Detección de actividad vocal (VAD)

PAV - Práctica 2

Adrià Guinovart
Héctor Antona

25 de octubre de 2020

1. Introducción

Esta práctica se centra en la detección de actividad vocal o, en otras palabras, dado un fichero de audio, analizar en qué partes se está hablando y en cuales tan solo se detecta ruido de fondo.

El método que seguiremos para lograr esto se centrará en el uso de autómatas de estados finitos, con siglas FSA en inglés, y que, más adelante, veremos cómo, y siguiendo qué criterios, lo hemos implementado. Esta metodología se basa en definir una serie de estados y, para cada momento de tiempo, nos encontraremos en uno de ellos. A priori y el modo más sencillo sería teniendo un estado de Voz y uno de Silencio entre los cuales movernos pero ya veremos como es de gran utilidad emplear unos estados adicionales para poder tomar decisiones más acertadas.

Para determinar dichos estados nos basaremos en los parámetros trabajados en la práctica aneterior, del mismo modo que reutilizaremos parte de su código y, sobretodo, emplearemos el cálculo de la potencia media de la señal (en decibelios) para realizar nuestras decisiones a la hora de escojer un estado u otro. Además del cálculo de la potencia, también recurriremos a otras estrategias para mejorar nuestra capacidad de detección. Estas consisten en asumir que tanto los tramos de voz como los de silencio tendrán una duración mínima, lo cual nos ayudará a no detectar falsos silencios si, por ejemplo, la persona que graba el audio habla muy pausadamente o deja algún espacio corto entre palabras. Sin embargo hemos tenido que ser cuidadosos de no excedernos con esta técnica ya que, en caso de detectar erróneamente algun tramo, esta presunción nos penalizaría, prolongando dicho error.

Una vez establecido los conceptos y metodología que vamos a seguir, procederemos a presentar la implementación y los resultados del código desarrollado a lo largo de la práctica.

2. Ejercicios básicos

2.1. Complete el código de los ficheros *main_vad.c* y *vad.c* para que el programa realice la detección de actividad vocal. Escriba las funciones de análisis o incorpore al proyecto los ficheros de la primera práctica *pav_analysis.c* y *pav_analysis.h*. Recuerde incorporar las cabeceras necesarias en los ficheros correspondientes. Tiene completa libertad para implementar el algoritmo del modo que considere más oportuno, pero el código proporcionado puede ser un buen punto de partida para hacerlo usando un autómata de estados finitos (FSA). Encontrará en los ficheros sugerencias para ello, marcadas con la palabra TODO (del inglés *to do*, a realizar). Ayúdese de la visualización de la señal y sus transcripciones usando wavesurfer, de la opción *–verbose* del programa *vad*, de la colocación de chivatos y de cualquier otra técnica que se le pueda ocurrir para conseguir que el programa funcione correctamente. Recuerde que el fichero de salida sólo debe incluir las etiquetas V y S.

En primer lugar, antes de ir a la implementación del código y explicar su funcionamiento, es necesario establecer la base teórica con la cual vamos a trabajar, ya que hay distintas maneras posibles de enfocar el problema. Como ya se ha anticipado en la introducción, emplearemos autómatas de estados finitos (FSA), concretamente el más complejo que se propone en el enunciado de la práctica y se representa mediante el siguiente esquema:

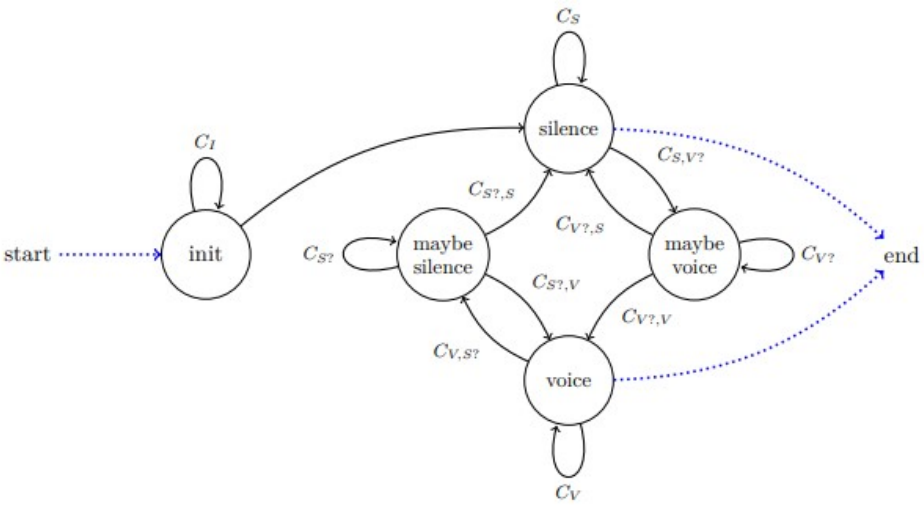

Figura 1: Esquema correspondiente al FSA implementado.

Las etiquetas de *start* y *end* no forman parte del propio FSA, simplemente indican que la secuencia de estados es de duración finita. En nuestro problema particular, indicarían el inicio y fin de la señal de voz analizada, respectivamente.

Una vez presentada la idea general del modelo implementado, vamos a empezar a comentar la implementación de cada uno de los estados posibles. Primero de todo explicar las variables que definimos al principio del documento. Comenzando por *count*, esta nos servirá para hacer un cálculo en las primeras muestras de la señal de audio, *firstFrames* para decidir sobre cuantas de las primeras tramas haremos esos cálculos, también *k0*, *k1* y *k2* que se utilizarán para almacenar el nivel medio del ruido y los dos thresholds a partir de los cuales tomar decisiones de cambio de estado y *offset1* y *offset2* se utilizarán para el cálculo de los thresholds ya mencionados. Por último, *undefinedFrames* se empleará para controlar cuanto tiempo se lleva en un estado indefinido mientras que *framesMS* y *framesMV* serán el número de muestras en las que permaneceremos en un estado indefinido antes de plantearnos cambiar de estado.

El esquema se compone de 5 estados posibles, uno de los cuales tan solo se visita al principio (estado *init*), y una vez se abandona ya no se puede volver a él. Este estado se emplea para calcular el nivel de silencio de referencia k_0 durante los primeros tramos de señal, donde se asume que prevalece el estado de silencio. El código implementado para este estado del fichero *vad.c* del switch case del método *vad* se muestra a continuación:

```
case ST_INIT:
  if (count < firstFrames) {
    count++;
    k0 = k0 + pow(10, f.p / 10);
  }
  else {
    k0 = 10 * log10(k0 / firstFrames);
    offset1 = -0.1 * k0 + 2.0;
    offset2 = -0.033 * offset1 + 2.0;

    k1 = k0 + offset1;
    k2 = k1 + offset2;
    vad_data->state = ST_SILENCE;
  }
  break;
```

Con la comparación entre *count* y *firstFrames*, nos aseguramos de calcular la referencia de potencia del silencio de la señal durante las primeras *firstFrames* muestras. Una vez han transcurrido, se calculan los umbrales k_0, k_1 y k_2 (los valores numéricos escogidos ya se argumentarán en el siguiente apartado correspondiente a la optimización). Finalmente, se pasa al estado de silencio ST_SILENCE, dada la asunción de que los primeros tramos de señal corresponden a silencio.

Vamos a proceder a explicar la implementación de los estados principales: el estado de silencio (ST_SILENCE) y el estado de voz (ST_VOICE). Son los dos estados donde se detecta silencio o voz con determinación, respectivamente. El código correspondiente se muestra debajo:

```
  case ST_SILENCE:
    undefinedFrames = 0;
    if (f.p > k1) vad_data->state = ST_MV;
    break;

  case ST_VOICE:
    undefinedFrames = 0;
    if (f.p < k1) vad_data->state = ST_MS;
    break;
```

Lo primero que se hace en ambos estados es definir los *undefinedFrames* a cero, que son usados para controlar el número de tramas durante los cuales se está en un estado indefinido en el que hay una cierta incerteza. Dado que ST_SILENCE y ST_VOICE no tienen incerteza, se reinician a cero. En el caso del estado de silencio, se mira si la potencia media del tramo es mayor que el umbral k_1, entonces es posible que pase a un estado de voz, por lo que pasa al estado de maybe voice, donde se determinará si realmente corresponde a un tramo de voz. Por el contrario, en el caso de estar en el estado de voz, se mira si la potencia es menor que el umbral k_1, y si es el caso, se pasa al estado de maybe silence.

Los últimos dos estados considerados son aquellos que son considerados como incertidumbre y transición entre los dos estados principales. Estos son maybe voice (ST_MV) y maybe silence (ST_MS). La implementación correspondiente a estos estados se muestra en el siguiente código:

```
case ST_MS:
    if (undefinedFrames < framesMS) undefinedFrames++;
    else
      if (f.p < k1) vad_data->state = ST_SILENCE;
      else vad_data->state = ST_VOICE;
    break;

  case ST_MV:
    if (undefinedFrames < framesMV) undefinedFrames++;
    else
      if (f.p > k2) vad_data->state = ST_VOICE;
      else vad_data->state = ST_SILENCE;
    break;

  case ST_UNDEF:
    break;
  }
```

Para ambos casos, se mira cuantas tramas se lleva en dicho estado de incertidumbre y se compara con un umbral (distinto para cada uno de los estados). El motivo por el que esto es necesario es para evitar cambios de estado muy repentinos y evitar falsas detecciones. Un ejemplo de ello podría ser el siguiente: supongamos que estamos analizando la señal de audio correspondiente a una frase cualquiera. Entre palabras hay un mínimo silencio natural, que no debería considerarse como tal. Si al terminar una palabra se pasa de ST_VOICE a ST_MS y no se espera un cierto tiempo a ver si realmente se trata de silencio o sigue siendo voz, se puede pasar a ST_SILENCE sin que realmente lo sea. La

asignación de dichos umbrales será explicada en el siguiente apartado. Para el caso de maybe silence, una vez pasado el tiempo mínimo asignado en el estado, se compara la potencia con el umbral k_1 y, si es menor a este umbral, entonces se pasa a ST_SILENCE. Si por el contrario, no lo es, se pasa a ST_VOICE. Esto no se correspondría exactamente con la forma de proceder en dicho caso, pero tal y como hemos asignado los valores hemos concluido que el umbral k_0 y k_2 son demasiado extremos, por lo que se obtiene un mejor resultado con el umbral k_1. En realidad, tiene bastante sentido, ya que una vez se han pasado los tramos de incertidumbre necesarios (correspondientes a la mínima duración de silencio entre palabras), un nivel de potencia superior a k_1 prácticamente asegura que se trata de voz y viceversa. Para el caso de maybe voice, se compara la potencia media de tramo con k_2, el umbral necesario para asegurar la detección de voz. En cambio, si no se supera dicho umbral, se considerará que se trata de silencio. Finalmente, el estado indefinido no se va a implementar, ya que no es necesario dado que los dos estados de incertidumbre ya han sido tenidos en cuenta.

Una vez explicados los principales factores del fichero *vad.c*, procedamos a explicar las principales modificaciones en el fichero *main_vad.c* (ambos se encuentran adjuntos en el Anexo). La parte de código fundamental para adaptar la implementación realizada en el fichero *vad.c* es la siguiente:

```
1     /* TODO: print only SILENCE and VOICE labels */
2     /* As it is, it prints UNDEF segments but is should be merge to the
      proper value */
3     if (state != last_state) {
4       if (t != last_t) {
5         if ((last_valid_state != state) && (state == ST_VOICE || state
      == ST_SILENCE)
6             && (last_state == ST_MS || last_state == ST_MV)){
7
8
9             fprintf(vadfile, "%.5f\t%.5f\t%s\n", last_t *
      frame_duration, (t-1) * frame_duration, state2str(last_valid_state))
      ;
10          last_valid_state = state;
11          last_t = t-1;
12        }
13        last_state = state;
14      }
15    }
```

Esta parte se corresponde con la escritura en el fichero con extensión *.vad*, donde se anotan los intervalos de voz y silencio detectados por el programa. Para ello, declaramos una variable llamada *last_valid_state* de tipo *VAD_STATE*, la cual se inicializa a ST_SILENCE (se asume que la señal empieza en silencio). Esta variable se empleará para escribir los tramos de voz y silencio a posteriori; esto es, una vez se cambie de silencio a voz o viceversa, se escribirá el estado anterior y viceversa (solo adoptará uno de estos dos estados). La condición de escritura del último estado válido será la siguiente: si el *state* (devuelto por la función *vad* del fichero *vad.c*) es distinto a *last_state*, el último tramo escrito es distinto al tramo por escribir, y si el último estado válido es distinto al estado actual y el estado actual es ST_VOICE o ST_SILENCE y el último estado (last_state) no

es un estado de incertidumbre (ST_MS o ST_MV), entonces se escribe en el fichero con extensión *.vad*, el tramo correspondiente al último estado válido. El motivo principal por el cual lo hacemos así, es debido a que podría darse el caso de estar en un estado de voz o silencio, pasar al estado de incertidumbre y volver al estado de voz o silencio anterior. En este caso, si no se usara una variable adicional para escribir el último estado válido una vez este cambia, entonces se deberían de escribir dos veces tramos correspondientes al mismo estado. Es por ello que una vez se cambia de voz a silencio o de silencio a voz, aunque sea pasando por los estados intermedios, se escribe el estado anterior al cambio con su intervalo correspondiente).

2.2. Optimice los algoritmos y sus parámetros de manera que se maximice la puntuación de la detección de la base de datos de desarrollo (*db.v4*).

Ya presentados todos los conceptos y la estructura básica del algoritmo, ahora pasaremos a comentar las mejoras y optimizaciones que hemos implementado. A lo largo de la explicación haremos referencia al código de *vad.c* adjuntado en el annexo.

Siguiendo en orden el código, en *vad_open()* se inicializan varias variables a 0, las cuales se mencionarán después. También se inicializan *firstFrames* a 10 para tomar un nivel de referencia de la potencia a partir de las 10 primeras muestras del fichero de audio y así tomar un valor más fiable. Además, probando valores razonablemente pequeños, encontramos que es óptimo que *framesMS* valga 11, haciendo de esta manera que, al entrar en el estado maybe silence y hasta que no pasen 11 tramas, no se plantea la opción de cambiar de estado y, así, evitar detectar falsos silencios causados por pausas cortas o espacios entre palabras. Sin embargo, encontramos que la variable que realizaría la misma función pero para el estado maybe voice, *framesMV*, encuentra su valor óptimo a 0 por lo que realmente no ejerce ningún efecto sobre el código.

Pasando a la función *vad_close()*, decidimos que, al acabar el fichero y en caso de encontrarnos en un estado incierto, decidimos acabar con el estado anterior al de incerteza por el cual hemos pasado. Se ve claramente en las líneas 87 y 88.

Finalmente y etrando en la función de mayor peso a la hora de la implementación, pasamos a comentar él método *vad()*, concretamente la estructura *switch* donde se trata cada estado por separado:

Para el caso inicial, asumiremos que empezamos en silencio, pero antes de pasar a ese estado, permaneceremos en un estado anterior ST_INIT a lo largo de un número *firstFrames* de tramas (en nuestro caso 10) para calcular *k0* a través del promediado de la potencia media de las primeras muestras. Una vez concluidas estas primeras muestras, calculamos los otros dos thresholds *k1* y *k2* en proporción a *k0*, haciendo así que dependa de los parámetros de cada señal a la hora de evaluarla. Una vez hechos los cálculos los cuales se afinaron a base de probar valores que maximizasen los resultados, ahora sí, pasamos al primer estado real que es silencio (ST_SILENCE).

Continuamos y ahora pasamos a analizar qué hacemos en los estados ST_SILENCE y ST_VOICE los cuales, a efectos prácticos, se tratan de forma bastante similar. En ambos casos se reestablece el número de tramas indefinidas a 0 y, en caso de bajar (en el caso de ST_VOICE) o subir del threshold (en el caso de ST_SILENCE) intermedio *k1*, se pasará al estado de incertidumbre correspondiente.

Por último, para los casos de incertidumbre, se tratan también de manera bastante similar. En el caso de maybe silence (por lo que provendremos del estado ST_VOICE), como ya hemos comentado varias veces, permanecemos las primeras muestras sin molestarnos en tomar decisiones. Una vez superado este transitorio, comprobaremos si estamos por debajo del threshold intermedio *k1* y, si es así, cambiamos al estado ST_SILENCE mientras que en caso contrario regresamos al estado anterior ST_VOICE.

En contraposición, en el estado maybe voice no se emplea ingún transitorio ya que *framesMV* está inicializado a 0 así que se pasa directamente a la toma de decisiones. En caso de superar el threshold superior *k2* se cambia al estado ST_VOICE pero si esta condición no se satisface, se permanece en el estado anterior de silencio.

Empleando las técnicas y tomando las consideraciones descritas, hemos obtenido los siguientes resultados:

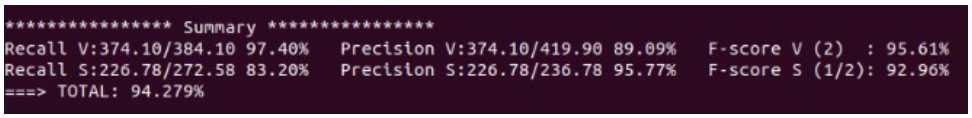

Figura 2: Resultado sobre la base de datos de audios proporcionada.

3. Conclusión

Tras haber entendido, analizado y explicado tanto los conceptos como el código que atañe a esta práctica, hemos observado uno resultados bastante satisfactorios alcanzando un 94,279 % de F-score final evaluando toda la base de datos de ficheros de audio.

Fijandonos mejor en las precisions y recalls tanto de voz como de silencio, vemos que nuestro código actúa excelentemente a la hora de detectar la voz, detectando el 97,4 % de ésta y también con una alta precisión de casi el 90 %, por lo que generamos muy pocos falsos negativos. Esto es importante de destacar ya que suele existir un tradeoff entre precision y recall por lo que obtener ambos valores tan elevados nos confirma la eficacia de nuestro método.

Sin embargo, en el caso de detectar el silencio es donde nuestro código encuentra su mayor flaqueza, obteniendo tan solo un 83,2 % de recall a la hora de detectar silencio. Por contra, la precisión en la detección de silencio es mucho más elevada, superando el 95 % y haciendo evidente el tradeoff del que hablábamos anteriormente.

Con estos valores vemos que aún nos quedaría algún margen de mejora en lo que a la detección del silencio se refiere. También cabe resaltar que los datos disponibles constaban de una variabilidad importante y cabida al error humano a la hora de etiquetar los datos a mano por lo que, dados los algorítmos y herramientas empleadas, consideramos estos resultados más que satisfactorios.

4. Anexo

4.1. *vad.c*

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "pav_analysis.h"

#include "vad.h"

const float FRAME_TIME = 10.0F; /* in ms. */

int count;
int firstFrames;
float k0;
float k1;
float k2;
float offset1, offset2;
int undefinedFrames;
int framesMS, framesMV;


/*
   As the output state is only ST_VOICE, ST_SILENCE, or ST_UNDEF,
   only this labels are needed. You need to add all labels, in case
   you want to print the internal state in string format */

const char *state_str[] = {
  "UNDEF", "S", "V", "MV", "MS", "INIT"
};

const char *state2str(VAD_STATE st) {
  return state_str[st];
}

/* Define a datatype with interesting features */
typedef struct {
  float zcr;
  float p;
  float am;
} Features;

/*
 * TODO: Delete and use your own features!
 */

Features compute_features(const float *x, int N) {
  /*
   * Input: x[i] : i=0 .... N-1
   * Ouput: computed features
   */
  /*
   * DELETE and include a call to your own functions
```

```
51     *
52     * For the moment, compute random value between 0 and 1
53     */
54   Features feat;
55   feat.p = compute_power(x, N);
56   feat.am = compute_am(x, N);
57   feat.zcr = compute_zcr(x, N, (float)1000*N/FRAME_TIME);
58   return feat;
59 }
60 
61 /*
62  * TODO: Init the values of vad_data
63  */
64 
65 VAD_DATA * vad_open(float rate) {
66   VAD_DATA *vad_data = malloc(sizeof(VAD_DATA));
67   vad_data->state = ST_INIT;
68   vad_data->sampling_rate = rate;
69   vad_data->frame_length = rate * FRAME_TIME * 1e-3;
70 
71   count = 0;
72   firstFrames = 10;
73   k0 = 0.0;
74 
75   undefinedFrames = 0;
76 
77   framesMS = 11;
78   framesMV = 0;
79 
80   return vad_data;
81 }
82 
83 VAD_STATE vad_close(VAD_DATA *vad_data) {
84   /*
85    * TODO: decide what to do with the last undecided frames
86    */
87   if (vad_data->state == ST_MS) vad_data->state = ST_VOICE;
88   if (vad_data->state == ST_MV) vad_data->state = ST_SILENCE;
89   VAD_STATE state = vad_data->state;
90 
91   free(vad_data);
92   return state;
93 }
94 
95 unsigned int vad_frame_size(VAD_DATA *vad_data) {
96   return vad_data->frame_length;
97 }
98 
99 /*
100  * TODO: Implement the Voice Activity Detection
101  * using a Finite State Automata
102  */
103 
104 VAD_STATE vad(VAD_DATA *vad_data, float *x) {
105 
```

```
106   /* TODO: You can change this, using your own features,
107    * program finite state automaton, define conditions, etc.
108    */
109
110   Features f = compute_features(x, vad_data->frame_length);
111   vad_data->last_feature = f.p; /* save feature, in case you want to
       show */
112
113   switch (vad_data->state) {
114   case ST_INIT:
115     if (count < firstFrames) {
116       count++;
117       k0 = k0 + pow(10, f.p / 10);
118     }
119     else {
120       k0 = 10 * log10(k0 / firstFrames);
121       offset1 = -0.1 * k0 + 2.0;
122       offset2 = -0.033 * offset1 + 2.0;
123
124       k1 = k0 + offset1;
125       k2 = k1 + offset2;
126       vad_data->state = ST_SILENCE;
127     }
128     break;
129
130   case ST_SILENCE:
131     undefinedFrames = 0;
132     if (f.p > k1) vad_data->state = ST_MV;
133     break;
134
135   case ST_VOICE:
136     undefinedFrames = 0;
137     if (f.p < k1) vad_data->state = ST_MS;
138     break;
139
140   case ST_MS:
141     if (undefinedFrames < framesMS) undefinedFrames++;
142     else
143       if (f.p < k1) vad_data->state = ST_SILENCE;
144       else vad_data->state = ST_VOICE;
145     break;
146
147   case ST_MV:
148     if (undefinedFrames < framesMV) undefinedFrames++;
149     else
150       if (f.p > k2) vad_data->state = ST_VOICE;
151       else vad_data->state = ST_SILENCE;
152     break;
153
154   case ST_UNDEF:
155     break;
156   }
157
158
159   if (vad_data->state == ST_SILENCE || vad_data->state == ST_VOICE ||
```

```
      vad_data->state == ST_MS || vad_data->state == ST_MV)
    return vad_data->state;
  else
    return ST_UNDEF;
}

void vad_show_state(const VAD_DATA *vad_data, FILE *out) {
  fprintf(out, "%d\t%f\n", vad_data->state, vad_data->last_feature);
}
```

4.2. *main_vad.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <sndfile.h>

#include "vad.h"
#include "vad_docopt.h"

#define DEBUG_VAD 0x1

int main(int argc, char *argv[]) {
  int verbose = 0; /* To show internal state of vad: verbose =
   DEBUG_VAD; */

  SNDFILE *sndfile_in, *sndfile_out = 0;
  SF_INFO sf_info;
  FILE *vadfile;
  int n_read = 0, i;

  VAD_DATA *vad_data;
  VAD_STATE state, last_state, last_valid_state = ST_SILENCE;

  float *buffer, *buffer_zeros;
  int frame_size;         /* in samples */
  float frame_duration;   /* in seconds */
  unsigned int t, last_t; /* in frames */

  char  *input_wav, *output_vad, *output_wav;

  DocoptArgs args = docopt(argc, argv, /* help */ 1, /* version */ "2.0
   ");

  verbose    = args.verbose ? DEBUG_VAD : 0;
  input_wav  = args.input_wav;
  output_vad = args.output_vad;
  output_wav = args.output_wav;

  if (input_wav == 0 || output_vad == 0) {
    fprintf(stderr, "%s\n", args.usage_pattern);
    return -1;
  }

  /* Open input sound file */
```

```
41  if ((sndfile_in = sf_open(input_wav, SFM_READ, &sf_info)) == 0) {
42    fprintf(stderr, "Error opening input file: %s\n", input_wav);
43    return -1;
44  }
45
46  if (sf_info.channels != 1) {
47    fprintf(stderr, "Error: the input file has to be mono: %s\n",
      input_wav);
48    return -2;
49  }
50
51  /* Open vad file */
52  if ((vadfile = fopen(output_vad, "wt")) == 0) {
53    fprintf(stderr, "Error opening output vad file: %s\n", output_vad);
54    return -1;
55  }
56
57  /* Open output sound file, with same format, channels, etc. than
      input */
58  if (argc == 4) {
59    if ((sndfile_out = sf_open(output_wav, SFM_WRITE, &sf_info)) == 0)
     {
60      fprintf(stderr, "Error opening output wav file: %s\n", output_wav
     );
61      return -1;
62    }
63  }
64
65  vad_data = vad_open(sf_info.samplerate);
66  /* Allocate memory for buffers */
67  frame_size   = vad_frame_size(vad_data);
68  buffer       = (float *) malloc(frame_size * sizeof(float));
69  buffer_zeros = (float *) malloc(frame_size * sizeof(float));
70  for (i=0; i< frame_size; ++i) buffer_zeros[i] = 0.0F;
71
72  frame_duration = (float) frame_size/ (float) sf_info.samplerate;
73  last_state = ST_UNDEF;
74
75  for (t = last_t = 0; ; t++) { /* For each frame ... */
76    /* End loop when file has finished (or there is an error) */
77    if  ((n_read = sf_read_float(sndfile_in, buffer, frame_size)) !=
      frame_size) break;
78
79    if (sndfile_out != 0) {
80      /* TODO: copy all the samples into sndfile_out */
81
82    }
83
84    state = vad(vad_data, buffer);
85    if (verbose & DEBUG_VAD) vad_show_state(vad_data, stdout);
86
87    /* TODO: print only SILENCE and VOICE labels */
88    /* As it is, it prints UNDEF segments but is should be merge to the
      proper value */
89    if (state != last_state) {
```

```
      if (t != last_t) {
        if ((last_valid_state != state) && (state == ST_VOICE || state
    == ST_SILENCE)
            && (last_state == ST_MS || last_state == ST_MV)){


            fprintf(vadfile, "%.5f\t%.5f\t%s\n", last_t *
    frame_duration, (t-1) * frame_duration, state2str(last_valid_state))
    ;
          last_valid_state = state;
          last_t = t-1;
        }
        last_state = state;
      }
    }

    if (sndfile_out != 0) {
      /* TODO: go back and write zeros in silence segments */
    }
  }

  state = vad_close(vad_data);
  /* TODO: what do you want to print, for last frames? */
  if (t != last_t)
    fprintf(vadfile, "%.5f\t%.5f\t%s\n", last_t * frame_duration, t *
    frame_duration + n_read / (float) sf_info.samplerate, state2str(
    state));

  /* clean up: free memory, close open files */
  free(buffer);
  free(buffer_zeros);
  sf_close(sndfile_in);
  fclose(vadfile);
  if (sndfile_out) sf_close(sndfile_out);
  return 0;
}
```