Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo e implementación de un asistente virtual para Linux

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Cristian Fernández Pardo

Tutor: Carlos David Martínez Hinarejos

2013-2014

Resumen

El objetivo del presente trabajo es profundizar en los fundamentos teóricos necesarios en el reconocimiento del habla y en el diseño e implementación para el desarrollo de un asistente virtual que, mediante sencillas órdenes por voz, sea capaz de ejecutar una serie de órdenes. Este trabajo resulta interesante ya que hoy en día la tecnología está en constante evolución y hay que adaptarse a las nuevas tendencias.

Con ello se pretende desvelar los conocimientos teóricos necesarios en el ámbito del reconocimiento automático del habla, el diseño de los modelos del lenguaje previos a la implementación y una posterior optimización de la aplicación. Además, se detallan las herramientas y tecnologías de las que hemos hecho uso durante el trabajo.

Palabras clave: voz, iATROS, reconocimiento del habla, asistente virtual, Ubuntu, Markov

Abstract

The aim of this work is to deepen into the theoretical foundations of speech recognition and the design and implementation of a virtual assistant able to execute a series of commands through voice. Current technology is constantly evolving and we must adapt to new trends; this is the main interest covered in this project.

The aim is to research about the necessary theoretical knowledge in the field of automatic speech recognition, the design of language models before the implementation and further optimization of the application. In addition, the tools and technology used are also detailed.

Keywords: voice, iATROS, speech recognition, virtual assistant, Ubuntu, Markov.

Tabla de contenidos

1 Introducción

En el proyecto hemos utilizado la tecnología del reconocimiento de voz para implementar un pequeño asistente para sistemas Linux (Ubuntu 13.10) y que, mediante sencillas órdenes habladas, el usuario pueda interactuar con el sistema.

Recientemente se ha diseñado e implementado un asistente por voz para Ubuntu por un joven programador llamado James McClain [6], disponible en código abierto y en inglés. Por otro lado, también se ha desarrollado, y está en constante evolución, reconocedores del habla en el sector de los dispositivos móviles (Google Glass, smartphones, tablets…).

En nuestro caso, sí que es cierto que ya hay ciertos asistentes en el mercado para sistemas como Microsoft Windows, pero no tantos para Linux. El reconocimiento del habla mediante el uso de ordenadores se utiliza generalmente en el sector de la investigación, aunque también existen productos comerciales. A continuación introduciremos los aspectos básicos y contenido teórico más relevante.

1.1 Reconocimiento del habla

El reconocimiento automático de voz o del habla [1] (RAH), consiste en el procesado de la voz humana para obtener la información, es decir, traducirla a texto. Por lo tanto, el objetivo del RAH es obtener una representación simbólica discreta de una señal vocal continua.

Existen una serie de dificultades o conflictos para el RAH como pueden ser: la variación fonética inter-locutor (entre hablantes) o intra-locutor (en un mismo hablante), los estilos del habla, las características del entorno… Por ello, el desarrollo de un sistema de RAH requiere de dos fases:

- Fase de entrenamiento o *training*: muestras de la lengua recogidas en un corpus etiquetado.
- Fase de reconocimiento: a partir de los datos adquiridos durante la primera fase, se clasifica una nueva muestra (señal vocal).

Cabe resaltar la diferencia entre RAH y procesamiento del lenguaje natural. RAH es la tecnología que traduce el habla a texto, mientras que el procesamiento del lenguaje natural consiste en analizar las palabras obtenidas y otorgarles un significado dentro del contexto. En este proyecto hacemos uso de iATROS, software de RAH, el cual comentaremos con más detalle en el Sección 3.1.

1.2 Tecnologías utilizadas

El sistema base para nuestro asistente virtual ha sido la distribución de Linux Ubuntu 13.10. La razón por la que escogimos esta distribución fue porque es una de las más utilizadas, con una gran comunidad donde encontrar soporte y ayudas y porque era una de las únicas distribuciones que se podían instalar junto a un sistema Windows 8.1.

El motivo de escoger Linux en lugar de Windows fue porque el reconocedor de voz que hemos utilizado, que comentaremos más adelante en el Capítulo 3, está implementado para funcionar sobre una distribución Linux. Además era algo más novedoso e innovador desarrollar un asistente por voz para Linux, puesto que existen en menor cantidad. En cambio, para sistemas Windows poco a poco van apareciendo reconocedores de voz como *Dragon Naturally Speaking* o el propio de Windows (Reconocimiento de voz).

Por un lado, nuestro asistente hace uso del reconocedor de voz iATROS que comentaremos en el Capítulo 3. Es un reconocedor automático del habla desarrollado por el personal del PRHLT (Pattern Recognition and Human Language Technology) centro de investigación de la UPV (Universitat Politècnica de València).

Por otro lado, hemos hecho uso de varios lenguajes de programación para el desarrollo del asistente, más concretamente del intérprete. El lenguaje Python ha sido seleccionado para la implementación del intérprete, que detallaremos en el Capítulo 4. En cambio, el reconocedor de voz ha sido implementado en lenguaje C, como veremos en el Capítulo 3. Además, hemos implementado algunos *scripts* que nos han facilitado el trabajo de implementación de los modelos del reconocedor, que se comentan en el Capítulo 3; en concreto se han utilizado dos *scripts* implementados en lenguaje Bash. Por último, el

tratamiento de fonemas se ha realizado a través de un *script* escrito en Perl, dedicado al tratamiento del español.

1.3 Objetivos

El objetivo principal en nuestro proyecto es implementar un asistente virtual que permita gestionar y manipular un sistema Linux mediante sencillas órdenes de voz, para ello necesitaremos:

- Estudiar los aspectos y funcionamiento básicos del RAH y de iATROS.
- Diseñar los modelos léxico y de lenguaje para iATROS.
- Modificar iATROS para que guarde la frase reconocida en un fichero de texto.
- Implementar un pequeño intérprete que a partir de la frase reconocida ejecute la orden correspondiente.
- Implementar un *script* ejecutable que inicie tanto iATROS como el intérprete.
- Optimizar los parámetros para reducir el error en el reconocimiento.

2 Teoría del reconocimiento del habla

En este Capítulo presentamos las bases de la teoría del reconocimiento del habla, componente básico de nuestro proyecto. Para ello destacamos los siguientes componentes:

- Modelos ocultos de Markov (HMM): son modelos probabilistas temporales en el que cada estado del proceso está definido por una única variable aleatoria discreta.
- Algoritmo de Viterbi: éste algoritmo va íntimamente ligado a los HMM y sirve para hallar la secuencia de estados más problable en un HMM. En nuestro caso nos devolverá una secuencia de sonidos (palabras, fonemas…) más probable.

2.1 Modelos ocultos de Markov (HMM)

Un modelo oculto de Markov [7] (HMM por sus siglas en inglés, Hidden Markov Model) se utiliza en el reconocimiento de voz para representar el modelo acústico del que se compone cada sonido (usualmente fonema). Para poder entender su funcionamiento necesitaremos hablar primero de los procesos de Markov.

Un proceso de Markov es un proceso estocástico para el que se cumple la propiedad de Markov: estando en un estado *n*, la probabilidad de pasar al estado *n+1* no depende de los estados visitados anteriormente, sino solo del actual. Por ese motivo los procesos de Markov son llamados procesos estocásticos sin memoria. Un HMM es un modelo probabilista temporal en el que cada estado del proceso está definido por una única variable aleatoria discreta, es decir, es un modelo estadístico de Markov en el que se asume que el sistema modelado es un proceso de Markov con parámetros no observados (ocultos).

En un modelo de Markov el estado es visible, por lo que los únicos parámetros son las probabilidades de transición entre estados; en cambio, en un HMM el estado no es visible directamente, únicamente las variables influidas por el estado. Cada estado tiene una distribución de probabilidades

sobre los posibles símbolos de salida; por lo tanto, la secuencia de símbolos generada puede aportar información en relación a la secuencia de estados.

El objetivo de los HMM es determinar el valor de los parámetros ocultos a partir de los parámetros observables. En nuestro caso, dados los parámetros del modelo y una secuencia de salida, deberemos encontrar la secuencia más probable de estados ocultos que puedan haber generado dicha salida (frase o palabra). Para este problema se hace uso del Algoritmo de Viterbi explicado en el Sección 2.2. Los HMM son ideales para modelar procesos que tienen cambios de estado a lo largo del tiempo. En nuestro caso, para una aplicación de reconocimiento del habla, podemos considerar una palabra como cambios de fonemas a lo largo del tiempo.

Un HMM se define como una quíntupla $\{Q, \sum, \pi, A, B\}$ donde:

- Q es un conjunto de estados.
- $\sum$ es un conjunto de símbolos observables (símbolos de salida).
- π es el vector de probabilidades iniciales.
- A es la matriz de probabilidades de transición entre estados.
- B es la matriz de probabilidades de emisión de símbolos.

La secuencia de observaciones se denota como un conjunto $O = \{o_1, o_2, \dots, o_n\}$.

Un ejemplo gráfico de un HMM es como el que se muestra en la Figura 1.

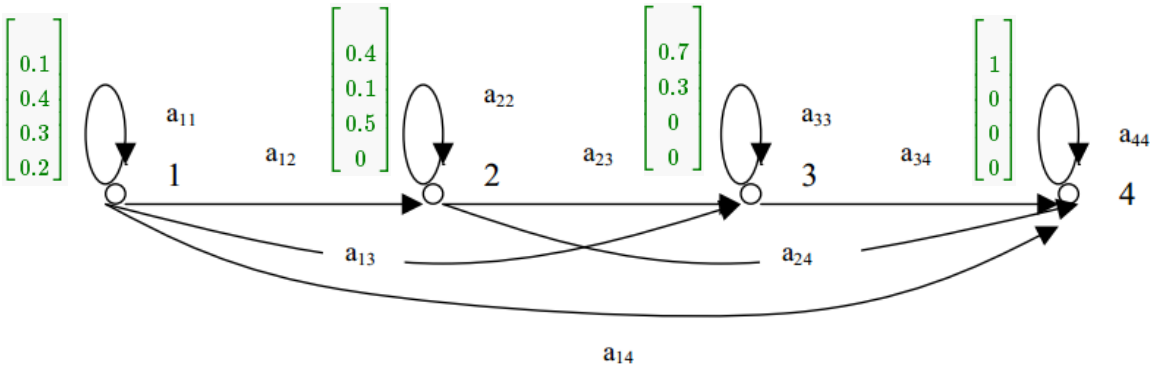

Figura 1: Representación de un HMM como un grafo dirigido.

2.2 Algoritmo de Viterbi

Uno de los problemas de los HMM es que los estados que generan la secuencia de observaciones quedan ocultos, y por el contrario, para el RAH es muy importante conocer exactamente la secuencia de estados que genera la secuencia de observaciones, con el objetivo de poder determinar con qué palabras nos encontramos (previamente definidas en los modelos léxico y del lenguaje). Para esto se utiliza el algoritmo de Viterbi [8].

El algoritmo permite encontrar la secuencia de estados $S = (q_1, q_2, \dots, q_n)$ más probable en un HMM, a partir de una observación $O = (o_1, o_2, \dots, o_n)$, quiere decir que obtiene la secuencia óptima de estados que mejor explica la secuencia de observaciones.

Como se ha comentado, el algoritmo consiste en encontrar la secuencia de estados como un todo, utilizando un criterio de máxima verosimilitud. Para facilitar el cálculo, se define la siguiente variable auxiliar:

$$\delta_t(i) = \max_{q_1, q_2, \dots, q_{t-1}} P(q_1, q_2, \dots, q_t = i, o_1, o_2, \dots, o_t | \mu) \qquad (1)$$

La cual da la más alta probabilidad que pueda tener la secuencia de observaciones y la secuencia de estados hasta el instante *t*, cuando el proceso está en el estado *i* para el modelo μ. Se puede comprobar que se cumple la siguiente relación recursiva:

$$\delta_{t+1}(j) = \left[\max_{1 \le i \le N} \delta_t(i)(a_{ij}) \right] b_j(o_{t+1}) \qquad (2)$$

donde:

$$\delta_1(j) = \pi_j b_j(o_1), \quad 1 \le j \le N \qquad (3)$$

De esta manera, el procedimiento para encontrar la secuencia más probable de estados comienza con el cálculo de $\delta_t(j)$ haciendo uso de la recurrencia mostrada en la Ecuación (2), manteniendo siempre un apuntado al estado que

va ganando, es decir, aquel que posea la mayor probabilidad. Finalmente se encuentra el estado *j* como:

$$j^* = \arg max_{1 \le j \le N} \delta_t(j) \tag{4}$$

El algoritmo en pseudocódigo es el que se muestra en la Figura 2:

1. Inicialización:

   $\delta_t(i) = \pi_i B_i(O_1)$, donde $1 \le i \le N$

2. Recursión:

   a. $\delta_{t+1}(\mathrm{i}) = \left[\max_{1 \le i \le N} \delta_t(i)\left(A_{ij}\right)\right] B_j(O_{t+1})$

   b. $\varphi_{t+1}(j) = \arg \max_{1 \le i \le N} \delta_t(i) A_{ij}$

   Donde t = 1, 2, ..., T-1, $1 \le i \le N$ y $1 \le j \le N$

3. Terminación:

   $q_T^* = \arg \max_{1 \le i \le N} \delta_T(i)$

4. Reconstrucción de la secuencia de estados más probable:

   En orden inverso, es decir, desde el final hasta el principio:

   $q_t^* = \varphi_{t+1}(q_{t+1}^*)$, donde t = T-1, T-2, ..., 1

Figura 2: Pseudocódigo del algoritmo de Viterbi.

3 Tecnologías utilizadas

En este Capítulo introduciremos las aplicaciones o tecnologías externas utilizadas para el desarrollo del proyecto, bien sea que nuestro proyecto ha hecho uso de estas aplicaciones durante el desarrollo o bien durante la ejecución del mismo. En concreto hablaremos de:

- iATROS: un reconocedor del habla desarrollado por el centro de investigación PRHLT (Pattern Recognition and Human Language Technology).
- Eutranscribe: *script* desarrollado en Perl para la transcripción fonética en español. Dada una palabra el *script* devuelve los fonemas que la conforman.
- Entorno del sistema: el desarrollo del proyecto está basado en el entorno de escritorio propio del sistema.
- Herramientas del sistema: para la gestión de las órdenes se han utilizado algunos *scripts* y también se mencionan la metodología y el tratamiento de los comandos.

3.1 iATROS

iATROS [9] (del inglés Improved Automatic Trainable RecOgnition System) es una implementación de ATROS, el cual es un reconocedor de voz anterior. iATROS ha sido adaptado tanto para el reconocimiento del habla como para el reconocimiento de texto manuscrito y ha sido desarrollado por el PRHLT (Pattern Recognition and Human Language Technology), centro de investigación perteneciente a la UPV (Universitat Politècnica de Valencia).

iATROS proporciona una estructura modular que puede ser utilizada para construir diferentes sistemas, en cuyo núcleo existe una búsqueda de Viterbi sobre una red de modelos ocultos de Markov. iATROS proporciona herramientas estándar de reconocimiento del habla *off-line* y *online* (en base a módulos ALSA).

El uso de iATROS será en modo *online* para la aplicación desarrollada y en modo *off-line* para las pruebas de rendimiento. Además se han modificado

varios ficheros de configuración propios de iATROS que se nombran a continuación:

- **Fichero de configuración de la señal de audio** (conf.feat):

En el modo *online* el proceso que realizamos es de adquisición de audio desde iATROS; por ello este fichero contiene la configuración que define los parámetros de la señal de audio de entrada. Los parámetros más importantes se adjuntan en la Tabla 1.

Tabla 1: Parámetros del fichero de la señal de audio.

SampleFrec	Indica la frecuencia de muestreo de la señal de audio.
Channels	El número de canales de audio de entrada.
Bits	El tamaño de la muestra empleado en la cuantificación del audio.
InputDevice	Dispositivo de entrada de audio.

- **Fichero de configuración del sistema** (fsm_configuracion.cnf):

Este fichero contiene los parámetros que hacen que el reconocedor iATROS funcione, como son las rutas de los ficheros de entrada y algunos parámetros básicos. En concreto resaltaremos los parámetros que se comentan en la Tabla 2.

Tabla 2: Parámetros representativos del fichero de configuración.

hmm	Ruta del fichero del modelo acústico
lexicon	Ruta del fichero del modelo léxico
grammar	Ruta del fichero del modelo del lenguaje
gramar-type	Indica el modo en el que están construidos los ficheros de los modelos: FSM (finite state machine), NGRAM (n-gramas).
beam	Se utiliza como criterio de poda en el algoritmo de búsqueda.
gramar-scale-factor	Otorga mayor importancia al modelo de lenguaje con respecto al acústico cuanto más grande sea.
Word-insertion-penalty	Cuanto más alto más penaliza la inclusión de palabras, por lo que favorece la decodificación de palabras largas.

- **Ficheros de modelos** (léxico, lenguaje, acústico):

Para el diseño del reconocedor es necesarios de estos tres modelos que definan el lenguaje que el sistema es capaz de reconocer: el modelo léxico, el modelo del lenguaje y el modelo acústico. A continuación los explicamos con más detalle.

3.1.1 Modelo acústico

El modelo acústico comprende todos los sonidos (fonemas) que pueden pronunciarse en nuestro lenguaje, almacenando sus características. Cada sonido está representado por un HMM continuo, con las salidas modeladas por una distribución probabilista de mixturas gaussianas. El modelo acústico que hemos utilizado para el proyecto utiliza 64 gaussianas para cada estado.

Cada fonema está asociado a un identificador que lo representa en nuestro modelo, de modo que pueda utilizarse el identificador en las transiciones entre estados del modelo léxico (p.ej. el fonema /a/ estará representado por “a”). También hay que tener en cuenta que la transcripción fonética no se deriva de la grafía de la palabra, sino que hay que interpretarla fonéticamente. Para ello haremos uso de `eutranscribe` que comentaremos en la Sección 3.2.

Un ejemplo del código representado en el modelo acústico para el fonema /a/ se muestra en la Figura 3.

```
<MIXTURE> 1 3.474252e-03
<MEAN> 39
1.785863e+04 1.299498e+03 -6.432632e+02 1.513383e+03 -1.147531e+02 1.304942e+03 5.227284e+00
1.238573e+03 -1.836611e+01 1.322657e+03 2.116885e+02 1.140614e+03 -1.483093e+02 4.423981e+02
6.415623e+02 1.492168e+02 3.146061e+01 1.392591e+02 2.472162e+01 1.126314e+02 5.481182e+01 8.754484e+01
5.021929e+01 1.099550e+02 7.642397e+01 7.027842e+01 1.855742e+02 -5.750425e+01 1.617091e+01 1.649160e+01
-3.283789e+01 1.179788e+01 -2.388708e+01 3.539843e+00 -2.320968e+01 2.741090e+00 -3.192752e+01
3.130974e+00 2.895082e-01
<VARIANCE> 39
5.879569e+05 1.443083e+05 5.697394e+04 5.408928e+04 3.555932e+04 3.183869e+04 4.928634e+04
1.966820e+04
2.380357e+04 1.743093e+04 2.286620e+04 2.281169e+04 1.893668e+04 6.287507e+04 2.368605e+04
8.469077e+03
8.191718e+03 4.942003e+03 5.167835e+03 3.632337e+03 3.156643e+03 3.855771e+03 3.091294e+03 3.360326e+03
2.176172e+03 2.106594e+03 9.947387e+03 3.104086e+03 1.455257e+03 1.118613e+03 7.147718e+02 4.974401e+02
5.415772e+02 4.094380e+02 3.893118e+02 3.348078e+02 3.334549e+02 3.642191e+02 3.403601e+02
<GCONST> 4.081876e+02
```

Figura 3: Código para el modelado acústico del fonema /a/.

Los modelos acústicos empleados en el sistema implementado se entrenaron a partir de la base de datos Albayzín [12]. Albayzín es un corpus fonético en español, compuesto por 4800 adquisiciones fonéticamente equilibradas y con

diversas variedades dialectales de español, con un total de aproximadamente 2 horas de señal acústica. Para el entrenamiento de los modelos (HMM contínuos de tres estados, topología izquierda-derecha sin saltos) se empleó la herramienta HTK [13].

3.1.2 Modelo léxico

Es el que define la pronunciación de las palabras que forman el lenguaje a reconocer. Cada palabra está compuesta por un autómata finito determinista (AFD), de modo que el modelo léxico se trata de un fichero con el mismo número de autómatas que de palabras a reconocer. Para poder definir este modelo es necesario disponer del modelo acústico.

El diseño del fichero es sencillo. Para cada autómata mantiene un mismo formato donde solo varían dos variables, se puede observar en la Figura 4.

Name "*nombre*"
State 0 i=1
0 1 "*fonema_0/1*" p=1
02 "*fonema_0/2*" p=1
…
n-1 n "*fonema_n-1/n*" p=1
State n f=1

Figura 4: Estructura del autómata para el modelo léxico.

Donde lo único que varía son *nombre* y *fonema*; *nombre* será el nombre que representa el autómata y la palabra, y *fonema* el símbolo que representa al fonema en el modelo acústico. Observamos que existe una probabilidad asociada para cada estado, pero como es un autómata no ambiguo la probabilidad es 1 en todos los casos. Asimismo, el estado 0 del autómata viene indicado como que es el estado inicial con "i=1" y el último estado "n" (estado final) con "f=1".

Un ejempo del diseño de un autómata se muestra a continuación, en este caso para la palabra "copiar", en la Figura 5.

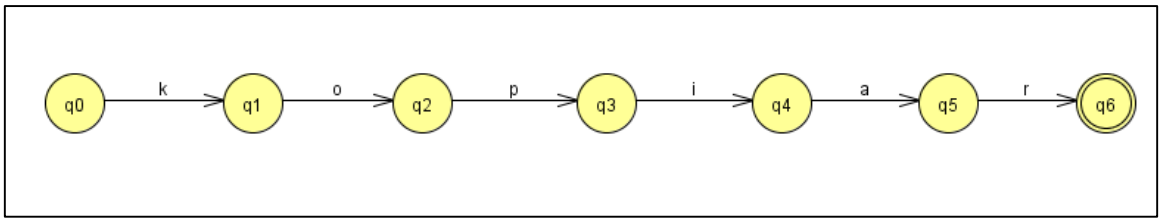

Figura 5: Diseño gráfico del autómata para la palabra Copiar.

El código para el autómata anterior, y algunos otros ejemplos, que se muestra en el modelo léxico de iATROS aparecen en la Figura 6.

```
Name "copiar"
State 0 i=1
0 1 "k" p=1
1 2 "o" p=1
2 3 "p" p=1
3 4 "i" p=1
4 5 "a" p=1
5 6 "r" p=1
State 6 f=1
```

```
Name "escritorio"
State 0 i=1
0 1 "e" p=1
1 2 "s" p=1
2 3 "k" p=1
3 4 "r" p=1
4 5 "i" p=1
5 6 "t" p=1
6 7 "o" p=1
7 8 "r" p=1
8 9 "i" p=1
9 10 "o" p=1
State 10 f=1
```

```
Name "fichero"
State 0 i=1
0 1 "f" p=1
1 2 "i" p=1
2 3 "c" p=1
3 4 "e" p=1
4 5 "r" p=1
5 6 "o" p=1
State 6 f=1
```

Figura 6: Código de diferentes autómatas del modelo léxico.

3.1.3 Modelo de lenguaje

Este modelo es el que define las frases de nuestro lenguaje. Para poder definir este modelo es necesario disponer del modelo léxico y, en consecuencia del modelo acústico. Este modelo también está compuesto por un AFD con un estado inicial y un único estado final (el final de la frase), de modo que en este AFD se encuentran todas las frases reconocibles en nuestro lenguaje. Las transiciones entre estados son las palabras a reconocer y además cuentan con una probabilidad de emisión asociada.

La Figura 7 muestra un ejemplo de representación de la orden “abrir calculadora”.

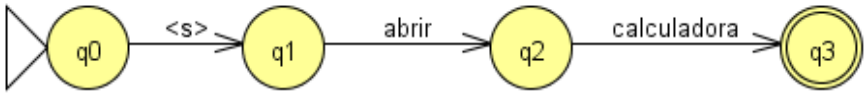

Figura 7: Diseño gráfico para un autómata del modelo de lenguaje.

Existen los modelos de lenguaje construidos con n-gramas. Sin embargo, para nuestra tarea no son apropiados puesto que para hacer uso de n-gramas se necesita tener un corpus de muestras ya existente. En nuestro caso hemos creado desde cero la gramática y por tanto no existían muestras anteriormente. Por ese mismo motivo hemos hecho uso de un tipo de gramática con estados finitos.

3.2 eutranscribe

Es un *script* escrito en Perl que se encarga de traducir las palabras españolas al alfabeto fonético definido en los modelos acústicos. Para ello algunas de las operaciones que realiza son:

- Eliminar acentos.
- Pasar todo a minúsculas.
- Elimina signos de puntuación especiales e iniciales.
- Funde las cadenas de blancos.
- Sustituye '*j*' por '*x*'.
- La doble '*r*' la transforma en '@'.
- La '*ñ*' se representa con el fonema /*h*/.
- La grafía "*qu*" se pronuncia como '*k*'.
- Las grafías "*gue*" y "*gui*" se cambian por "*ge*" y "*gi*" a no ser que lleven diéresis.
- La '*g*' antes de '*e*' e '*i*' se sustituyen por el fonema /*j*/ que en este caso sería '*x*'.
- La '*c*' delante de '*e*' e '*i*' se sustituye por '*z*'.
- La "*ch*" se cambia por '*c*'.
- La '*c*' cuando no es "*ch*" es una '*k*'.
- Si concurren dos nasales lingualveolares, se pronuncia solo una.
- La partícula "*ps*" a principio de palabra se pronuncia '*s*'.

- La '*w*' se pronuncia "*gu*".
- La '*v*' se transforma en '*b*'.

En nuestro caso, hemos utilizado `eutranscribe` para transcribir las palabras de las frases a reconocer: "abrir calendario", "silenciar audio", "minimizar ventana", etc. Con ello conseguíamos el modelo fonético que es capaz de reconocer iATROS. Algunos ejemplos más representativos se muestran en la Tabla 3.

Tabla 3: Ejemplos de transcripción mediante `eutranscribe`.

Palabra original	Palabra tras aplicar eutranscribe
firefox	fairfoks
calculadora	kalkuladora
cerrar	ze@ar
volver	bolber
reiniciar	@einiziar
deshacer	desazer
office	ofis
texto	testo
fichero	ficero

3.3 Gestión de órdenes utilizando herramientas del sistema

Para el tratamiento de las órdenes por voz se ha hecho uso de varias herramientas que ofrece Linux. La principal son las órdenes de Linux que se utilizan mediante terminal. En segundo lugar se ha hecho uso de algunas librerías específicas y/o aplicaciones de Ubuntu 13.10 instaladas a propósito; de entre ellas cabe resaltar *XMacro* [10].

La gestión de las órdenes se hace mediante un fichero llamado *ordenes.txt*, que veremos en la Sección 4.3, donde se encuentran las frases que nuestro intérprete es capaz de reconocer (que deberán estar debidamente representadas por su autómata en el fichero correspondiente) y las órdenes que les corresponden. De este modo nuestro intérprete está capacitado para evolucionar de manera que tan solo habría que añadir los autómatas en los

modelos de lenguaje y léxico, y posteriormente agregar la frase y la orden correspondiente en este archivo. Todo esto lo veremos con más detalle en el Capítulo 4.

En su mayoría las órdenes son las propias de Linux que se ejecutan desde terminal. Nuestro intérprete es capaz de reconocer la frase y ejecutar en segundo plano la orden. No obstante, hay algunas frases que hacen uso de aplicaciones o herramientas más concretas, como es el caso de XMacro.

XMacro es un paquete que contiene dos aplicaciones muy simples escritas en lenguaje C++, `xmacrorec` y `xmacroplay`; con ellas podemos grabar y reproducir eventos de teclado y de ratón. Con ello conseguimos poder reproducir eventos como los atajos y combinaciones de teclado, pulsaciones de teclas concretas, cerrar ventanas, etc. Esta herramienta nos ha sido muy útil para casos concretos en que Linux no dispone de una orden específica para realizar una tarea. Sí que es cierto que existen órdenes para apagar o reiniciar un ordenador, silenciar o activar el audio, ejecutar una aplicación, etc. Sin embargo, no existen órdenes para mostrar el escritorio o minimizar una ventana.

El uso de XMacro es muy sencillo. Primeramente, tras haber instalado XMacro, hemos hecho uso de `xmacrorec`, con la cual hemos grabado los eventos de ratón y/o teclado, y se han guardado en un fichero de texto. Posteriormente esta combinación de teclado se agrega al fichero de órdenes, pero asignándole el servidor donde queremos que se reproduzca mediante `xmacroplay`. En la Figura 8 se muestra un ejemplo para ejecutar la orden "cortar texto".

```
cortar texto    echo "KeyStrPress Control_L KeyStrPress x
    KeyStrRelease x KeyStrRelease Control_L" | xmacroplay :0
```

Figura 8: Código para ejecutar la orden "Cortar texto".

El texto que obtenemos, en este ejemplo, tras ejecutar `xmacrorec` es:

```
KeyStrPress Control_L KeyStrPress x KeyStrRelease x
KeyStrRelease Control_L.
```

Este código es el correspondiente a la combinación de teclado “Control+X”. Posteriormente, para que la ejecución de ésta orden se realice en la ventana que está en uso se añade “:0”; esto no es más que el identificador, pues por defecto el cero es la ventana que está en uso.

Por otro lado, también se ha hecho uso de tuberías (*pipes*) y herramientas y órdenes muy comunes en Linux como `awk`, `grep` o `ps`. Resultan de una gran utilidad las tuberías en Linux para gran cantidad de tareas que se realizan en terminal.

Una tubería o *pipe* en Linux consiste en tomar la salida de una orden y utilizarla como entrada de una segunda orden. Esto hace que con una sola combinación puedas realizar una búsqueda de un proceso y terminarlo, como es nuestro caso. Su sintaxis es muy sencilla: `orden1 | orden2 | orden3`.

También se utilizan las tuberías en XMacro; como se observa en la Figura 8, el resultado obtenido tras ejecutar la combinación de teclas “Control+X” se envía como entrada a la orden `xmacroplay` que lo muestra en la ventana de uso actual.

Además, como hemos comentado, se han utilizado algunas órdenes muy comunes en Linux: `awk`, `grep`, `ps`, `xargs`, `kill`, `amixer`, `shutdown`. En la Tabla 4 comentamos con más detalle cada uno de ellas.

Tabla 4: Órdenes de Linux utilizadas en los diferentes *scripts*.

Comando	Uso
`awk`	Se utiliza para manipular texto. Se le pasan como parámetros de entrada una acción sobre el texto y un texto o fichero de entrada. En nuestro caso la entrada se le pasa por tubería.
`grep`	Busca dentro de un fichero o un texto de entrada, las líneas que concuerdan con un patrón.
`ps`	Informa del estado de un proceso (Process Status), aportando información como su PID que es el que utilizamos nosotros para terminar el proceso.
`xargs`	Dada un fichero de entrada (en nuestro caso a través de tubería)

	ejecuta la orden que se le pasa como opción (en nuestro caso la orden `kill`) para cada una de las líneas de la entrada.
`kill`	A este proceso se le pasa como entrada un PID. El resultado es matar el proceso con el PID que se le pasa por la entrada.
`amixer`	Es un mezclador de sonidos para ALSA que permite controlar el volumen de nuestro sistema desde la terminal.
`shutdown`	Es una orden de Unix que reinicia, detiene, apaga el sistema y manda mensajes a los usuarios, todo dependiendo de las opciones de entrada que tenga. Haremos uso de ésta orden para apagar y reiniciar el sistema.

4 Desarrollo

En éste Capítulo vamos a explicar el desarrollo de nuestro asistente virtual: las implementaciones que hemos hecho, el diseño de los autómatas necesarios para algunos modelos y todos los *scripts* y aplicaciones que nos han servido para completar la tarea.

La primera fase del desarrollo ha sido una fase de investigación donde era necesario cubrir las necesidades teóricas para tener los conocimientos y la base necesaria, de manera que pudiésemos entender el funcionamiento de cada parte del reconocedor iATROS. La base teórica más representativa ha sido vista en el Capítulo 2.

En segundo lugar hemos diseñado las órdenes que nuestro asistente iba a ser capaz de reconocer e interpretar, además de diseñar los autómatas del modelo de lenguaje y del modelo léxico. Principalmente el diseño fue en papel, dejando para el final toda la implementación. Todo esto lo hemos visto en el Capítulo 3.

En tercer lugar se han implementado los modelos de lenguaje y léxico, el fichero de gestión de las órdenes que nuestro asistente es capaz de ejecutar, el intérprete que se encarga de obtener de iATROS la frase pronunciada y asociarla a la frase que es capaz de ejecutar (definida en el modelo de lenguaje), el lanzador del asistente (cuyo objetivo es centralizar la ejecución de iATROS y del intérprete) y, por último, la implementación de diferentes *scripts* de ayuda.

En la Figura 9 se muestra el esquema del desarrollo del asistente. En concreto, se remarcan las partes donde hemos desarrollado e implementado y, a continuación, se explica en cada apartado la implementación que se ha llevado a cabo.

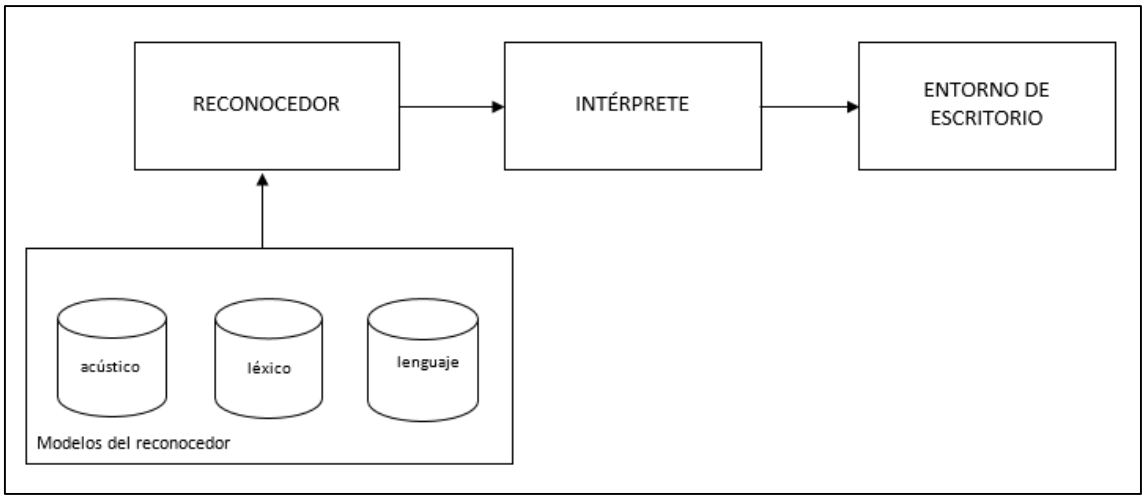

Figura 9: Esquema general del trabajo dividido por módulos de trabajo.

4.1 Modelo de lenguaje

Como se observa en el grafo del apartado 3.1.3, donde hablábamos del modelo de lenguaje y exponíamos un sencillo ejemplo, al inicio de cada frase el reconocedor detecta un silencio que marca con la etiqueta <s>. Para ello en nuestro autómata hemos tenido que añadir que el inicio de cada frase empiece por un silencio. Teniendo en cuenta esto, el diseño de nuestro autómata queda representado como se muestra en la Figura 10.

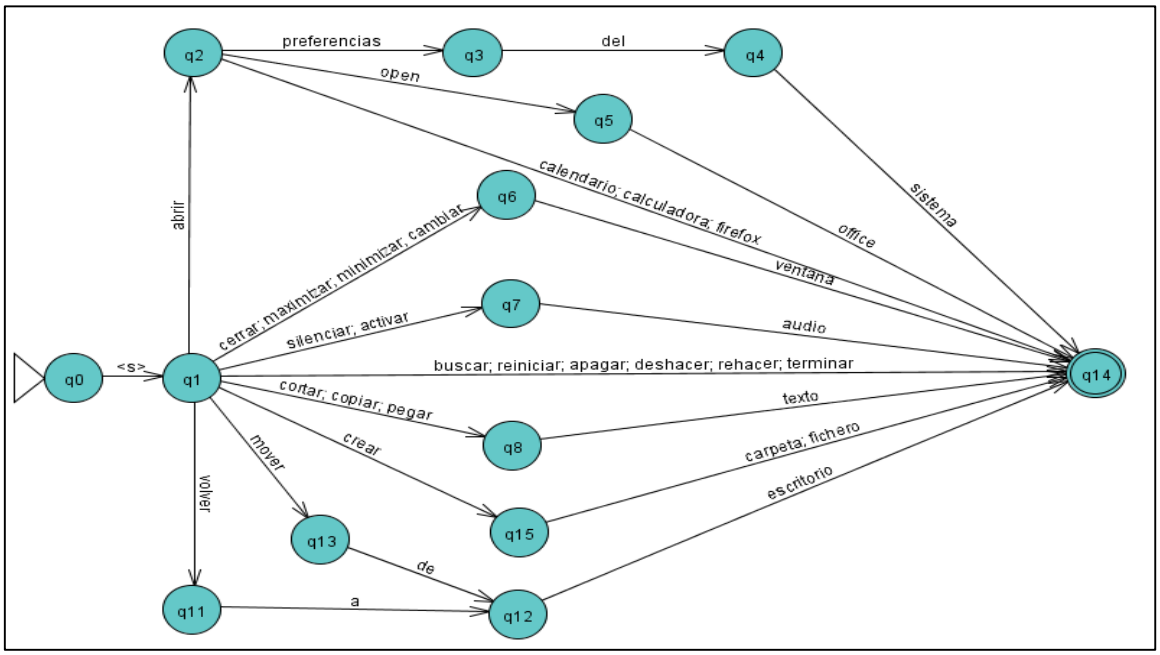

Figura 10: Autómata Finito Determinista de nuestro intérprete.

De este modo, estando en un estado inicial y al reconocer el silencio inicial pasamos al estado 1; desde este estado, reconociendo alguna de las palabras que se muestran en el grafo (abrir, volver, mover, crear, cerrar, silenciar…), pasaremos al siguiente estado (correspondiente a la palabra reconocida). Una vez lleguemos al estado 14 se termina y se devuelve la frase reconocida. Si existiése la frase en el fichero de órdenes, que veremos a continuación en la Sección 4.3, ejecutaría su órden correspondiente.

Es deducible que cuanto mayor sea el número de frases que se quieran reconocer, mayor será la complejidad del autómata. En nuestro caso se detectó un fallo en el diseño, puesto que se crearon dos estados para una misma palabra (crear), de modo que desde el estado 1 podíamos alcanzar el estado 9 y el estado 10 con la palabra *crear*. La solución fue eliminar dichos estados y crear uno nuevo (estado 15) que los uniese a ambos. Por eso se observa en el diseño del autómata que los estados 9 y 10 no existen.

Dentro del fichero del modelo de lenguaje también existen varios parámetros que es necesario rellenar correctamente: `NumStates`, `NumEdges` y `TotalChecksum`. `NumStates` corresponde al número de estados que existen en total en nuestro autómata (en nuestro caso son 16 porque tuvimos un fallo y creamos 2 estados que no se utilizan). `NumEdges` es el número de aristas que existen en el autómata. En el diseño aparecen varias palabras en una misma arista, pero lo correcto es contar cada palabra como una arista distinta; por tanto el total de aristas en nuestro proyecto es de 37. Por último `TotalChecksum` es el total que deben sumar las probabilidades de cada arista. Para asignar las probabilidades se aplica $p = 1/NumEdgesState$ siendo `NumEdgesStates` el número de aristas que salen desde ese estado.

En la Figura 11, se muestra una parte de la implementación del autómata, correspondiente a los primeros estados. Se observa que en cada estado existe una probabilidad de que se produzca el salto de un estado a otro. En nuestro caso hemos utilizado la misma probabilidad para todos los casos; sin embargo, existen casos en que una palabra se repite frecuentemente y para ello se le asigna una probabilidad más alta. Con esto conseguimos reducir el error en algunos casos.

```
State 1 c = 1
  1 2   "abrir"     p = 0.0526
  1 6   "cerrar"    p = 0.0526
  1 6   "minimizar"      p = 0.0526
  1 6   "maximizar"      p = 0.0526
  1 8   "cortar"    p = 0.0526
  1 8   "copiar"    p = 0.0526
  1 8   "pegar"     p = 0.0526
  1 14  "buscar"    p = 0.0526
  1 14  "reiniciar"      p = 0.0526
  1 14  "apagar"    p = 0.0526
  1 14  "deshacer"       p = 0.0526
  1 14  "rehacer"   p = 0.0526
  1 7   "silenciar"      p = 0.0526
  1 7   "activar"   p = 0.0526
  1 6   "cambiar"   p = 0.0526
  1 13  "mover"     p = 0.0526
  1 15  "crear"     p = 0.0526
  1 11  "volver"    p = 0.0526
  1 14  "terminar"  p = 0.0526
```

Figura 11: Códido del estado 1 del modelo de lenguaje.

4.2 Modelo léxico

El modelo léxico, como se ha comentado en el apartado 3.1.2, define la pronunciación de las palabras; por tanto nuestro fichero está formado por el conjunto de autómatas que definen cada una de las palabras, de modo que en este caso ya aparece la palabra representada por sus fonemas y no por sus caracteres. En la Figura 12 se muestran varios ejemplos.

```
Name
"cerrar"
State 0 i=1
0 1 "z" p=1
1 2 "e" p=1
2 3 "@" p=1
3 4 "a" p=1
4 5 "r" p=1
State 5 f=1
```

```
Name
"reiniciar"
State 0 i=1
0 1 "@" p=1
1 2 "e" p=1
2 3 "i" p=1
3 4 "n" p=1
4 5 "i" p=1
5 6 "z" p=1
6 7 "i" p=1
7 8 "a" p=1
8 9 "r" p=1
State 9 f=1
```

```
Name
"firefox"
State 0 i=1
0 1 "f" p=1
1 2 "a" p=1
2 3 "i" p=1
3 4 "r" p=1
4 5 "f" p=1
5 6 "o" p=1
6 7 "k" p=1
7 8 "s" p=1
State 8 f=1
```

```
Name
"office"
State 0 i=1
0 1 "o" p=1
1 2 "f" p=1
2 3 "i" p=1
3 4 "s" p=1
State 4 f=1
```

Figura 12: Ejemplos de autómata de cada palabra.

Los dos ejemplos más representativos de nuestro lenguaje son las palabras “Office” y “Firefox”. Al ser dos palabras inglesas la representación de los fonemas ha de ser en el lenguaje español, y por tanto hay que describirlas

como suena la palabra. Por tanto en el caso de la palabra “Office” en español se pronuncia /ofis/ y en el caso de”Firefox” su pronunciación es /fairfoks/.

Cabe resaltar que para la realización de estos autómatas se ha hecho uso de dos aplicaciones/*scripts* que han facilitado y automatizado la implementación del fichero, ya que resultaba tedioso tener que escribir uno a uno cada autómata de cada palabra.

En primer lugar se ha hecho uso de la aplicación `eutranscribe`. Como se ha comentado anteriormente en la Sección 3.2, esta aplicación convierte la palabra en lenguaje español a sus fonemas.

En segundo lugar, se implementó un *script* en lenguaje Bash, el cual se le llamó `script.sh`. Tras ejecutar este *scrpit*, solo había que pasarle la palabra que nos devolvía `eustranscribe` y el *script* se encarga de escribir su autómata correspondiente en el fichero `lexico.lx`. En la Figura 13 se muestra el código del *script*. Gracias a este *script* podíamos realizar cualquier autómata con tan solo escribir la palabra; por tanto, las palabras más largas, que contienen más líneas de código en su autómata, han sido sencillas y rápidas de implementar. Tras finalizar la ejecución del *script* se guardaban todos los cambios en el fichero.

```
#!/bin/bash
# -*- ENCODING: UTF-8 -*-

NUM=0
state=0
state2=1

while [ $NUM -ne 1 ]; do
    echo 'Introduce la palabra:'
    read pal
    echo 'Introduce el fonema:'
    read texto
    long=`expr length $texto`
    echo $texto $long
    echo  >> lexico.lx
    echo 'Name "'$pal'"' >> lexico.lx
    echo 'State 0 i=1' >> lexico.lx
    while [ $long -ne "0" ]; do
         inicial=`expr substr $texto $state2 1`
         echo $state $state2 '"'$inicial'" p=1' >>
       lexico.lx
```

```
        let "state+=1"
        let "state2+=1"
        let "long-=1"
    done
    echo 'State '$state' f=1' >> lexico.lx
    let "state=0"
    let "state2=1"
done
```

Figura 13: Código de `script.sh`.

4.3 Fichero de órdenes

Uno de los archivos más importantes en el funcionamiento del asistente es este. Es en este fichero donde asignamos la orden que se va a ejecutar cuando se reconozca una frase. Para ello hemos creado un simple fichero de texto que se consulta una vez se reconoce una frase, de modo que cuando tenemos la frase reconocida se recorre todo el fichero para determinar si existe una orden asociada a esa frase; si lo encuentra ejecuta el comando, en caso contrario devuelve error. Este proceso se realiza desde el intérprete que se verá en el siguiente apartado.

La estructura del fichero es muy sencilla; se divide en tantas filas como frases es capaz de reconocer nuestro asistente y, por cada fila, dos columnas: frase a reconocer y orden a ejecutar.

Con esta estructura nuestro intérprete recorre la primera columna de cada fila, buscando que cada una de las palabras que iATROS ha reconocido concuerde con alguna de las frases que existen en el fichero de órdenes. Esto hace posible que nuestro asistente crezca de manera rápida y sencilla, con tan solo tener que crear los autómatas y agregar la frase y la orden en una nueva línea del fichero.

En la Figura 14 se muestra un ejemplo del fichero con algunas de las frases que nuestro asistente reconoce.

```
buscar   echo "KeyStrPress Super_L KeyStrRelease Super_L" |
         xmacroplay :0
reiniciar shutdown -r now
apagar    shutdown -h now
silenciar audio     amixer sset Master mute
```

```
activar audio  amixer sset Master unmute; amixer sset
          Speaker unmute; amixer sset Headphone unmute;
crear carpeta  echo "Carpeta creada en" && pwd | awk
          '{print $1}' && mkdir carpetaCreada
crear fichero  echo "Fichero creado en" && pwd | awk
          '{print $1}' && touch ficheroCreado.txt
```

Figura 14: Código correspondiente al fichero de órdenes.

4.4 Intérprete

El intérprete que hemos desarrollado está escrito en Python. Elegimos este lenguaje de programación por su gran potencial en el tratamiento de cadenas de texto y en el sistema de almacenamento y recuperación de texto. El intérprete básicamente es un fichero Python (.py) con el código correspondiente al tratamiento de la orden reconocida en iATROS.

Para la implementación del intérprete hemos hecho uso de varias bibliotecas de Python como son: `os`, `pickle`, `sys`.

El funcionamiento de nuestro intérprete es bastante sencillo e intuitivo. Hay una primera fase donde se realizan las inicializaciones de las variables globales. La segunda fase es un bucle `while`, que conforma el cuerpo del intérprete, del que vamos a ir comentando su funcionamiento.

En primer lugar, hemos inicializado las variables globales, una de ellas es la variable *"frase",* que es donde guardaremos la frase que iATROS ha reconocido. Por tanto, el bucle estará en constante comprobación siempre y cuando la frase reconocida sea distinta de "terminar".

```
"parametros"
     frase = "inicio"
     comparador = 1
     i = 0

     while frase != "terminar\n":
```

Figura 15: Inicialización del intérprete.

En segundo lugar, existe un segundo bucle, llamado bucle de espera; éste sirve para hacer esperar al intérprete hasta que exista una frase reconocida por iATROS, de manera que mientras no exista el fichero de salida de iATROS, el intérprete se mantiene en un bucle infinito. Por el contrario, cuando iATROS

reconoce una frase y la guarda en el fichero de salida (`salida.txt`), nuestro intérprete lo detecta y accede a dicho fichero para interpretar la orden.

A continuación se realiza el tratamiento de la cadena reconocida. Primero de todo guardamos la frase en nuestra variable "*frase*" e inmediatamente eliminamos el fichero `salida.txt` para nada más terminar hacer esperar a nuestro intérprete hasta la siguiente orden. En caso de que la frase reconocida no sea un silencio, aplicamos una búsqueda de la frase reconocida en el fichero de órdenes y, en caso de encontrar una coincidencia, se ejecuta la orden asociada a dicha frase.

Se puede consultar el código del intérprete en los anexos de nuestro trabajo.

5 Mejoras y experimentación

En éste Capítulo vamos a comentar el proceso de experimentación y optimización aplicado a nuestro trabajo con el objetivo de minimizar la tasa de error en el proceso de reconocimiento.

En primer lugar, hemos creado un corpus de muestras sobre el que aplicar la optimización y la experimentación de los parámetros que comentaremos a continuación. En nuestro caso, las muestras son las 24 órdenes que nuestro asistente es capaz de interpretar. Para ello se han grabado las 24 órdenes pronunciadas por 5 personas distintas. Para la grabación hemos utilizado "arecord" [11], un grabador de sonidos para terminal de Linux que hace uso de los módulos ALSA de sonido, que soporta una gran cantidad de formatos. Las muestras están grabadas en un formato *Little Endian* utilizando 16 bits por cada muestra y en una frecuencia de muestreo de 16 kHz. La frecuencia determina la cantidad de muestras de sonido capturadas en cada segundo, mientras que el número de bits que se utilizan por muestra determina la precisión en que se ha guardado la muestra. Además, para facilitar la tarea hemos realizado un *script* en lenguaje Bash que automatizaba la grabación, y su posterior grabado en disco, para cada voluntario.

En segundo lugar, una vez obtenido el corpus de muestras hemos calculado los vectores de características para cada muestra (*Cepstrum*). Cada uno de estos vectores es el resultado de aplicar una serie de transformaciones, entre ellas la transformada de Fourier, y contienen las características que representan la muestra del sonido grabado. Con esto podremos tratar el sonido de la voz humana y realizar un análisis para determinar la tasa de error.

Para este tratamiento hemos hecho uso de la aplicación propia de iatros (`iatros-speech-cepstrals`). Además, para facilitar y reducir el tiempo en ésta tarea hemos realizado un *script* que automatizaba el proceso; este *script* accede a las grabaciones y realiza el cálculo de cepstrales para cada muestra y lo guarda en un fichero. Por último, se crea un fichero con las rutas de directorio donde se encuentra cada grabación, que nos servirá en el siguiente paso.

Continuando con el proceso, hemos implementado un tercer *script*, de experimentación, que realiza una ejecución de iatros en modo *off-line* con distintos valores en los parámetros más representativos que hemos comentado en la Tabla 1 de la Sección 3.1 (*beam*, *gramar-scale-factor* y *word-insertion-penalty*). Dicho *script* se encarga de realizar una ejecución de iatros para cada combinación de estos tres parámetros y guardar los resultados en un fichero de salida. Los valores que hemos utilizado se muestran en la Tabla 5.

Tabla 5: Valores de los parámetros de optimización del intérprete.

beam	20, 40, 60, 80, 100, 150, 200
gramar-scale-factor	1, 5, 15, 20
word-insertion-penalty	-10, -5, 0, 5, 10

Para realizar la comparativa de los resultados obtenidos tras la ejecución de cada combinación de valores se ha escrito un fichero de texto con la transcripción de las órdenes que el reconocedor es capaz de interpretar. Con esto, y mediante otro *script*, podemos realizar una comparativa de los resultados con las órdenes originales, y así conseguir la tasa de error. Para la implementación de este *script* se ha hecho uso de la orden `paste`, que copia la entrada de datos en el fichero que se le indica. Posteriormente se ha utilizado la aplicación `tasas`, implementada también por el centro de investigación PRHLT. Ésta aplicación calcula la tasa de error entre dos cadenas de texto. La sintaxis de dicho *script* es la mostrada en la Figura 16.

```
cat ficheroTranscripcion | paste –d “#” –ficheroSalida
      | sed “s/<s>//g” | sed “s/<\/s>//g”
      | tasas –f “#” –s “ “ –ie –
```

Figura 16: Sintaxis del script para el cálculo de la Tasa de Error.

Por último, guardamos en un fichero de salida el resultado de la tasa de error obtenido para cada fichero de salida obtenido en el *script* de experimentación. A continuación en la Tabla 6 mostramos la tasa de error para las diferentes combinaciones de los parámetros.

De este modo hemos calculado que la mejor combinación de los parámetros anteriormente comentados, que consigue minimizar la tasa de error y, por tanto, optimizar el funcionamiento del asistente, es la que se muestra en la Tabla 7.

Tabla 6: Tasas de error para las diferentes combinaciones de los parámetros.

	20					40				
	-10	-5	0	5	10	-10	-5	0	5	10
1	42.12	42.12	42.97	50.63	59.57	31.06	33.61	33.19	32.76	35.74
5	42.55	46.80	56.59	77.02	99.14	33.61	34.04	35.74	37.44	38.72
15	100	100	100	100	100	56.17	75.31	99.14	99.14	100
20	100	100	100	100	100	99.14	100	100	100	100

	60					80				
	-10	-5	0	5	10	-10	-5	0	5	10
1	31.91	30.63	31.06	33.19	32.34	34.8	35.74	36.59	36.59	38.29
5	32.34	31.91	31.06	32.76	33.61	35.74	37.44	39.14	39.57	39.57
15	38.72	40.42	40.42	44.68	54.04	39.57	39.57	41.70	40.85	40.85
20	46.80	55.31	71.91	99.14	99.14	42.55	42.55	45.95	46.38	49.36

	100					150				
	-10	-5	0	5	10	-10	-5	0	5	10
1	40.85	41.70	42.55	41.27	41.27	47.23	47.23	46.38	46.80	49.78
5	42.55	42.12	42.12	41.70	42.55	46.38	47.23	46.80	46.80	51.48
15	43.40	44.25	44.25	44.25	44.25	48.08	51.06	50.21	50.21	53.19
20	44.25	45.10	45.10	46.80	46.80	50.21	51.06	50.21	51.48	54.04

	200				
	-10	-5	0	5	10
1	51.06	52.34	53.19	53.61	53.61
5	52.34	53.61	54.46	54.46	52.76
15	53.61	53.61	55.31	56.17	55.31
20	56.17	57.02	57.02	56.17	56.17

Tabla 7: Resultados de los valores óptimos tras la experimentación.

beam	60
gramar-scale-factor	1
word-insertion-penalty	-5

6 Conclusiones

Durante el desarrollo de nuestro proyecto hemos mejorado nuestros conocimientos sobre el reconocimiento de formas ya vistos durante el Grado, más concretamente el del habla. Además, hemos aprendido nuevas materias y aplicaciones con respecto al reconocimiento automático del habla, como es iATROS, que no hemos visto durante el Grado de Informática.

En primer lugar hay que destacar lo innovador que resulta realizar un asistente virtual, puesto que es una tecnología muy atractiva hoy en día, ya que cada vez está integrándose en el uso cotidiano. El proyecto, en definitiva, ha resultado muy productivo, interesante, innovador y formativo.

En segundo lugar, cabe resaltar que los aspectos más complicados en el desarrollo de nuestro asistente han sido las implementaciones del intérprete y del modelo de lenguaje. El principal problema ha sido el mal tratamiento de las cadenas de texto que iATROS reconocía y su posterior *matching* en el fichero de órdenes. En un principio se implementaron todas las órdenes a ejecutar como un diccionario en Python. La problemática de tratar las cadenas de texto dentro de un diccionario nos hicieron pensar una alternativa más sencilla sin tener que salirnos del lenguaje de programación. El resultado fue un fichero de órdenes sencillo y claro mediante listas. Esto hace posible que nuestro asistente sea capaz de evolucionar rápidamente y de crecer en cuanto a frases capaz de reconocer, sin necesidad de modificar el código del intérprete. No obstante, sí que es necesario modificar el código de los modelos que utiliza iATROS.

Existen algunas mejoras que podríamos haber realizado en nuestro asistente, como por ejemplo crear una cantidad considerable de órdenes como para que nuestro asistente resulte útil en el día a día. También sería interesante diseñar una interfaz gráfica que permita interactuar con el asistente, así como detener el reconocedor de voz o reanudarlo, por ejemplo. Esta mejora resulta más complicada, puesto que el diseño gráfico de una aplicación es más complicado, ya que es necesario hacer uso de otro lenguaje de programación que cuente con bibliotecas gráficas y, por lo tanto, tendríamos que realizar una

integración entre el intérprete y la interfaz gráfica, o migrar toda la implementación al otro lenguaje.

Por otro lado, también sería interesante compilar el asistente de modo que pudiése ser portable a otros sistemas Unix, de manera que el ejecutable instalase todos los archivos necesarios y automatizase el proceso de instalación de iATROS y del intérprete. Del mismo modo, resultaría una mejora considerable crear un *script* que añadiese el autómata correspondiente a una frase nueva que deseamos añadir. Sin embargo, ésta mejora necesitaría un estudio previo y un trabajo bastante elaborado de implementación, que por tiempo no hemos realizado.

Cabe resaltar también que la tasa de error obtenida tras la experimentación es de un 30% aproximadamente debido a que se han dado muchos casos en que no se reconocía la orden grabada de la muestra del corpus, que hemos comentado en el Capítulo 5. Por tanto, eliminando todas aquellas órdenes no reconocidas la tasa de error real es de un 9%. Es decir, un 9% de las órdenes son mal reconocidas, un 21% aproximadamente, no se han reconocido y un 70% aproximadamente se reconocen correctamente.

En conclusión, hemos implementado un asistente capaz de reconocer una serie de órdenes cotidianas y sencillas, y ejecutarlas en el sistema Ubuntu. Con la capacidad de evolucionar de una manera sencilla y eficaz, con la sencillez de ejecución, con la parte más innovadora de hoy en día y con el fin de mejorar, motivar e impulsar la creación de asistentes virtuales para Linux.

Bibliografía

[1] Introducción al reconocimiento del habla. Visto el 11/06/2014. http://liceu.uab.es/~joaquim/speech_technology/tecnol_parla/recognition/speech_recognition/reconocimiento.html

[2] Fundamentals of Speaker Recognition. Visto y descargado el 08/07/2014. http://link.springer.com/book/10.1007%2F978-0-387-77592-0

[3] Pattern recognition and machine learning. Christopher M. Bishop. Springer (2006)

[4] Pattern recognition (second edition). Sergios Theoridis, Konstantinos Koutroumbas. Academic Press (2003)

[5] Inteligencia artificial, un enfoque moderno. Stuart Russell, Peter Norvig. Pearson Prentice Hall (2004)

[6] Blog sobre el reconocedor de voz de Linux implementado por James McClain. Visto el 17/07/2014. http://ospherica.es/ya-puedes-tener-reconocimiento-de-voz-en-tu-distro-linux/

[7] Documentación sobre los HMM por la Universidad Nacional de Colombia. Visto y descargado el 17/07/2014. http://www.virtual.unal.edu.co/cursos/ingenieria/2001832/lecturas/hmm.pdf

[8] Bases teóricas del Algoritmo de Viterbi. Visto el 17/07/2014. http://www.scielo.org.ve/scielo.php?pid=S1316-82120100002000004&script=sci_arttext

[9] Página web de iATROS. http://prhlt.iti.es/page/projects/multimodal/idoc/iatros

[10] Página web de Xmacro: http://xmacro.sourceforge.net/

[11] Arecord para Linux: http://quicktoots.linux-audio.com/toots/quick-toot-arecord_and_rtmix-1.html

[12] A. Moreno, D. Poch, A. Bonafonte, E. Lleida, J. Llisterri, J. B. Mariño, and C. Nadeu, "Albayzin speech database: design of the phonetic corpus", in Proceedings of EuroSpeech'93, (Berlin, Germany), pp. 175-178, sep 1993.

[13] Young, S.J., Evermann, G., M.J.F., Gales, T., Hain, D., Kershaw, G., Moore, J., Odell, Ollason, D., Povey, D., Valtchev, V., Woodland, P.C., 2006. "The HTK Book, version 3.4", Cambridge University Engineering Department, Cambridge, UK.

Anexos

Código del intérprete: interprete.py.

```
#!/usr/bin/python
#encoding:utf8
#" -*- coding: utf-8 -*-
import sys, pickle, os

if __name__ == '__main__':
      print "INTERPRETE XIGRY"
      "parametros"
      frase = "inicio"
      comparador = 1
      i = 0

      while frase != "terminar\n":
      while not
   os.path.isfile("/home/cristian/cristian/software/iatros-
   v1.0/build/salida.txt"):
                  k = 0
      salida = open("/home/cristian/cristian/software/iatros-
   v1.0/build/salida.txt","r")
            frase = salida.read()
      os.remove("/home/cristian/cristian/software/iatros-
   v1.0/build/salida.txt")
            if frase != "":
                  print "-----"
                  print frase
                  fraseSal = frase.split()
                  ordenes = open("ordenes.txt","r")
                  for linea in ordenes:
                        columnas = linea.split("\t")
                        fraseOrd = columnas[0].split()
                        comparador = 0
                        if len(fraseOrd) == len(fraseSal)-1:
                              comparador = 1
                              i = 0
                              while i < len(fraseOrd):
                                    if fraseOrd[i] != fraseSal[i+1]:
                                          comparador = 0
                                          break
                                    i = i + 1
                              if comparador == 0:
                                    continue
                              elif comparador == 1:
                                    os.system(columnas[1])
                        if comparador == 1:
                              break
                  if comparador == 0:
                        print "Patron no reconocido"
```

Código del fichero de órdenes: ordenes.txt

```
abrir calendarioSimpleAgenda&
abrir calculadora     gnome-calculator&
abrir preferencias del sistema  gnome-control-center&
abrir open office     libreoffice --writer &
abrir firefox   firefox&
cerrar ventana  echo "KeyStrPress Alt_L KeyStrPress F4
          KeyStrRelease F4 KeyStrRelease Alt_L" | xmacroplay :0
minimizar ventana     echo "KeyStrPress Control_L KeyStrPress
   Alt_L KeyStrPress KP_Insert KeyStrRelease KP_Insert
   KeyStrRelease Alt_L KeyStrRelease Control_L" | xmacroplay :0
maximizar ventana     echo "KeyStrPress Control_L KeyStrPress
   Alt_L KeyStrPress KP_Begin KeyStrRelease KP_Begin
   KeyStrRelease Alt_L KeyStrRelease Control_L" | xmacroplay :0
cortar texto    echo "KeyStrPress Control_L KeyStrPress x
     KeyStrRelease x KeyStrRelease Control_L" | xmacroplay :0
copiar texto    echo "KeyStrPress Control_L KeyStrPress c
     KeyStrRelease c KeyStrRelease Control_L" | xmacroplay :0
pegar texto     echo "KeyStrPress Control_L KeyStrPress v
     KeyStrRelease v KeyStrRelease Control_L" | xmacroplay :0
buscar     echo "KeyStrPress Super_L KeyStrRelease Super_L" |
    xmacroplay :0
reiniciar  shutdown -r now
apagar     shutdown -h now
silenciar audio amixer sset Master mute
activar audio   amixer sset Master unmute;
                amixer sset Speaker unmute;
                amixer sset Headphone unmute;
crear carpeta   echo "Carpeta creada en" && pwd | awk '{print
     $1}' && mkdir carpetaCreada
crear fichero   echo "Fichero creado en" && pwd | awk '{print
     $1}' && touch ficheroCreado.txt
volver a escritorio  echo "KeyStrPress Control_L KeyStrPress
     Super_L KeyStrPress d KeyStrRelease d KeyStrRelease
     Control_L KeyStrRelease Super_L" | xmacroplay :0
cambiar ventana echo "KeyStrPress Alt_L KeyStrPress Tab
     KeyStrRelease Tab KeyStrRelease Alt_L" | xmacroplay:0
mover de escritorio  echo "KeyStrPress Control_L KeyStrPress
     Super_L KeyStrPress Left KeyStrRelease Left KeyStrRelease
     Super_L KeyStrRelease Control_L" | xmacroplay:0
deshacer   echo "KeyStrPress Control_L KeyStrPress z
     KeyStrRelease z KeyStrRelease Control_L" | xmacroplay:0
rehacer    echo "KeyStrPress Control_L KeyStrPress y
     KeyStrRelease y KeyStrRelease Control_L" | xmacroplay:0
terminar   ps -A | grep lanzador.sh | grep -v grep | awk '{print
   $1}' | xargs kill | ps -A | grep iatros-speech-o | grep -v
   grep | awk '{print $1}' | xargs kill | ps -A | grep python |
   grep -v grep | awk '{print $1}' | xargs kill
```

Código del lanzador del asistente: lanzador.sh

```
#!/bin/bash
# -*- ENCODING: UTF-8 -*-

#ejecutamos el interprete
python interprete.py&


#ejecutamos iATROS
cd software/iatros-v1.0/build

./bin/iatros-speech-online -c
./models/fsm_configuracionXigry.cnf -p ./models/conf.feat

cd ..
cd ..
cd ..
```

Código del autómata del lenguaje: automata.gr

```
% Estados finitos para Asistente Virtual Xigry

Name   kk

NumStates       16

NumEdges        37

TotalChecksum   1

State 0 c = 1 i = 1
      0 1   "<s>" p = 1

State 1 c = 1
      1 2   "abrir"              p = 0.0526
      1 6   "cerrar"             p = 0.0526
      1 6   "minimizar"          p = 0.0526
      1 6   "maximizar"          p = 0.0526
      1 8   "cortar"             p = 0.0526
      1 8   "copiar"             p = 0.0526
      1 8   "pegar"              p = 0.0526
      1 14  "buscar"             p = 0.0526
      1 14  "reiniciar"          p = 0.0526
      1 14  "apagar"             p = 0.0526
      1 14  "deshacer"           p = 0.0526
      1 14  "rehacer"            p = 0.0526
      1 7   "silenciar"          p = 0.0526
      1 7   "activar"            p = 0.0526
      1 6   "cambiar"            p = 0.0526
      1 13  "mover"              p = 0.0526
      1 15  "crear"              p = 0.0526
      1 11  "volver"             p = 0.0526
      1 14  "terminar"           p = 0.0526
% numero de estados: 19
```

```
State 2 c = 1
      2 14 "calendario"          p = 0.2
      2 14 "calculadora"         p = 0.2
      2 3  "preferencias"        p = 0.2
      2 5  "open"                p = 2
      2 14 "firefox"             p = 0.2


State 3 c = 1
      3 4  "del"                 p = 1

State 4 c = 1
      4 14 "sistema"             p = 1

State 5 c = 1
      5 14 "office"              p = 1

State 6 c = 1
      6 14 "ventana"             p = 1

State 7 c = 1
      7 14 "audio"               p = 1

State 8 c = 1
      8 14 "texto"               p = 1

State 9 c = 1
      9 14 "carpeta"             p = 1

State 10 c = 1
      10 14 "fichero"            p = 1

State 11 c = 1
      11 12 "a"                  p = 1

State 12 c = 1
      12 14 "escritorio"         p = 1

State 13 c = 1
      13 12 "de"                 p = 1

State 14 c = 1 f = 1

State 15 c = 1
      15 14 "carpeta"            p = 0.5
      15 14 "fichero"            p = 0.5
```