Desarrollo de sistema para calibrar electromiogramas de forma automática

D. Anchuela Cantarero

Ingeniería Biomédica, Universidad San Pablo CEU, Madrid, España, d.anchuela@usp.ceu.es

Resumen

Este documento explica el proceso de diseño y creación de un sistema para la calibración de electromiogramas de forma automática, utilizando para ello las librerías de Python “Numpy” para un tratamiento de datos más cómodo, Scikit -learn para hacer uso de los algoritmos de machine learning y, por último, Matplotlib para el estudio de los datos de acuerdo con su visualización. El estudio se divide en tres partes principales: la primera es el procesado de la señal de EMG recogida a partir de una placa Bitalino, para obtener una muestra simplificada de la señal original. La segunda y en la que nos vamos a centrar más, es en la comparación de diferentes herramientas para el preprocesamiento de los datos, así como de métodos de agrupamiento para poder elegir la opción más optima, utilizando para ello el Davis Boulding Score. Por último, implementaremos la mejor opción en el juego T-Rex de Google, que en este caso funcionará a partir de señales de EMG producidas por el jugador, con el fin de que el algoritmo funcione para ajustar automáticamente la intensidad necesaria de fuerza para hacer saltar al dinosaurio.

Palabras clave: Electromiograma, machine learning, Bitalino, clustering.

# Recogida de muestras de electromiograma.

Para poder realizar un correcto estudio de qué técnicas de preprocesado de datos y de agrupamiento ofrecen un mejor rendimiento, es necesario obtener varias muestras de EMG de diferentes personas, para posteriormente promediar la puntuación obtenida, y estar seguros de que la opción escogida es la mejor para diferentes tipos de muestras.

Las muestras de EMG las recogemos utilizando el software *Open Signals,* una placa *Bitalino,* y tres electrodos*.* Para ello conectamos por Bluetooth la *Bitalino* al ordenador. Posteriormente la configuramos desde el *Open Signals* introduciendo la dirección MAC de la placa, y especificamos que vamos a recoger señales de EMG. Por otra parte, conectamos los electrodos a la terminal de EMG de la placa, y nos los ponemos en el brazo de la siguiente manera:

* Electrodos positivo y negativo conectados en dos puntos del antebrazo para recoger la diferencia de potencial del músculo al contraer.
* Electrodo neutro haciendo contacto con una zona del cuerpo sin actividad eléctrica como puede ser el codo.

Grabamos una muestra de actividad mioelectrica, y la guardamos como *.txt* para analizarla posteriormente.

Antes de pasar a analizar la señal, debemos obtener los datos de la señal simplificada ejecutando nuestro fichero *run\_featurizers.py* desde la consola. Esto es así ya que una señal que representa una contracción en un EMG presenta muchas oscilaciones durante la contracción. En nuestro caso, estas oscilaciones nos complicarían mucho el trabajo ya que, como necesitamos analizar los puntos de esta señal, nos podemos encontrar que, en una contracción, el valor de EMG es 0 debido a que está oscilando. Haciendo este procesado de señal, nos quedamos con la señal simplificada en donde solamente se distingue si hay contracción o no (ver figura 1).

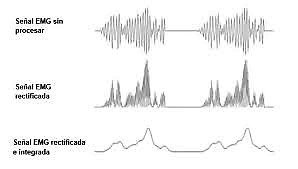


Figura 1: Ilustración esquemática del procesado de señal de EMG.

# Preprocesamiento de los datos

Una vez tenemos las muestras de la señal simplificada de EMG, necesitamos preprocesarlas para optimizar el funcionamiento del algoritmo de *clustering*.

## Escalamiento [1]

El primer paso, que es obligatorio antes de iniciar con el preprocesado de los datos, es hacer un escalamiento.

Esto es necesario ya que, en general, vamos a hacer un estudio de los datos, fijándonos muchas veces en la posición de nuestros puntos. Para realizar correctamente diferentes comparaciones, los puntos deben estar en la misma magnitud.

Para normalizar los datos, debemos ejecutar estas líneas de código sobre nuestros datos:

from sklearn import preprocessing

d\_escalados = preprocessing.minmax\_scale(datos)

Lo que nos devolverá los datos que teníamos, pero ahora entre 0 y 1.

## Transformación de variable [2]

El objetivo de este punto es aplicar una transformación matemática a nuestros datos para intentar que adopten una forma lo más parecida posible a una distribución normal. Este tipo de tratamiento para los datos es muy útil para corregir problemas de posibles valores atípicos y para reducir la dependencia entre media y varianza. Lo que acaba provocando una gran mejora en el funcionamiento de los algoritmos de *clustering*.

Para este estudio probaremos los siguientes tipos de transformaciones:

1. Transformación logarítmica:

Datos\_log = np.log(0.0000001+datos)

Lo que logramos con esto es forzar a nuestros datos a coger forma de gaussiana, evitando el log (0) con ese sumando de 0.0000001.

1. Transformación de raíz:

Datos\_raiz = np.sqrt(0.01+datos)

En este caso, sustituimos cada dato que tenemos por su raíz cuadrada. Como la raíz de 0 es 0, no sería estrictamente necesario sumar 0.01, pero lo hacemos igualmente, porque de otra forma, los datos que son 0 no se verían transformados.

1. Transformación recíproca:

Datos\_reci = 1/(0.01+datos)

En este caso transformamos cada dato por su inverso, aplicando el sumando de 0.01 para evitar la división por 0 en los casos que se dé.

Realmente, solo por definición, la transformación logarítmica es la que mejor debería funcionar con nuestros datos ya que funciona mejor con datos de medida. La transformación de raíz funciona mejor para datos de conteo, y la transformación recíproca funciona mejor para tasas. De todas formas, es interesante estudiar el rendimiento de las 3.

## Reducción de dimensionalidad.

Los sets de datos que obtenemos del archivo de simplificación de la señal de EMG, tienen en realidad varias dimensiones que se utilizan para formar cada punto del EMG. El estudio de estas dimensiones por separado no tiene demasiado significado, pero sin embargo se tienen en cuenta a nivel de computación. De esta manera, muchas veces se da que hay dimensiones adicionales que en realidad son combinaciones lineales de otras dimensiones, por lo que puede ser importante aplicar una reducción de dimensionalidad para optimizar los cálculos, teniendo en cuenta la cantidad de información útil que se podría perder. Para ello aplicaremos las siguientes instrucciones:

from sklearn.decomposition import PCA

pca= PCA(n\_components=1)

datos = pca.fit\_transform(datos)

## Separación de datos de prueba y entrenamiento

El funcionamiento de los modelos de *clustering* se basa en pasarle una serie de datos al modelo para entrenarlo, analizándolos y definiendo los grupos, de manera que más adelante le podamos pasar nuevos datos, y el algoritmo sea capaz de clasificarlos en esos grupos para los cuales ya ha sido entrenado. Para que esto funcione correctamente, no podemos pasarle nuestro archivo de datos para entrenar, y luego volverle a pasar esos datos para que prediga a qué grupo pertenecen ya que el modelo va a predecirlos perfectamente porque esos son los datos para los cuales ha sido entrenado, en este caso se estaría produciendo el conocido *overfiting* o sobreajuste de los datos.

En este sentido, debemos hacer una separación entre los datos que son para entrenar al modelo y los datos que se utilizan para predecir. Para cada muestra diremos que utilice un 30% de los datos para entrenar y el otro 70% para predecir. Esto lo hacemos con las siguientes sentencias:

From sklearn.model\_selection import train\_test\_split

X\_train,X\_test= train\_test\_split(x,test\_size=0.3)

# *Clustering*

Una vez tenemos ya nuestros datos separados, necesitamos emplear nuestro modelo de *clustering*, que será el que determinará los diferentes grupos en los que se detecta que se separan nuestros datos.

Hay muchos algoritmos de *clustering* diferentes, pero recordando cuál es nuestro propósito, debemos hacer una selección sobre cuáles nos serán útiles y cuáles no. En nuestro caso, vamos a emplear el sistema para hacer saltar el dinosaurio del juego, dependiendo de si el usuario está contrayendo o no el músculo. En consecuencia, nosotros solo tenemos dos posibles grupos en los que se tienen que dividir los datos (si hay o no contracción). Por eso, de todos los algoritmos de agrupamiento que hay, solo podemos quedarnos con aquellos a los cuales, nosotros le podemos pasar como argumento el número de *clusters* que tiene que haber. Son los siguientes:

* *KMeans*
* *Gaussian Mixture Model*
* Agrupamiento jerárquico

## *KMeans* [3]

Este algoritmo selecciona aleatoriamente 2 puntos (porque en nuestro caso tenemos 2 clusters) de nuestro set de datos. Estos 2 puntos serán los centroides de nuestros *clusters*. Posteriormente evalúa los demás puntos para asignarlos a un grupo, determinando la distancia más cercana que hay entre ese punto en concreto con cada uno de los *clusters* (ver figura 2).

Gráfico, Gráfico de dispersión

Descripción generada automáticamenteGráfico, Gráfico de dispersión

Descripción generada automáticamente

Figura 2: Ejemplo de funcionamiento de *KMeans.*

Como hemos comentado, la selección de los *clusters* es aleatoria, pero el análisis se va a repetir hasta que en análisis consecutivos se obtengan los mismos resultados, lo que significará que ese es el mejor agrupamiento posible.

Para implementarlo teniendo en cuenta la separación entre datos de prueba y datos de entrenamiento, tenemos que ejecutar las siguientes líneas de código:

from sklearn.cluster import KMeans

X\_train,X\_test= train\_test\_split(x,test\_size=0.3)

algoritmo= KMeans(n\_clusters=2, init='k-means++', max\_iter=300, n\_init=100)

X\_train = pca.fit\_transform(X\_train)

algoritmo.fit(X\_train)

X\_test = pca.transform(X\_test)

predicciones=algoritmo.predict(X\_test)

## *Gaussian Mixture Model* [4]

Este tipo de algoritmo es muy parecido a *KMeans* pero nos ofrece un poco más de flexibilidad ya que estamos suponiendo que los datos se distribuyen de forma gaussiana (lo cual puede ser muy bueno ya que una de las transformaciones era para hacer que nuestros datos se ajusten mejor a una gaussiana) mientras que en *KMeans* se presupone que los datos se distribuyen de forma circular usando la media. Esto se traduce en que en este caso podemos tener *clusters* de forma elíptica ya que ahora contamos con una desviación estándar tanto en el eje X como en el eje Y (ver figura 3).

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

Figura 3: Ejemplo de funcionamiento de *Gaussian Mixture Model.*

En este caso, a diferencia del anterior, se va a calcular la probabilidad que tiene cada punto de pertenecer a un *cluster* o a otro. Cuanto más cerca esté el punto del centroide del *cluster*, más probable será que pertenezca a él.

Por último, repetimos el proceso para un nuevo conjunto para maximizar las probabilidades de los puntos, optimizando así el algoritmo.

De una manera muy similar a *KMeans*, este modelo se implementa de la siguiente manera.

from sklearn.mixture import GaussianMixture

X\_train,X\_test= train\_test\_split(x,test\_size=0.3)

algoritmo= GaussianMixture(n\_components=2, covariance\_type="full", n\_init=100)

X\_train = pca.fit\_transform(X\_train)

algoritmo.fit(X\_train)

X\_test = pca.transform(X\_test)

predicciones=algoritmo.predict(X\_test)

## Agrupamiento Jerárquico [5]

Inicialmente, el modelo trata a cada dato como un *cluster* diferente. Posteriormente se van fusionando los *clusters* que están más cerca entre sí, hasta que llegamos al numero de *clusters* deseados. Para determinar la distancia entre *clusters* se puede hacer con distintas métricas, pero nosotros utilizaremos la distancia euclídea (ver figura 4).

Gráfico, Gráfico de burbujas

Descripción generada automáticamente

Figura 4: Ejemplo de funcionamiento de agrupamiento jerárquico*.*

Por otra parte, este algoritmo tiene una diferencia fundamental con respecto a los otros dos y es que no incluye la funcionalidad de predecir la clasificación para nuevos datos, por lo que, en la implementación, deberemos incluir también un clasificador, que en este caso será el *KNeighborsClassifier,* cuyo funcionamiento es muy similar al de *KMeans* en el sentido de que se basa en la distancia entre *cluster* y datos.

Dicho esto, pasamos a ver la implementación de este algoritmo:

from sklearn.mixture import AgglomerativeClustering

X\_train,X\_test= train\_test\_split(x,test\_size=0.3)

agrupameinto\_jerárquico = AgglomerativeClustering(n\_clusters=2, affinity='euclidean', linkage='ward')

X\_train = pca.fit\_transform(X\_train)

agrupameinto\_jerárquico.fit(X\_train)

X\_test = pca.transform(X\_test)

Clasificador= KNeighborsClassifier(n\_neighbors=2, metric= 'minkowski', p=2)

Clasificador.fit(X\_train, agrupameinto\_jerárquico.labels)

predicciones= Clasificador.predict(X\_test)

# Estudio del número de *clusters* [6]

Llegados a este punto, aunque nosotros tenemos que utilizar obligatoriamente dos *clusters* ya que solo tenemos dos posibles estados, es interesante estudiar si nuestros datos realmente tienen las propiedades necesarias para ser separados exactamente en dos grupos, o si por el contrario, aunque nosotros estemos forzando el número de *clusters*, el número para un agrupamiento óptimo es otro.

Para visualizar esto, podemos utilizar el algoritmo del codo, en el que se valora la inercia obtenida por el método KMeans. En este tipo de algoritmo se ve representado como el mayor cambio en la inercia se da cuando tenemos el número óptimo de *clusters*. Por esa forma que toma la línea en el cambio brusco, a este algoritmo se le da el nombre de “algoritmo del codo” (ver figura 5).

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Figura 5: Resultado gráfico del algoritmo del codo*.*

inercia=[]

for i in range(1,8):

algoritmo= KMeans(n\_clusters=i, init ='k-means++', max\_iter=300, n\_init=10)

algoritmo.fit(xi)

inercia.append(algoritmo.inertia\_)

En este caso se ve como el cambio más brusco está en el número 2, lo que significa que efectivamente es 2 el número óptimo de *clusters* para nuestra muestra.

También podemos hacer un estudio del número óptimo de *clusters* basándonos en el algoritmo de agrupamiento jerárquico, es decir teniendo en cuenta la distancia entre los distintos “*subclusters*”. En este caso utilizaremos como medida la distancia euclídea en vez de la inercia (ver figura 6).

Imagen que contiene Histograma

Descripción generada automáticamente

Figura 6: Resultado gráfico del estudio del dendrograma*.*

import scipy.cluster.hierarchy as shc

dendrograma= shc.dendrogram(shc.linkage(x,method='ward'))

En este caso también observamos como la distancia más larga se da cuando tratamos de unir los 2 *clusters* principales, por lo que también es 2 el número óptimo.

# Elección de la mejor opción

Llegados a este punto, toca hacer un análisis de cuál de todas las opciones que hemos estudiado nos ofrece un rendimiento óptimo a la hora de agrupar los datos en 2 *clusters*. Para ello, haremos uso del conocido *Davies* *Bouldin* *Score*[7]. Esta es una métrica que evalua la calidad del agrupamiento ofreciendo una puntuación entre 0 y 1. Las puntuaciones más altas son aquellas que se otorgan a los agrupamientos con una probabilidad de error más alta, mientras que las puntuaciones más próximas a 0, se otorgan cuando los datos se encontraban notablemente agrupados y en consecuencia el *clustering* es muy fiable. El *Davies* *Bouldin* *Score* se implementa de la siguiente manera:

db\_index=davies\_bouldin\_score(X\_test, predicciones)

En este caso, tenemos que estudiar la puntuación de todas las combinaciones de todos los modelos de *clustering*, con la posibilidad de hacer o no reducción de dimensionalidad y con todas las opciones de transformación de variable. De esta manera obtendremos una combinación que nos dará un *Davies Boulding Score* más bajo, y por tanto la que determinaremos como la mejor para implementarla en el juego del dinosaurio.

Para hacer esto, hacemos uso de varios bucles anidados para ir comprobando cada posible combinación para cada muestra de EMG que tenemos. Posteriormente pasamos a promediar las puntuaciones obtenidas de cada muestra con respecto a cada combinación para obtener un resultado general sobre el cual poder sacar conclusiones verídicas.

Obtenemos los resultados siguientes:

Gráfico, Histograma

Descripción generada automáticamente

Figura 7: Resultado del promedio de cada combinación.

|  |  |  |
| --- | --- | --- |
| Promedio de cada combinación | | |
| Combinación | Código del gráfico | D. B. Score |
| Transformación recíproca, con reducción de dimensionalidad y *Gaussian* *Mixture* *m*. | Rcip-S\_RDim-GMix | 0.127341 |
| Transformación recíproca, con reducción de dimensionalidad y Agrupamiento jerárquico | Rcip-S\_RDim-AJer | 0.128295 |
| Transformación recíproca, con reducción de dimensionalidad y *KMeans* | Rcip-S\_RDim-KM | 0.128231 |
| Transformación recíproca, sin reducción de dimensionalidad y *Gaussian Mixture m*. | Rcip-N\_RDim-GMix | 0.245182 |
| Transformación recíproca, sin reducción de dimensionalidad y Agrupamiento jerárquico | Rcip-N\_RDim-AJer | 0.250003 |
| Transformación recíproca, sin reducción de dimensionalidad y *KMeans* | Rcip-N\_RDim-KM | 0.263444 |
| Sin transformación, con reducción de dimensionalidad y *Gaussian Mixture m*. | N\_G-S\_RDim-GMix | 0.600014 |
| Sin transformación, con reducción de dimensionalidad y Agrupamiento jerárquico | N\_G-S\_RDim-AJer | 0.408509 |
| Sin transformación, con reducción de dimensionalidad y *KMeans* | N\_G-S\_RDim-KM | 0.400273 |
| Sin transformación, sin reducción de dimensionalidad y *Gaussian* *Mixture* *m.* | N\_G-N\_RDim-GMix | 0.608986 |
| Sin transformación, sin reducción de dimensionalidad y Agrupamiento jerárquico | N\_G-N\_RDim-AJer | 0.483506 |
| Sin transformación, sin reducción de dimensionalidad y *KMeans* | N\_G-N\_RDim-KM | 0.456339 |
| Transformación logarítmica, con reducción de dimensionalidad y *Gaussian* *Mixture* *m*. | L-S\_RDim-GMix | 0.061909 |
| Transformación logarítmica, con reducción de dimensionalidad y Agrupamiento jerárquico | L-S\_RDim-AJer | 0.066555 |
| Transformación logarítmica, con reducción de dimensionalidad y *KMeans* | L-S\_RDim-KM | 0.063993 |
| Transformación logarítmica, sin reducción de dimensionalidad y *Gaussian Mixture m*. | L-N\_RDim-GMix | 0.138259 |
| Transformación logarítmica, sin reducción de dimensionalidad y Agrupamiento jerárquico | L-N\_RDim-AJer | 0.140532 |
| Transformación logarítmica, sin reducción de dimensionalidad y *KMeans* | L-N\_RDim-KM | 0.147391 |
| Transformación de raíz, con reducción de dimensionalidad y *Gaussian Mixture m*. | SQRT-S\_RDim-GMix | 0.530139 |
| Transformación de raíz, con reducción de dimensionalidad y Agrupamiento jerárquico | SQRT-S\_RDim-AJer | 0.354500 |
| Transformación de raíz, con reducción de dimensionalidad y *KMeans* | SQRT-S\_RDim-KM | 0.368275 |
| Transformación de raíz, sin reducción de dimensionalidad y *Gaussian Mixture m*. | SQRT-N\_RDim-GMix | 0.441626 |
| Transformación de raíz, sin reducción de dimensionalidad y Agrupamiento jerárquico | SQRT-N\_RDim-AJer | 0.424745 |
| Transformación de raíz, sin reducción de dimensionalidad y *KMeans* | SQRT-N\_RDim-KM | 0.444387 |
| 1. Estas puntuaciones cambian ligeramente en cada ejecución del programa ya que los datos de entrenamiento y prueba se escogen aleatoriamente siguiendo la proporción establecida. | | |

De esta manera, como ya habíamos introducido anteriormente, observamos que la mejor combinación es haciendo una transformación logarítmica a los datos, aplicando la reducción de dimensionalidad y utilizando el modelo de *Gaussian* *Mixture* *model*. Este resultado tiene mucho sentido ya que, como explicamos anteriormente, el *Gaussian* *Mixture* *model* asume que los datos están *gaussianizados*, y en este caso es así ya que previamente han sido transformados con el logaritmo. Además, se puede observar como en todos los casos, la reducción de dimensionalidad ofrece una mejora en el rendimiento del *clustering*.

# Implementación del modelo en el juego T-Rex de Google

Explicación implementación

# Agradecimientos

Por último, me gustaría agradecer a mi tutor Tino toda la dedicación, conocimiento y apoyo que me ha transmitido durante todo el transcurso de este proyecto.

# Referencias

1. sklearn.preprocessing.minmax\_scale. scikit-learn. Published 2021. Accessed November 21, 2021. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.minmax\_scale.html
2. ‌ Pita Fernández, S., & Pértega Díaz, S. (1997). Relación entre variables cuantitativas. Cad Aten Primaria, 4, 141-4.
3. Cambronero, C. G., & Moreno, I. G. (2006). Algoritmos de aprendizaje: knn & kmeans. Intelgencia en Redes de Comunicación, Universidad Carlos III de Madrid, 23.
4. Instituto de Informatica UACh. [INFO337] Modelos de mezcla de Gaussianas parte 1. YouTube. Published online December 10, 2020. Accessed November 21, 2021. https://www.youtube.com/watch?v=s3sTSA\_GXV8&t=5106s&ab\_channel=InstitutodeInformaticaUACh
5. Ligdi González. Algoritmo Agrupamiento Jerárquico - Práctica - Aprende IA. Aprende IA. Published November 10, 2020. Accessed November 21, 2021. https://aprendeia.com/algoritmo-agrupamiento-jerarquico-practica/
6. ‌ Villardón, J. L. V. (2007). Introducción al análisis de clúster. Departamento de Estadística, Universidad de Salamanca. 22p.
7. sklearn.metrics.davies\_bouldin\_score. scikit-learn. Published 2021. Accessed November 21, 2021. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies\_bouldin\_score.html#