto de marcos de referencia del esqueleto. Se ha elegido el torso por dos motivos: es un punto central de las transformaciones obtenidas mediante OpenNI, y no afecta la distancia del usuario respecto al Kinect para la posición relativa a otros puntos del esqueleto.

En un espacio de tres dimensiones, un único ángulo no puede representar la orientación entre dos puntos. Es necesario conocer la dirección del vector entre ambos puntos, la cual representamos mediante tres ángulos respecto a los ejes X, Y, Z. La Figura 5.2 muestra cuáles son estos ángulos representativos (θ_x, θ_y, θ_z). Se calculan de la siguiente forma:

$\theta_x = arctan(\frac{y}{x})$
$\theta_y = arctan(\frac{x}{y})$
$\theta_z = arctan(\frac{x}{z})$

OpenNI proporciona suficientes puntos para obtener 14 orientaciones de vectores (tres ángulos por cada par de puntos), lo cual constituye un total de 42 ángulos. Es decir, el vector de características es de dimensionalidad 42.

5.3. Obtención de datos

Disponemos de un programa 'save_angle', que guarda vectores de 42 ángulos obtenidos a partir de sucesivas lecturas del *skeleton tracker*. El proceso propuesto para obtener datos es el siguiente:

1. Se introduce un nombre de postura por la entrada estándar. El programa espera 10 segundos.
2. Una persona se sitúa frente al Kinect realizando la postura introducida.
3. El programa hace 100 lecturas sucesivas de datos. Calcula cada ángulo y lo guarda en el fichero 'angledata'.

En caso de que hubiera dos personas, no sería necesario incluir una espera tan larga en el segundo paso. Esto se ha hecho para que una sola persona pueda trabajar recogiendo información desde el sensor Kinect. Siempre se recogen los datos asociados al usuario 1, por lo que, si se han detectado más usuarios con el sensor, los datos recogidos pueden ser erróneos. Además, este usuario tiene que haber sido detectado y calibrado correctamente para poder recoger datos. En caso contrario, el programa mostrará un error y esperará en un bucle hasta que 'openni_tracker' empiece a hacer seguimiento del usuario 1.

Las escrituras en el fichero 'angledata' son de tipo *append*: el fichero no se sobrescribe en ejecuciones consecutivas del programa. Esto es útil para ir añadiendo más información a los ficheros de datos cuando sea necesario. No importa el orden en el que se introduzcan los nombres de las posturas. Se pueden incluso repetir posturas ya introducidas anteriormente en el fichero, añadiendo más información al conjunto de datos.

Posteriormente importaremos los datos obtenidos en Weka, por lo que es útil que los datos se impriman en un formato fácil de procesar. Estas posturas clasificadas tienen que estar en un fichero de texto, en el formato de entrada soportado por Weka: ARFF. Un fichero de este formato tiene dos secciones: una cabecera, seguida de la sección 'data', que contiene las instancias de datos del fichero. En la cabecera se definen el conjunto de clases y el vector de características. Por ejemplo, una posible cabecera es la siguiente:

```
@relation posture
@attribute class {stand, sit, duck}
@attribute head_x real
@attribute head_y real
@attribute head_z real
@attribute neck_x real
@attribute neck_y real
(...)
```

A lo que le seguirán los datos recogidos, con la siguiente forma:

```
@data
stand,1.57155,-0.000756497,-0.0542928, (...)
stand,1.57155,-0.000756497,-0.0542928, (...)
stand,1.57121,-0.000418227,-0.0535468, (...)
stand,1.56893,0.00186669,0.221096,1.5708, (...)
stand,1.56936,0.00143966,0.182156,1.5708, (...)
stand,1.56925,0.00154483,0.209601,1.5708, (...)
```

El programa 'save_angle' imprime los datos en un formato similar al del ejemplo. Esto permite generar ficheros de Weka de forma muy sencilla: sólo hay que copiar y pegar esta salida bajo la cabecera (por motivos prácticos, incluyendo la línea @data al final de la cabecera, y dejando sólo las instancias de datos como salida de 'angledata'). Eso se puede realizar de forma rápida con la utilidad 'cat' de Linux.

```
$ cat cabecera datos > fichero_weka.arff
```

5.4. Preprocesado

Anteriormente hemos visto que el vector de características tiene 42 dimensiones en total. Esta alta dimensionalidad puede suponer un problema que debemos resolver en una primera fase de preprocesado. Utilizaremos el conjunto de clases 'brazos' como ejemplo y referencia para esta fase.

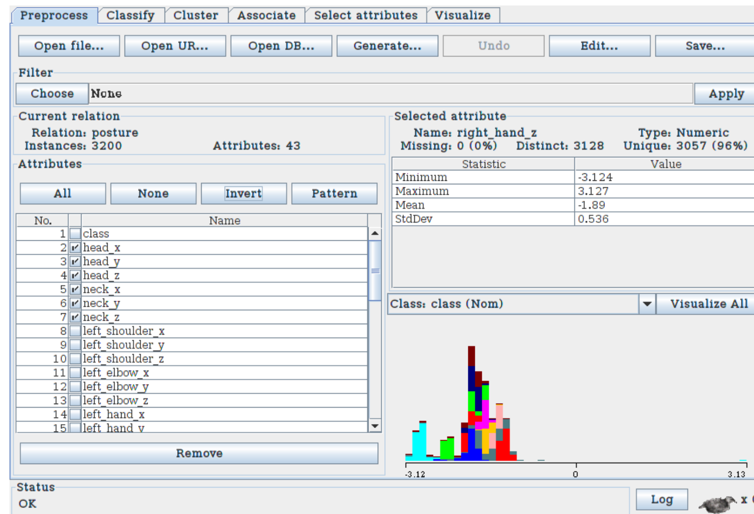


Figura 5.3: Selección de atributos en Weka.

5.4.1. Selección de atributos

Debido a que vamos a clasificar las posturas de los brazos, nos conviene utilizar puntos del esqueleto relacionados con ellos. En concreto: left_shoulder, left_elbow, left_hand, right_shoulder, right_elbow y right_hand. El resto de datos, que corresponden a diferentes partes del cuerpo, sólo incrementarán el número de atributos y añadirán ruido al clasificador. Además, pueden generar predicciones erróneas: si, por ejemplo, algunos usuarios tienden a inclinar la cabeza al levantar el brazo, se relacionará "cabeza inclinada" con "brazo levantado".

La selección de atributos se puede realizar de forma manual en Weka, en la pestaña 'Preprocess' (Fig. 5.3). A partir de lo anterior el resultado son 18 atributos.

5.4.2. Detección de *outliers*

Las lecturas obtenidas con el sensor Kinect pueden tener errores. En caso de que haya errores en la lectura, el *tracker* detecta los puntos del esqueleto muy alejados de su posición real, lo que produce que aparezcan *outliers*: instancias con valores muy alejados del rango de valores que debería tener su clase. Los *outliers* empeorarán la eficacia de los algoritmos de aprendizaje automático. Además, si detectamos muchos *outliers*, sabremos que hemos realizado lecturas erróneas, por lo que puede que haya que modificar el conjunto de datos u obtener uno nuevo.

Utilizaremos el filtro de rango intercuartílico ('InterquartileRange') para detectar *outliers*.

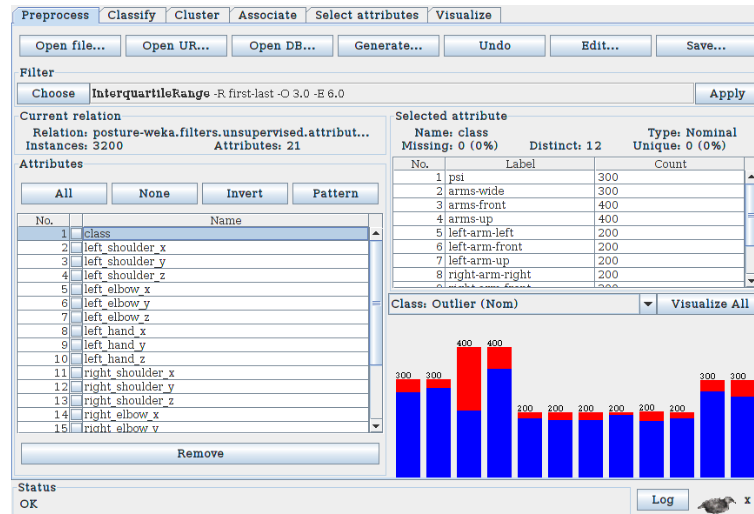


Figura 5.4: Distribución de *outliers*.

Este filtro crea dos nuevos atributos: 'Outlier' y 'ExtremeValue', con dos posibles valores: 'no' y 'yes'. El primer atributo, 'Outlier', permite observar dónde se han encontrado los *outliers*. Como se puede comprobar en la Figura 5.4, se han encontrado muchos *outliers* en la tercera clase: arms-front. Esto puede significar que muchas de las lecturas de esa clase en el conjunto de datos pueden ser erróneas, por lo que convendría revisar estos datos y volver a generar lecturas de la postura arms-front.

Antes de comenzar con la evaluación de clasificadores, nos puede interesar eliminar los *outliers* detectados. Para ello, disponemos del filtro 'RemoveWithValues', que permite seleccionar el índice del atributo (en nuestro caso, 20 - Outlier) y el índice del valor a eliminar (last - Yes). Una vez hecho esto ya no quedará ninguna instancia detectada como *outlier* (Fig. 5.5). Como no vamos a volver a utilizar los atributos 'Outlier' y 'ExtremeValue', ahora podemos eliminarlos manualmente, de la misma forma que anteriormente en el paso de selección de atributos.

5.5. Clasificadores

Weka proporciona muchos algoritmos de clasificación. Algunos de ellos son más eficientes en la generación del modelo, mientras que otros lo son en la clasificación. En nuestro caso, el algoritmo debe ser más rápido en el proceso de clasificación, porque el objetivo es obtener la postura en "tiempo real" al recibir nuevos datos del sensor Kinect.

Los árboles de decisión son rápidos (se basan en cadenas de comparaciones) y fáciles

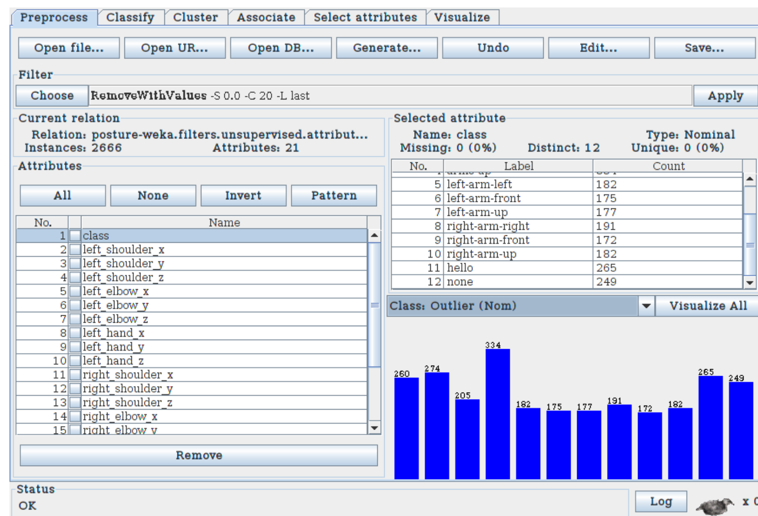


Figura 5.5: Conjunto de datos una vez eliminados los *outliers*.

de representar. Es por esto que, en caso de obtener buenos resultados, un generador de árboles de decisión sería una elección deseable.

El conjunto de datos utilizado para la evaluación de clasificadores tiene un total de 2666 instancias y 19 atributos. Las clases a clasificar son las de la categoría ‘brazos’.

5.5.1. J48

J48 es la implementación en Weka del algoritmo C4.5, un conocido algoritmo para generar árboles de decisión [Quinlan, 1993]. Este algoritmo utiliza la ganancia de la información para seleccionar los atributos que se dividirán en los nodos de decisión. Se ha aplicado el algoritmo con poda (*postpruning*). El resultado es un árbol como el que se puede observar en la Figura 5.6.

Resultados. Se ha evaluado el modelo generado en Weka utilizando validación cruzada (*cross-validation*). En la tabla 5.1 se pueden observar los resultados obtenidos.

5.5.2. J48graft

J48graft [Webb, 1999] es también el algoritmo C4.5, pero añade una técnica de “injerto” (*graft*) para hacer más preciso el árbol de decisión, insertando nuevos nodos de forma

Tabla 5.1: Resultados de la validación y matriz de confusión del clasificador J48

Instancias clasificadas correctamente	2656	99.6249 %
Instancias clasificadas incorrectamente	10	0.3751 %
Área de curva ROC mínima	0.994	
Área de curva ROC máxima	1	

```
  a   b   c   d   e   f   g   h   i   j   k   l   <-- classified as
259   0   0   1   0   0   0   0   0   0   0   0 |   a = psi
  0 273   0   1   0   0   0   0   0   0   0   0 |   b = arms-wide
  0   0 205   0   0   0   0   0   0   0   0   0 |   c = arms-front
  0   3   0 330   1   0   0   0   0   0   0   0 |   d = arms-up
  0   0   0   0 181   1   0   0   0   0   0   0 |   e = left-arm-left
  0   0   0   0   2 173   0   0   0   0   0   0 |   f = left-arm-front
  0   0   0   0   0   0 177   0   0   0   0   0 |   g = left-arm-up
  0   0   0   0   0   0   0 191   0   0   0   0 |   h = right-arm-right
  0   0   0   0   0   0   0   0 172   0   0   0 |   i = right-arm-front
  0   0   0   0   0   0   0   0   0 182   0   0 |   j = right-arm-up
  0   0   0   0   0   0   0   0   0   0 265   0 |   k = hello
  0   0   0   0   0   0   1   0   0   0   0 248 |   l = none
```

inductiva. Como consecuencia, los árboles de decisión son más grandes, pero algo más precisos que los generados con J48. Se ha aplicado el algoritmo con poda (*postpruning*).

Resultados – tabla 5.2. Utilizando el mismo método de validación que en el clasificador anterior, comprobamos que los resultados no son muy diferentes. El árbol generado es mucho más grande que el anterior: el tamaño del anterior era de 27 nodos, mientras que este es de 85. No parece necesario utilizar un árbol mucho más grande y complejo por una sola instancia más clasificada correctamente en el conjunto de pruebas. Además, el indicador de la curva ROC muestra resultados muy similares (excelentes) en ambos casos.

5.5.3. BFTree

Utiliza el algoritmo *best first* [Shi, 2007] para generar árboles de decisión. Este algoritmo expande el árbol de decisión a partir del *mejor* nodo disponible, mientras que C4.5 lo expande en profundidad. Esto hace que la poda del árbol pueda ser diferente a la de otros algoritmos. Se la ha aplicado poda (*postpruning*) al árbol generado con este algoritmo.

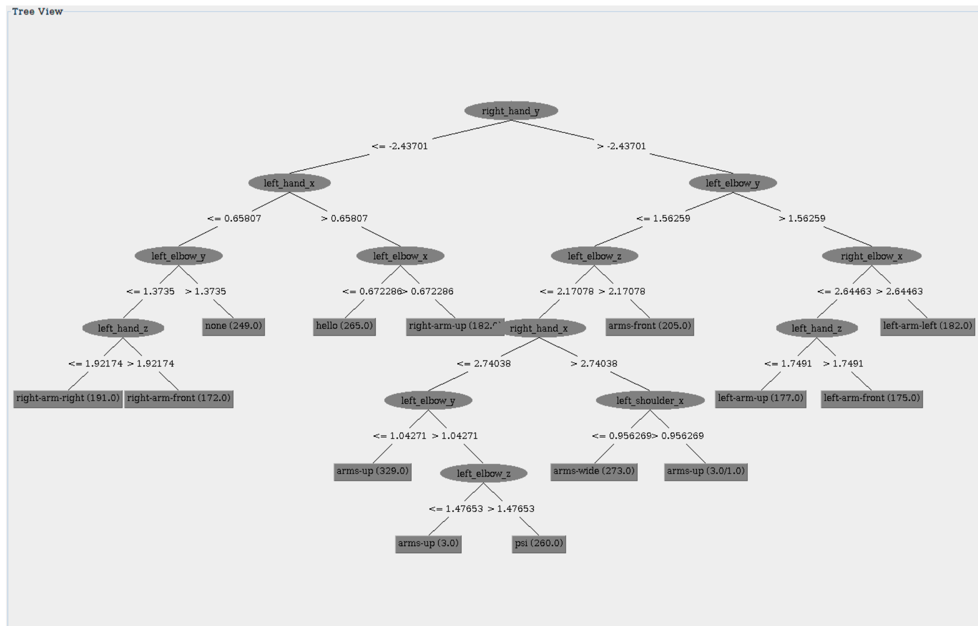


Figura 5.6: Árbol de decisión generado con J48.

Resultados – tabla 5.3. Respecto a la tasa de acierto y los indicadores ROC, los resultados han sido idénticos a los obtenidos con J48. Se pueden ver pequeñas diferencias en la matriz de confusión, es decir, los errores de clasificación se dan en diferentes clases.

5.5.4. Clasificadores bayesianos

Se han evaluado dos clasificadores diferentes: *Naive Bayes* [John and Langley, 1995] y redes bayesianas (con el clasificador BayesNet de Weka [`http://weka.sourceforge.net/doc.dev/weka/classifiers/bayes/BayesNet.html`]).

Resultados con Naive Bayes – tabla 5.4. Los resultados han sido muy similares a los obtenidos con los árboles de decisión. Aunque no sea muy difícil implementar un clasificador *Naive Bayes* en C++ y tampoco sería excesivamente lento, se ha decidido descartar su uso debido a que no ha tenido mejores resultados que los árboles de decisión, y llevaría más tiempo implementarlo.

Resultados con BayesNet – tabla 5.5. En el caso de la red bayesiana, los resultados han sido mejores que los obtenidos con *Naive Bayes* o árboles de decisión.

Tabla 5.2: Resultados de la validación y matriz de confusión del clasificador C4.5-graft

Instancias clasificadas correctamente	2657	99.6624 %
Instancias clasificadas incorrectamente	9	0.3376 %
Área de curva ROC mínima	0.994	
Área de curva ROC máxima	1	

```
   a   b   c   d   e   f   g   h   i   j   k   l   <-- classified as
 260   0   0   0   0   0   0   0   0   0   0   0 |   a = psi
   0 273   0   1   0   0   0   0   0   0   0   0 |   b = arms-wide
   0   0 205   0   0   0   0   0   0   0   0   0 |   c = arms-front
   0   3   0 330   1   0   0   0   0   0   0   0 |   d = arms-up
   0   0   0   0 181   1   0   0   0   0   0   0 |   e = left-arm-left
   0   0   0   0   2 173   0   0   0   0   0   0 |   f = left-arm-front
   0   0   0   0   0   0 177   0   0   0   0   0 |   g = left-arm-up
   0   0   0   0   0   0   0 191   0   0   0   0 |   h = right-arm-right
   0   0   0   0   0   0   0   0 172   0   0   0 |   i = right-arm-front
   0   0   0   0   0   0   0   0   0 182   0   0 |   j = right-arm-up
   0   0   0   0   0   0   0   0   0   0 265   0 |   k = hello
   0   0   0   0   0   0   1   0   0   0   0 248 |   l = none
```

5.5.5. Nearest neighbour

En este algoritmo, a cada nueva instancia a clasificar se le asigna la clase del vecino más cercano. Este clasificador es de tipo *lazy*, donde toda la carga computacional reside en la clasificación de instancias, y no en la generación del modelo. Además, el clasificador trabaja directamente sobre el conjunto de datos, o conjunto de entrenamiento. Esto implica que todo el conjunto de datos tiene que estar cargado en memoria durante la ejecución del clasificador. El procesamiento es mucho más costoso que en un árbol o un conjunto de reglas: dada una nueva instancia, el clasificador necesita calcular una distancia euclídea desde esta instancia hasta todas las instancias del conjunto de entrenamiento.

Resultados – tabla 5.6. Aunque sea un algoritmo muy costoso, se ha decidido evaluar este algoritmo y comprobar los resultados obtenidos. Sorprendentemente, a pesar de ser el clasificador más simple, el resultado ha sido mejor que con cualquier otro evaluado anteriormente.

Tabla 5.3: Resultados de la validación y matriz de confusión del clasificador BFTree

Instancias clasificadas correctamente	2656	99.6249 %
Instancias clasificadas incorrectamente	10	0.375 %
Área de curva ROC mínima	0.994	
Área de curva ROC máxima	1	

a	b	c	d	e	f	g	h	i	j	k	l	<-- classified as
259	0	0	0	0	0	1	0	0	0	0	0	a = psi
0	274	0	0	0	0	0	0	0	0	0	0	b = arms-wide
0	0	205	0	0	0	0	0	0	0	0	0	c = arms-front
1	2	0	329	0	0	0	0	0	1	0	1	d = arms-up
0	0	0	0	180	0	0	0	0	0	0	2	e = left-arm-left
0	0	0	0	0	175	0	0	0	0	0	0	f = left-arm-front
0	0	0	0	0	0	177	0	0	0	0	0	g = left-arm-up
0	0	0	0	0	0	0	191	0	0	0	0	h = right-arm-right
0	0	0	0	0	0	0	0	172	0	0	0	i = right-arm-front
0	0	0	0	0	0	0	0	0	182	0	0	j = right-arm-up
0	0	0	0	0	0	0	0	0	0	265	0	k = hello
1	0	0	0	1	0	0	0	0	0	0	247	l = none

5.5.6. Nearest neigbour generalizado

Weka también tiene un algoritmo similar al del vecino más cercano: *Nearest neighbour* con generalización [Martin, 1995]. Este algoritmo utiliza ejemplares generalizados no anidados. Estos ejemplares son hiper-rectángulos, que pueden interpretarse como reglas if-then que clasifican las instancias en clases. Estas reglas son muy eficientes y fáciles de integrar en un clasificador de posturas. Vistos los resultados obtenidos en el algoritmo del vecino más cercano, este algoritmo resulta especialmente interesante.

Resultados – tabla 5.7. Los resultados son similares pero ligeramente peores a los obtenidos al evaluar el algoritmo *nearest neighbour*.

5.5.7. Random Forest

Los árboles de decisión han dado buenos resultados anteriormente, pero estos resultados podrían mejorarse utilizando *random forests* [Breiman, 2001]. Este algoritmo genera árboles de decisión con atributos seleccionados aleatoriamente. Cada uno de estos árboles

Tabla 5.4: Resultados de la validación y matriz de confusión del clasificador Naive Bayes

Instancias clasificadas correctamente	2656	99.6249 %
Instancias clasificadas incorrectamente	10	0.375 %
Área de curva ROC mínima	0.998	
Área de curva ROC máxima	1	

```
   a   b   c   d   e   f   g   h   i   j   k   l   <-- classified as
 258   0   0   2   0   0   0   0   0   0   0   0 |   a = psi
   0 271   0   3   0   0   0   0   0   0   0   0 |   b = arms-wide
   0   0 203   2   0   0   0   0   0   0   0   0 |   c = arms-front
   1   0   0 333   0   0   0   0   0   0   0   0 |   d = arms-up
   0   0   0   0 182   0   0   0   0   0   0   0 |   e = left-arm-left
   0   0   0   0   0 175   0   0   0   0   0   0 |   f = left-arm-front
   0   0   0   0   0   0 177   0   0   0   0   0 |   g = left-arm-up
   0   0   0   0   0   0   0 191   0   0   0   0 |   h = right-arm-right
   0   0   0   0   0   0   0   0 172   0   0   0 |   i = right-arm-front
   0   0   0   0   0   0   0   0   0 182   0   0 |   j = right-arm-up
   0   0   0   1   0   0   0   0   0   1 263   0 |   k = hello
   0   0   0   0   0   0   0   0   0   0   0 249 |   l = none
```

generará su propio resultado al clasificar una nueva instancia, y, a través de una técnica como el voto de la mayoría, la clasificarán conjuntamente.

Resultados – tabla 5.8. Se ha generado un *random forest* de 100 árboles. Los resultados son mejores que con árboles de decisión únicos, y similares a los obtenidos con redes bayesianas o el algoritmo de *nearest neighbour* generalizado. Tan solo 2 de 2666 instancias han sido clasificadas incorrectamente, por lo que la matriz de confusión muestra muy pocos falsos positivos. La clase más problemática parece ser ‘arms-up’ junto con ‘psi’ y ‘arms-wide’. Todas ellas consisten en tener los dos brazos parcial o totalmente levantados, por lo que no son muy diferentes. Además, otros clasificadores también han mostrado falsos positivos en estas tres clases. Un experimento de clasificación en directo podría confirmarnos si, efectivamente, ocurren más errores al clasificar estas posturas.

Tabla 5.5: Resultados de la validación y matriz de confusión del clasificador BayesNet

Instancias clasificadas correctamente	2661	99.8125 %
Instancias clasificadas incorrectamente	5	0.1875 %
Área de curva ROC mínima	1	
Área de curva ROC máxima	1	

a	b	c	d	e	f	g	h	i	j	k	l	<-- classified as
260	0	0	0	0	0	0	0	0	0	0	0	a = psi
0	274	0	0	0	0	0	0	0	0	0	0	b = arms-wide
0	0	205	0	0	0	0	0	0	0	0	0	c = arms-front
1	1	1	330	1	0	0	0	0	0	0	0	d = arms-up
0	0	0	0	182	0	0	0	0	0	0	0	e = left-arm-left
0	0	0	0	0	175	0	0	0	0	0	0	f = left-arm-front
0	0	0	0	0	0	177	0	0	0	0	0	g = left-arm-up
0	0	0	0	0	0	0	191	0	0	0	0	h = right-arm-right
0	0	0	0	0	0	0	0	171	0	1	0	i = right-arm-front
0	0	0	0	0	0	0	0	0	182	0	0	j = right-arm-up
0	0	0	0	0	0	0	0	0	0	265	0	k = hello
0	0	0	0	0	0	0	0	0	0	0	249	l = none

Tabla 5.6: Resultados de la validación y matriz de confusión del clasificador Nearest Neighbour

Instancias clasificadas correctamente	2665	99.9625 %
Instancias clasificadas incorrectamente	1	0.0375 %
Área de curva ROC mínima	0.999	
Área de curva ROC máxima	1	

a	b	c	d	e	f	g	h	i	j	k	l	<-- classified as
260	0	0	0	0	0	0	0	0	0	0	0	a = psi
0	274	0	0	0	0	0	0	0	0	0	0	b = arms-wide
0	0	205	0	0	0	0	0	0	0	0	0	c = arms-front
0	1	0	333	0	0	0	0	0	0	0	0	d = arms-up
0	0	0	0	182	0	0	0	0	0	0	0	e = left-arm-left
0	0	0	0	0	175	0	0	0	0	0	0	f = left-arm-front
0	0	0	0	0	0	177	0	0	0	0	0	g = left-arm-up
0	0	0	0	0	0	0	191	0	0	0	0	h = right-arm-right
0	0	0	0	0	0	0	0	172	0	0	0	i = right-arm-front
0	0	0	0	0	0	0	0	0	182	0	0	j = right-arm-up
0	0	0	0	0	0	0	0	0	0	265	0	k = hello
0	0	0	0	0	0	0	0	0	0	0	249	l = none

Tabla 5.7: Resultados de la validación y matriz de confusión del clasificador Nearest Neighbour generalizado

Instancias clasificadas correctamente	2664	99.925 %
Instancias clasificadas incorrectamente	2	0.075 %
Área de curva ROC mínima	0.997	
Área de curva ROC máxima	1	

a	b	c	d	e	f	g	h	i	j	k	l	<-- classified as
260	0	0	0	0	0	0	0	0	0	0	0	a = psi
0	274	0	0	0	0	0	0	0	0	0	0	b = arms-wide
0	0	205	0	0	0	0	0	0	0	0	0	c = arms-front
0	1	0	332	0	0	0	0	0	0	0	1	d = arms-up
0	0	0	0	182	0	0	0	0	0	0	0	e = left-arm-left
0	0	0	0	0	175	0	0	0	0	0	0	f = left-arm-front
0	0	0	0	0	0	177	0	0	0	0	0	g = left-arm-up
0	0	0	0	0	0	0	191	0	0	0	0	h = right-arm-right
0	0	0	0	0	0	0	0	172	0	0	0	i = right-arm-front
0	0	0	0	0	0	0	0	0	182	0	0	j = right-arm-up
0	0	0	0	0	0	0	0	0	0	265	0	k = hello
0	0	0	0	0	0	0	0	0	0	0	249	l = none

Tabla 5.8: Resultados de la validación y matriz de confusión del clasificador Random Forest

Instancias clasificadas correctamente	2662	99.85 %
Instancias clasificadas incorrectamente	4	0.15 %
Área de curva ROC mínima	1	
Área de curva ROC máxima	1	

a	b	c	d	e	f	g	h	i	j	k	l	<-- classified as
260	0	0	0	0	0	0	0	0	0	0	0	a = psi
0	274	0	0	0	0	0	0	0	0	0	0	b = arms-wide
0	0	205	0	0	0	0	0	0	0	0	0	c = arms-front
1	1	0	330	1	0	0	0	0	1	0	0	d = arms-up
0	0	0	0	182	0	0	0	0	0	0	0	e = left-arm-left
0	0	0	0	0	175	0	0	0	0	0	0	f = left-arm-front
0	0	0	0	0	0	177	0	0	0	0	0	g = left-arm-up
0	0	0	0	0	0	0	191	0	0	0	0	h = right-arm-right
0	0	0	0	0	0	0	0	172	0	0	0	i = right-arm-front
0	0	0	0	0	0	0	0	0	182	0	0	j = right-arm-up
0	0	0	0	0	0	0	0	0	0	265	0	k = hello
0	0	0	0	0	0	0	0	0	0	0	249	l = none

5.6. Experimentos con posturas generales

Vistos los resultados obtenidos clasificando posturas concretas con los brazos, se decidió experimentar si los resultados serían tan buenos con posturas más generales. Se ha elegido el siguiente grupo de posturas: de pie, sentado, inclinado y tumbado. Se pueden encontrar trabajos anteriores [Le et al., 2013] que clasifican estas mismas posturas.

Se ha decidido realizar una evaluación de todos los algoritmos anteriores: J48, *nearest neighbour*, *nearest neighbour* generalizado, red bayesiana y *random forest*. En este caso, no se ha realizado una selección de atributos previamente, porque no se clasifica la postura de un grupo de extremidades en concreto, sino de todo el cuerpo. El filtro de rango intercuartílico tampoco ha sido útil para detectar *outilers*, ya que, si se hubiera aplicado como en el caso anterior, se habría eliminado gran parte del conjunto de muestras (1040 de 2551, cerca del 40 %).

5.6.1. Evaluación de clasificadores

Se han evaluado los diferentes clasificadores utilizando validación cruzada con 10 rodajas (*10 fold cross-validation*). En la tabla 5.9 se muestran las tasas de acierto obtenidas con cada clasificador.

Tabla 5.9: Tasas de acierto clasificando posturas generales

Clasificador	Tasa de acierto
J48	97.9616 %
Nearest Neighbour	97.9616 %
Nearest Neighbour generalizado	95.7272 %
Red bayesiana	98.2242 %
Random forest	**99.556 %**

El clasificador *random forest* obtiene la mayor tasa de acierto. Se ha podido observar, además, que el indicador de área de la curva ROC es 1 (el mejor valor posible) con este clasificador, para todas las clases (stand, sit, bend, lie).

En la tabla 5.10, puede comprobar que la postura *stand* (de pie) ha sido clasificada correctamente en todos los casos. El resto de posturas muestran algún error de clasificación, pero en cantidades despreciables. Si se dieran los mismos resultados en clasificación en directo, este sería un excelente clasificador.

Tabla 5.10: Matriz de confusión del clasificador Random Forest con posturas generales

```
   a   b   c   d   <-- classified as
 251   0   0   0 |   a = stand
   0 198   1   1 |   b = sit
   0   0 199   1 |   c = bend
   1   0   0 249 |   d = lie
```

5.7. Clasificación en directo

Se han evaluado en directo clasificadores como: árboles de decisión comunes (generados con el algoritmo C4.5), y también *random forests*. Para ahorrar tiempo y recursos, esta evaluación se ha realizado de forma imprecisa: no se ha calculado la tasa de acierto, verdaderos o falsos positivos, ni la matriz de confusión. El experimento consiste en lo siguiente: un usuario se sitúa frente al Kinect y un monitor, y el monitor muestra el resultado obtenido (la postura clasificada). El sistema clasifica las posturas del usuario en bucle, ejecutando de nuevo el clasificador tras detenerse 0.1 segundos. De esta forma, el usuario debería observar en pantalla la postura en la que se encuentra, según el clasificador.

Los resultados observados han sido satisfactorios: se ha realizado el mismo experimento con al menos 5 usuarios diferentes y los resultados observables se correspondían con la postura que el usuario realizaba. Ocasionalmente se mostraban resultados erróneos, algunos podrían deberse a lecturas erróneas del esqueleto, otros a un modelo sobreajustado. Posturas como ‘sentado’ o ‘tumbado’ eran más problemáticas que las posturas de brazos. Estos resultados erróneos eran más frecuentes en árboles de decisión comunes que en *random forests*, por lo que la hipótesis de errores por modelo sobreajustado es más sólida. El componente aleatorio de los *random forests* ha ayudado a corregir este problema de sobreajuste.

5.8. Conclusiones

Los resultados obtenidos durante la evaluación con validación cruzada han sido muy satisfactorios con todos los clasificadores. El algoritmo *nearest neighbour* ha sido el que mejores resultados ha obtenido en el caso de las posturas de brazos. Es un clasificador más lento que un árbol de decisión, y además necesita disponer de la base de datos de

muestras en todo momento (por lo que requiere más espacio de almacenamiento para los modelos). Si queremos conseguir una eficiencia comparable a la de los árboles de decisión, una opción es utilizar el *nearest neighbour* generalizado, donde el clasificador obtenido es un conjunto de reglas similar a un árbol de decisión. De todas formas, hemos comprobado que los resultados son ligeramente menos precisos con este algoritmo, y no mejoran la eficacia de los árboles de decisión.

Los árboles de decisión son clasificadores rápidos y fáciles de implementar. Además, han dado buenos resultados utilizando algoritmos de generación basados en la ganancia de información.

Los random forests han mejorado la eficacia de los árboles de decisión, sobre todo tras observar los resultados de la clasificación en directo. En cuanto a su complejidad de tiempo, es la misma que la de un sólo árbol de decisión multiplicada por un factor constante. En el caso de 100 árboles, requiere 100 veces más tiempo, por lo que en la práctica sí que se nota una clasificación más lenta. De todas formas, los árboles de decisión son lo suficientemente rápidos como para que esta sobrecarga no resulte excesiva.

Ha sido necesario implementar los siguientes componentes, tanto para realizar los experimentos de clasificación en directo como para el programa principal de detección de posturas:

1. Representación de árboles de decisión genéricos.

2. Lector de árboles de decisión desde ficheros de texto. Obtiene una representación interna del árbol de decisión a partir de un fichero de texto.

3. Representación de *random forests* a partir de árboles de decisión genéricos. Esta representación desempeñará las funciones de clasificador.

6. CAPÍTULO

Reconocimiento de movimientos

El sistema de detección de posturas que acabamos de diseñar es el primer paso para nuestro último objetivo: la detección y el seguimiento de movimientos. Haciendo uso de lo que ya tenemos, se puede plantear el siguiente mecanismo para comprobar si un movimiento se ha realizado:

1. Detectar postura inicial
2. Detectar si el usuario está en movimiento o parado
3. Detectar postura final

Si nuestro programa es capaz de detectar estos tres eventos, somos capaces de detectar si un usuario completa un movimiento correctamente, aunque sólo sea comprobando la posición inicial y final. El programa tendrá que almacenar en memoria el estado actual del usuario: esto se puede plantear como una sencilla máquina de estados finita, donde las transiciones entre estados se disparan al coincidir una postura junto con un usuario parado, o al pasar de estar parado a estar en movimiento.

La detección de posturas nos permite comprobar si el usuario se encuentra en la postura inicial o final. Por tanto, el siguiente paso es conseguir comprobar si el usuario está en movimiento o no.

6.1. Detección de movimiento

Como vimos en el capítulo 4, ROS permite obtener la posición de las partes del esqueleto de las que OpenNI hace seguimiento a través de 'tf', hasta 10 segundos atrás en el tiempo. En cada punto podemos obtener dos posiciones en diferentes instantes de tiempo, lo que nos proporcionará la velocidad media de ese periodo de tiempo:

$$v = \frac{\Delta s}{\Delta t}$$

Definiremos un valor frontera m para v, de forma que ese valor indique si un punto está en movimiento o está parado. Podemos decir que un usuario está en movimiento si, en alguno de sus puntos, v es mayor que m, es decir, si $max(v_i) > m \mid i = 1..n$, donde $1..n$ son los índices de los puntos del esqueleto. El valor de m es un parámetro que debe ser ajustado para obtener buenos resultados. Después de realizar diferentes pruebas, se ha elegido el valor $m = 0{,}2$ m/s.

Habrá un v_i asociado a cada uno de los puntos del esqueleto. Si nos interesa saber si entre todo el cuerpo hay algo en movimiento, tendremos que tener en cuenta todos los puntos. Aún así, muchas veces queremos saber sólo si una parte del cuerpo está en movimiento, por ejemplo, el brazo izquierdo. En ese caso, sólo comprobaremos la velocidad de left_elbow y left_hand. Con este subconjunto más pequeño haremos lo mismo: la velocidad máxima entre las dos indicará la velocidad del brazo.

Obtendremos la velocidad media desde hace t segundos hasta el momento actual.

$$v([-t,0]) = \frac{dist(s_0,s_t)}{t}$$

Donde *dist* es la función de la distancia euclídea entre dos puntos, s_0 es la posición actual y s_t es la posición hace t segundos.

Esta forma de calcular la velocidad media es correcta, pero solamente utiliza dos mediciones de la posición para calcular la velocidad de movimiento. 'tf' almacena las posiciones de varios instantes diferentes, y en intervalos muy pequeños. Podemos aprovechar esta información para calcular la velocidad de manera más precisa y ofrecer una respuesta más rápida. Para ello, partimos $[-t,0]$ en dos intervalos de tiempo de misma longitud: $i_1 = [-t,-t/2]$, $i_2 = [-t/2,0]$. Para cada uno de ellos, calculamos su velocidad media. También calculamos la velocidad media global: $v_g = v([-t,0])$. Después, calculamos la media ponderada entre los tres resultados obtenidos:

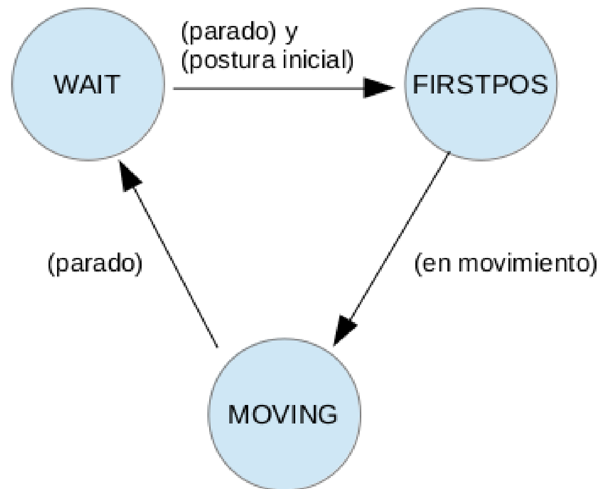


Figura 6.1: Esquema básico de los estados y transiciones.

$$v_m = \frac{0{,}25*v(i_1)+0{,}75*v(i_2)+v_g}{(0{,}25+0{,}75+1)}$$

El objetivo de la media ponderada es hacer que la velocidad más reciente y la global tengan más peso que la más antigua (que corresponde a la primera mitad del espacio de tiempo). De esta forma, el tiempo de reacción del sistema mejorará, pero no se perderá la información proporcionada por el primer intervalo.

6.2. Estados de movimiento

Siguiendo el esquema descrito de postura inicial, movimiento y postura final, se ha diseñado una máquina de estados para representar el estado actual del movimiento, las transiciones entre estados y las condiciones de transición.

En la Figura 6.1 se puede ver un diagrama básico para la máquina de estados. Este autómata no está diseñado para producir ninguna salida, simplemente para representar el estado del sistema. Las comprobaciones adicionales se realizarán en los instantes inmediatamente posteriores a una transición de estados. Por ejemplo, entre el estado *moving* y *wait* querremos saber, además de que el usuario se ha parado, si lo ha hecho en la postura final. Por tanto, habrá una comprobación más durante esta transición de estados: comprobar si la postura actual es igual a la postura final.

6.3. Precisión del movimiento

Se han tenido en cuenta dos enfoques diferentes para calcular la precisión de un movimiento realizado por el usuario. El primero y más sencillo es calcular la precisión de la postura final, es decir, tener una imagen estática conocida de la postura final y compararla con la que ha realizado el usuario. El otro enfoque ha sido tomar el movimiento como una sucesión de posturas intermedias, donde el usuario debe pasar por cada una de ellas para realizar el movimiento correctamente.

6.3.1. Postura final

Una postura estática es definida como un conjunto de ángulos entre torso y el resto de puntos del esqueleto. En el caso del brazo, consistiría en 6 ángulos: respecto a los ejes x, y, z, para el codo y la mano. Para comparar esta postura estática, la postura final es conocida para el programa: la carga desde un fichero al iniciarse. Cuando el usuario detiene el movimiento (en la transición de estados *moving* $\rightarrow$ *wait*), se compara la postura en la que se ha detenido con la postura que el programa conoce. Si ambas posturas coinciden, significa que el usuario ha alcanzado la postura final.

La forma de obtener la precisión de la postura final, comparándola con la postura conocida, es tomar el vector de ángulos asociado a cada postura como un punto en el espacio euclídeo. Después, se calculará la distancia euclídea entre los puntos asociados a la postura final y la postura conocida. La idea es obtener, a partir de esto, una precisión relativa (por ejemplo, 80 %), para lo que es necesario saber cuál es la distancia máxima entre estos dos puntos. Al estar obtenidos a partir de ángulos, las posiciones de los puntos estarán acotadas entre $[-\pi, \pi]$. En este espacio, la máxima distancia posible entre dos puntos en una dimensión es de 2π. Generalizando a n dimensiones:

$$d_{max} = \sqrt{(2\pi)^2 + (2\pi)^2 + (2\pi)^2 + (\ldots)} = \sqrt{n * (2\pi)^2} = 2\pi\sqrt{n}$$

Cuando obtengamos una distancia entre la postura del usuario y la postura final, podremos calcular el error relativo (o la *imprecisión*), de la siguiente forma:

$$e_r = \frac{dist}{2\pi\sqrt{n}}$$

El dato que se le mostrará al usuario será el inverso, es decir, la *precisión* de la postura final.

$$p = 1 - e_r$$

6.3.2. Posturas intermedias

Para un correcto reconocimiento del movimiento, es necesario que el programa sea capaz de reconocer los estados intermedios de este movimiento. Estos estados intermedios pueden verse como una sucesión de posturas estáticas, donde el usuario pasará por ellas cuando esté realizando el movimiento. Lo primero que se ha de resolver es cuáles son estas posturas intermedias.

Estas posturas intermedias del movimiento se han reconocido mediante *clustering*. Los elementos a dividir entre clusters son posturas estáticas (muchas de ellas), obtenidas a partir de usuarios que estaban realizando el movimiento a reconocer. Estas posturas han sido grabadas a lo largo de todo el proceso del movimiento.

El algoritmo elegido para *clustering* ha sido K-Means. El motivo para elegir este algoritmo es, principalmente, que genera un centroide para cada cluster. Este centroide es un punto en el espacio euclídeo, donde las dimensiones son los ángulos de cada punto del esqueleto (exactamente lo mismo que una postura estática). Esto permite calcular la precisión relativa de una postura respecto al centroide, de la misma forma que hemos hecho en el apartado anterior con las posturas finales.

6.3.3. Clustering en Weka

Importamos en Weka un fichero de datos con diferentes posturas estáticas, que han sido obtenidas a partir de un usuario que estaba realizando el movimiento a modelar. Utilizando la interfaz gráfica de Weka, en la pestaña ‘Cluster’, seleccionamos el algoritmo ‘SimpleKMeans’. Con este algoritmo, es necesario seleccionar también el número de clusters a generar. En el caso del movimiento para el que más pruebas se han realizado (“brazo derecho”), se obtuvieron resultados satisfactorios con 8 clusters.

En la salida del clusterer (*clusterer output*) obtenemos los centroides de cada cluster y su posición. Esta es toda la información que necesitamos para nuestro modelo de posiciones intermedias de un movimiento.

En la Figura 6.2 se puede observar la apariencia de la herramienta para *clustering* de Weka, junto con una salida obtenida a partir del algoritmo K-Means.

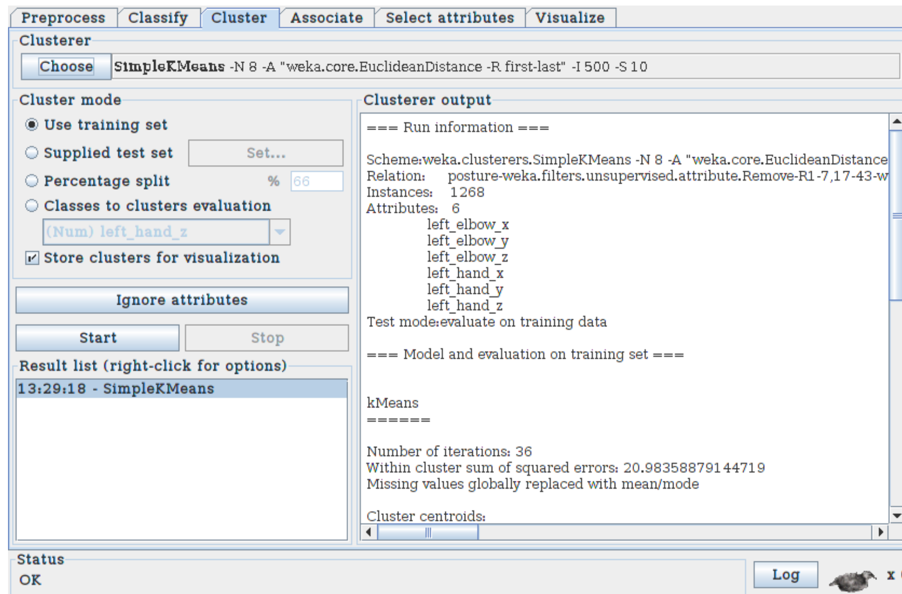


Figura 6.2: Clustering en Weka con el algoritmo K-Means.

6.3.4. Precisión de posturas intermedias

Dados los centroides de los clusters para posturas intermedias, se ha utilizado el siguiente algoritmo para calcular la precisión del movimiento:

while *en movimiento* **do**
    inicializar *maximos*;
    leer *postura*;
    $c \longleftarrow obtener_cluster(postura)$;
    if $precision(c, postura) > maximos[c]$ **then**
        $maximos[c] \longleftarrow precision(c, postura)$;
    end
end
$precision_movimiento \longleftarrow media_aritmetica(maximos)$;

Algorithm 2: Precisión del movimiento.

En este algoritmo, la función *precisión* calcula la precisión de la postura respecto al centroide del cluster basándose en la distancia, como hemos hecho para posturas estáticas. Se reconoce cuál es el cluster de la postura leída a partir del centroide más cercano (es decir, a mínima distancia). En este algoritmo, la precisión se verá influida por dos factores importantes: la cantidad de clusters reconocidos a partir del movimiento (es decir, el número de clusters que coinciden con alguna de las posturas que forman parte del movimiento), y lo cerca que lleguen a estar estas posturas del centroide del cluster en el que han sido

clasificadas. La cantidad de clusters influye de la siguiente forma: si un cluster nunca es reconocido a lo largo del movimiento, su distancia máxima será 0. Esto disminuirá el resultado de la media aritmética del último paso del algoritmo.

7. CAPÍTULO

Implementación

En capítulos anteriores se han expuesto diferentes procesos de *machine learning* y el proceso mediante el cual se clasificaban y reconocían posturas y movimientos. En este capítulo mostraremos varios detalles acerca de la implementación de este último proceso de reconocimiento. En primer lugar, mostraremos cómo se han implementado los árboles de decisión en nuestro programa para poder importar modelos generados con Weka. A continuación, proporcionamos varios detalles sobre la implementación en ROS, en concreto, sobre los módulos en los que se ha dividido el programa y cómo se realiza la comunicación entre ellos. Por último, mostramos una interfaz gráfica que ha sido desarrollada para visualizar la detección de posturas y movimientos.

7.1. Árboles de decisión

Los árboles de decisión están representados en la clase 'modeltree' [A.1]. Esta clase representa los árboles de forma recursiva: una instancia de 'modeltree' puede tener hijos de tipo 'modeltree' a la izquierda y/o derecha, o ser un nodo hoja. Los nodos hoja incluyen una clase como valor (por ejemplo, 'sentado'). El resto de nodos incluyen un atributo y un valor a comparar. Un 'modeltree' tiene, además, un método 'classify' (clasificar) para clasificar un vector de características de forma recursiva. En el Algoritmo 3 se puede observar una representación en pseudocódigo de este método de clasificación.

Un *random forest* basado en la clase 'modeltree' es lo siguiente: la clase 'randomforest' [A.2] consiste en un vector de clases 'modeltree', que son las raíces de los árboles de

Algoritmo clasificar (árbol, vector_entrada):
if no_hoja **then**
  if $árbol.valor < vector_entrada[árbol.atr]$ **then**
    return clasificar(árbol.$hijo_izquierdo, vector_entrada$);
  else
    return clasificar(árbol.$hijo_derecho, vector_entrada$);
end
return árbol.$clase$;

Algorithm 3: Algoritmo clasificar

decisión. Después, la clasificación se hace mediante voto de la mayoría: el resultado mayoritario entre los árboles de decisión será el resultado del *random forest*.

Esta representación interna de árboles de decisión y *random forests* debe ser construida previamente. Estos modelos están almacenados en ficheros, cuyo formato es exactamente el de la salida de Weka. Como ejemplo, el siguiente sería un fragmento de uno de estos ficheros:

```
RandomTree
==========
left_knee_z < 2.28
|   right_hand_z < -0.58
|   |   left_foot_z < 0.66
|   |   |   right_foot_x < -2.09
|   |   |   |   left_hand_z < -0.69 : bend (152/0)
|   |   |   |   left_hand_z >= -0.69
|   |   |   |   |   left_hand_z < 1.38
|   |   |   |   |   |   head_x < 1.57 : lie (6/0)
|   |   |   |   |   |   head_x >= 1.57 : sit (2/0)
|   |   |   |   |   left_hand_z >= 1.38 : bend (35/0)
|   |   |   right_foot_x >= -2.09
```

Este fichero es leído de forma secuencial, y a partir de aquí se construyen los nodos de cada árbol de decisión y los enlaces entre nodos. Para construir los enlaces correctamente, se utiliza una pila de nodos, donde a cada nuevo nivel de profundidad del árbol se empila un nodo. El formato del fichero provoca que el orden de lectura se realice en profundidad, comenzando con los hijos izquierdos y, a continuación, con los derechos. La función de la pila es regresar al nodo padre una vez se terminen de enlazar los nodos del hijo

izquierdo, para continuar con el hijo derecho. Una vez se termine de enlazar un nodo, este es eliminado de la pila. La pila debe, por tanto, comenzar y terminar vacía. Alcanzará su tamaño máximo cuando se estén procesando las hojas más profundas del árbol de decisión. [A.3]

7.2. Módulos y comunicación con ROS

El paquete completo se compone de tres ejecutables de ROS: el detector de posturas estáticas, el detector de movimientos y la interfaz gráfica. Cada uno de ellos se suscribe a algunos ROS *topics* (a partir de ahora, también nos referiremos a ellos como “entradas”) y publica información en otros *topics* (que reconocemos como “salidas”).

Detector de posturas
Entradas:

- /tf (transformaciones). Aquí se encuentran los puntos del esqueleto detectados por Kinect.

Salidas:

- /kinect_postures/postures. Se publica un string, que identifica la postura reconocida.

Detector de movimientos
Entradas:

- /tf (transformaciones). Aquí se encuentran los puntos del esqueleto detectados por Kinect.

Salidas:

- /kinect_postures/status. Estado del detector de movimientos (máquina de estados). Se representa mediante un entero positivo.
- /kinect_postures/moves. La salida es un vector de 3 valores, representa un movimiento finalizado. El primer valor es un booleano que indica si el movimiento se ha realizado correctamente. Los siguientes dos valores indican la precisión del movimiento y de la postura final (representados en coma flotante con precisión simple, con valores entre 0 y 1).

Interfaz gráfica
Entradas:

- /kinect_postures/status. Estado del detector de movimientos.
- /kinect_postures/postures. String que identifica la última postura estática.
- /kinect_postures/moves. Vector que representa el último movimiento.

Salidas:
La interfaz gráfica no publica en ningún ROS topic.

La Figura 7.1 muestra una representación en grafo (obtenida con 'rqt_graph') de todas las entradas y salidas descritas, durante una ejecución simultanea de todos los módulos.

7.3. Interfaz gráfica del usuario

El detector de posturas y movimientos desarrollado en este proyecto dispone de una interfaz gráfica de usuario (Fig. 7.2), diseñada con Qt. Esta interfaz gráfica muestra, de forma más visible, información de la salida del detector de posturas y movimientos. De esta forma se mejora la interacción con el usuario, ya que éste sabe cuál es el estado del programa.

La interfaz gráfica muestra lo siguiente:

1. **Postura actual**: postura estática obtenida por del clasificador de posturas.
2. **Estado del detector de movimientos**: estado de la máquina de estados del detector de movimientos (esperando / primera postura / en movimiento).
3. **Texto guía**: guía para el usuario, basado en los últimos estados del detector de movimientos. Por ejemplo, "En espera. Puede comenzar el movimiento"
4. **Información sobre el último movimiento**: se muestra si el movimiento es correcto o incorrecto, la precisión del movimiento y la precisión de la postura final.

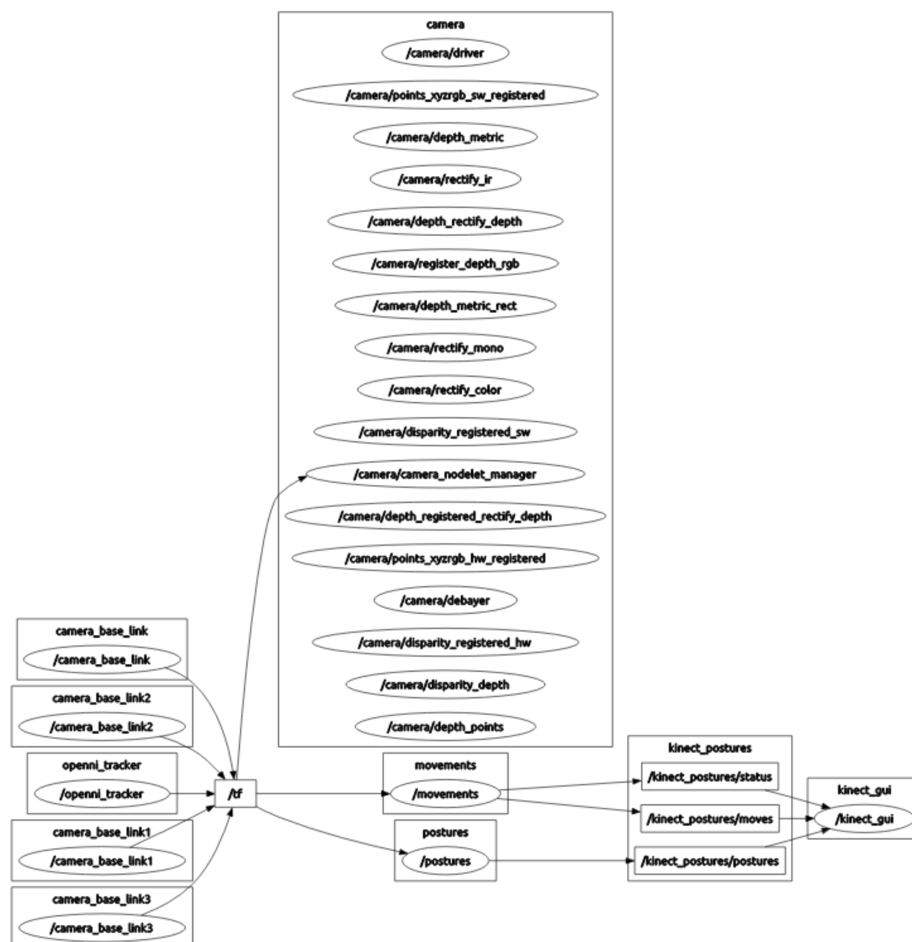


Figura 7.1: Representación en grafo de los nodos y *topics* de ROS durante una ejecución de todos los componentes.

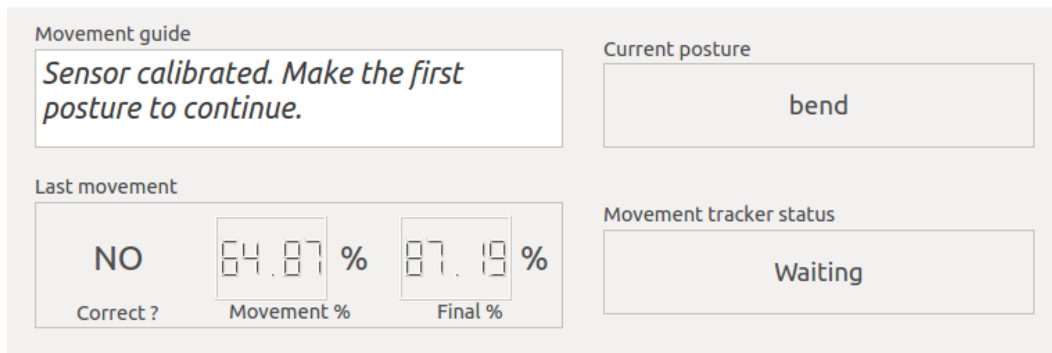


Figura 7.2: Ventana de la interfaz gráfica del usuario.

Esta ventana gráfica está implementada en un ejecutable independiente de ROS. Este ejecutable funciona como un *subscriber* que escucha en los *topics* donde publican el detector de posturas y el detector de movimientos.

8. CAPÍTULO

Conclusiones

El resultado del proyecto cumple con los objetivos planteados al inicio. El programa final es capaz de reconocer posturas definidas previamente, incluso de entre varios grupos de posturas diferentes. También reconoce movimientos, mostrando con qué nivel de precisión se han realizado. A pesar de que sólo se haya experimentado un tipo de movimiento (levantar el brazo derecho), el planteamiento realizado es aplicable a cualquier tipo de movimientos. Una experimentación más extensa podría demostrar su eficacia en este aspecto.

Además del reconocimiento de posturas estáticas y movimientos, se ha añadido una interfaz gráfica a la propuesta inicial del proyecto. Esta interfaz proporciona retroalimentación adicional al usuario, facilitando el uso del programa.

El proceso de *machine learning* ha sido, probablemente, la parte más importante de este proyecto. De este proceso se pueden extraer varias conclusiones.

Las mismas técnicas no son siempre eficaces, incluso en escenarios similares. Durante el proceso de aprendizaje supervisado, cuando se quiso diferenciar entre diferentes posturas sólo con brazos, se empleó la técnica de rango intercuartílico para detectar *outliers*. Se obtuvieron buenos resultados en este caso, por lo que se volvió a emplear la misma técnica para diferenciar posturas como: de pie/sentado/tumbado/encorvado. En este caso los resultados no fueron tan satisfactorios.

Eficacia de los *random forests*. Los experimentos realizados para clasificar posturas con *random forests* han demostrado su eficacia para clasificar posturas, al menos cuando el vector de características está formado por puntos del esqueleto. Los *random forests*

obtenían tasas de acierto similares a otros clasificadores en validación cruzada, pero han sido más eficaces que los árboles de decisión únicos clasificando en directo.

***Clustering* para modelar movimientos.** Un movimiento se compone de varias imágenes estáticas, una tras otra. Para este proyecto se grabaron muchas imágenes de un mismo movimiento y se agruparon en *clusters*. Cada *cluster* es un fragmento del movimiento. En las pruebas realizadas, este tipo de representación ha mostrado diferencias entre los movimientos que se realizaban correctamente y los que no. De todas formas, no es capaz de diferenciarlos en todos los escenarios, presenta algunos problemas y es mejorable.

8.1. Posibles mejoras

Ruido aleatorio. El conjunto de datos utilizado para el aprendizaje supervisado se ha obtenido realizando lecturas consecutivas de la información del esqueleto, en varias ocasiones y con diferentes posturas. Esta es una tarea larga y es fácil cometer errores (por ejemplo, introduciendo mal el nombre de la postura antes de comenzar a leer datos). Una forma de mejorar esto sería hacer menos lecturas desde el sensor e introducir ruido de forma aleatoria, para después utilizar los datos con ruido para aprendizaje supervisado. De esta forma podemos aumentar el conjunto de entrenamiento a cualquier número de instancias deseado, simplemente añadiendo ruido aleatorio para generar una nueva instancia.

Problemas del modelo de movimientos. El modelo de movimientos basado en *clusters* tiene un defecto: no importa el orden en el que los *clusters* son detectados. Esto significa que sea cual sea el orden en el que se realice un movimiento, este movimiento será igualmente aceptado. De todas formas, esto no afecta al movimiento completo, sino a las posturas intermedias (recordemos que el modelo completo está basado en posturas estáticas para la postura inicial y final, y *clusters* para las posturas intermedias). Otro defecto importante es que, durante el reconocimiento de posturas intermedias, se tiene en cuenta la máxima cercanía al centroide de la postura detectada, pero no la mínima cercanía. Esto provoca que algunos movimientos que se desvían de su trayectoria original o que incluso añaden más movimientos innecesarios sean detectados como correctos, siempre que pasen por todas las posturas intermedias reconocidas como parte del movimiento. Por ejemplo, si un movimiento consiste en levantar el brazo izquierdo hasta la altura de la cabeza, será detectado como correcto también si se levanta también el brazo derecho al mismo tiempo. También funcionaría levantar el brazo izquierdo, moverlo arbitrariamente,

regresar a la trayectoria original y completar el movimiento.

8.2. Posibles aplicaciones

Al ser una aplicación desarrollada para ROS, se puede integrar directamente al sistema de un robot que use este software. Un uso posible es la interacción con personas por visión a través del sensor Kinect. Las personas situadas frente a un robot pueden no ser simples obstáculos, sino interactuar con el robot a través del lenguaje corporal y los gestos. Por ejemplo, una persona podría indicar al robot en qué dirección ir, cuándo debe parar, o cuándo debe continuar con gestos de manos.

8.3. Comentarios adicionales

Se dedicaron muchas horas del proyecto a la configuración del entorno de ROS + Kinect + OpenNI, debido a que no funcionaba correctamente con ROS Indigo y Ubuntu 14.04 (las versiones estables más recientes a fecha de diciembre de 2014). A pesar de que la versión anterior se había probado antes y funcionaba correctamente, se insistió en instalar la última versión. Después de muchas horas de intentos y pruebas fallidas, finalmente se regresó a ROS Hydro y Ubuntu 12.04, por lo que fueron "horas perdidas". El código del proyecto no está fuertemente ligado a la versión de ROS, por lo que también debería ser compatible con futuras versiones, o fácilmente exportable.

Anexos

A. ANEXO

Implementación de árboles de clasificación

En el capítulo 7 (implementación) se han dado algunos detalles acerca de cómo están implementados los árboles de clasificación y *random forests*. Para ello, se ha hecho referencia a diferentes clases y fragmentos de código, en ocasiones, acompañando una versión en pseudocódigo de la implementación. Como complemento a ese capítulo, en este anexo se incluyen los fragmentos de código reales a los que se hace referencia.

A.1. Clase 'modeltree'

– modeltree.h –

```
#define TREE_TYPE_ATTRIBUTE 0
#define TREE_TYPE_CLASS 1
#define TREE_POS_LEFT 0
#define TREE_POS_RIGHT 1

// Recursive tree for classifying postures. An object of this class contains a tree node.
class modeltree {
public:
    // Root constructor
    modeltree(int type, std::string attr, float val);
    // Child constructor
    modeltree(int type, std::string attr, float val, modeltree* parent, int pos);
    // Classify: Recursive method to classify a dataset. Returns the posture of the leaf.
    std::string classify (std::map<std::string, float> dataset);
private:
    // type: 1=leaf, 0=rest of nodes.
```

```
17      int treetype;
18      // attribute, value: attribute and value to compare, to continue with left or right child.
19      std::string attribute;
20      float value;
21      // left and right child nodes.
22      modeltree* left;
23      modeltree* right;
24  };
```

– modeltree.cpp –

```
1   modeltree::modeltree(int type, std::string attr, float val) {
2       treetype = type;
3       attribute = attr;
4       value = val;
5   }
6
7   modeltree::modeltree(int type, std::string attr, float val, modeltree* parent, int pos) {
8       treetype = type;
9       attribute = attr;
10      value = val;
11      if (pos == TREE_POS_LEFT)
12          parent->left = this;
13      else if (pos == TREE_POS_RIGHT)
14          parent->right = this;
15  }
16
17  std::string modeltree::classify(std::map<std::string,float> dataset) {
18      if (treetype == TREE_TYPE_ATTRIBUTE) {
19          float data_value = dataset.at(attribute);
20          if (data_value <= value)
21              return left->classify(dataset);
22          else
23              return right->classify(dataset);
24      }
25      else {
26          return attribute;
27      }
28  }
```

A.2. Clase ‘randomforest’

– randomforest.h –

```
struct posture_votes {
    std::string posture;
    int votes;
};

class randomforest {
public:
    randomforest();
    void createFromFile(const char*  file);
    std::string classify(std::map<std::string,float> dataset);
private:
    std::vector<modeltree*> roots;
};
```

A.3. Generación de *random forests* desde fichero

– randomforest.cpp –

```
void randomforest::createFromFile(const char*  file) {
    std::stack< std::pair<modeltree*, int> > nodes;
    std::ifstream filestream;
    filestream.open(file);
    if (!filestream) {
        LoadModelFileException ex;
        throw ex;
    }
    std::string w1;
    while (filestream >> w1) {
        if (w1 == "==========") {
            int nextside = -1;
            std::string line;
            while (std::getline(filestream, line)) {
                std::stringstream cline(line);
                std::string word;
                cline >> word;
                if (word == "") continue;
                if (word == "Size") break;
                int depth, side;
                float value;
                std::string attribute;
                depth = 0;
```

```
24                 while (word == "|") {
25                     depth++;
26                     cline >> word;
27                 }
28                 attribute = word;
29                 cline >> word;
30                 side = nextside;
31                 if (word == "<")
32                     nextside = 0;
33                 else
34                     nextside = 1;
35                 cline >> value;
36                 if (side < 0) {
37                     modeltree* root = new modeltree(0, attribute, value);
38                     nodes.push(std::pair<modeltree*, int>(root,0));
39                     roots.push_back(root);
40                 } else {
41                     while (depth < nodes.top().second) {
42                         nodes.pop();
43                     }
44                     if (depth > nodes.top().second) {
45                         modeltree* child = new modeltree(0, attribute, value, nodes.top().first,
     side);
46                         nodes.push(std::pair<modeltree*, int>(child, depth));
47                     }
48                 }
49                 if (!cline.eof()) {
50                     cline >> word;
51                     if (word == ":") {
52                         cline >> attribute;
53                         modeltree* leaf = new modeltree(1, attribute, 0, nodes.top().first,
     nextside);
54                     }
55                 }
56             }
57         }
58     }
59     filestream.close();
60 }
```

Bibliografía

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine Learning*.

[John and Langley, 1995] John, G. H. and Langley, P. (1995). Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo. Morgan Kaufmann.

[Le et al., 2013] Le, T., Nguyen, M., and Nguyen, T. (2013). Human posture recognition using human skeleton provided by Kinect. *IEEE*.

[Martin, 1995] Martin, B. (1995). Instance-based learning: Nearest neighbor with generalization. Master's thesis, University of Waikato, Hamilton, New Zealand.

[OSRF, 2015] OSRF (2015). ROS documentation.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*, volume 1. Morgan Kaufmann.

[Shi, 2007] Shi, H. (2007). Best-first decision tree learning. Master's thesis, University of Waikato, Hamilton, NZ. COMP594.

[Webb, 1999] Webb, G. (1999). Decision tree grafting from the all-tests-but-one partition. San Francisco, CA. Morgan Kaufmann.

[Witten et al., 2005] Witten, I., Frank, E., and Hall, M. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2 edition.

[Xiao et al., 2012] Xiao, Z., Mengyin, F., Yi, Y., and Ningyi, L. (2012). 3D human postures recognition using Kinect. *IEEE*.