Detección de gestos mediante clasificación de señales EMG

Germán Urrea

12 de julio de 2022

Introducción

El objetivo de este proyecto es entrenar clasificadores para gestos de manos, esto a partir de datos obtenidos de señales mioeléctricas obtenidos del dataset "EMG data for gestures Data Set", disponible en el "UCI Machine Learning Repository". Posterior al entrenamiento se comprobará la efectividad de los mejores clasificadores entrenados, para ello se clasificarán muestras de un conjunto de prueba creado para una competencia en la página *kaggle*, creada en el marco del curso EL4106-Inteligencia computacional.

A continuación se resume brevemente el contenido de este informe:

1) **Lectura y exploración de datos**: Se explica el proceso de lectura de los datos, así como una exploración inicial de estos.
2) **Procesamiento de datos**: Se explica la metodología a utilizar para definir el tipo de pre-procesado que se le dará a los datos, la cuál involucra el uso de clasificadores, por lo que también se explican algunos aspectos del entrenamiento de estos.
3) **Entrenamiento de clasificadores**: Se detallan los clasificadores a utilizar para el entrenamiento, el procedimiento de este y los resultados obtenidos en los conjuntos de entrenamiento, validación y prueba. También se realiza un análisis de los resultados obtenidos y posibles mejoras que podrían implementarse para obtener mejores resultados.

1. Lectura y exploración de datos

El dataset está dividido en datos de entrenamiento, validación y prueba. Las características comunes de estos datos es la presencia de 8 canales EMG, contando los datos de entrenamiento/validación con el tiempo de captura en ms y un label que indica el gesto realizado en ese instante.

1.1. Lectura de datos de entrenamiento y validación

Los datos de entrenamiento y validación en un principio corresponden a los mismos datos, ya que la separación se realiza más adelante en el proyecto. Estos datos corresponden a 30 sujetos distintos, con 2 capturas distintas para cada sujeto. Las capturas de cada sujeto se encuentran en carpetas llamadas "$subxx$" en donde xx corresponde al número del sujeto, cada captura se encuentra en archivos de texto llamados "1.txt" y "2.txt".

A continuación se muestra el código utilizado para leer los datos de entrenamiento. Este código retorna una lista de dataframes con los datos de cada captura y la información necesaria para distinguir a cada sujeto y su captura, en caso de ser necesario.

Código 1: Lectura de datos de entrenamiento/validación.

```
def read_data(path):
    '''
    path: Carpeta en donde se encuentran todas las carpetas con la data de los sujetos
     ↪ .
    '''
    d_read = lambda d : pd.read_csv(d, sep='\t', header = 0)
    path_train = path +'/subj{}'
    common_names = ['1.txt', '2.txt']
    data_list = []
    for i in range(1, 31):
        sub_n = str(i).zfill(2) # Número del sujeto
        folder = path_train.format(sub_n) # Ruta de la carpeta
        for file in common_names:
            df_file = d_read(f'{folder}/{file}')
            df_file['subject'] = i
            df_file['capture'] = int(file[0])

            data_list.append(df_file)
    return data_list
```

1.2. Lectura de datos de prueba

Los datos de prueba están contenidos en el archivo "windows_test.csv", que al ser leído contiene filas de 6400 datos, estás filas corresponden a los 8 canales concatenados, en donde cada 800 datos consecutivos corresponden a un canal. Para este dataset no se poseen etiquetas, ya que las predicciones se deben subir a la competencia creada en *Kaggle*.

A continuación se muestra el código utilizado para leer los datos de prueba. Estos se almacenan en un arreglo de *numpy* de dimensiones $(672, 800, 8)$

Código 2: Lectura de datos de prueba.

```
def read_test_data(path_test):

    df_test = pd.read_csv(path_test, sep=",", header=0)
    # Almacenar ventanas 800x8 en un arreglo de numpy
    test = np.array(df_test.drop('Id', axis=1)).reshape(672, 8, 800).transpose((0, 2, 1)
     ↪ )

    return test
```

1.3. Definición de conjuntos de entrenamiento y validación

Para separar los datos etiquetados en entrenamiento y validación primero se seleccionan sujetos para cada propósito mediante la función *train_test_split* de *sklearn*, posteriormente se separa la lista de dataframes creada con anterioridad de acuerdo a los sujetos seleccionados. A continuación se muestra el código utilizado para este propósito.

Código 3: Lectura de datos de prueba.

```
train_sub, val_sub = train_test_split(np.arange(1, 31), train_size = 0.8,
    ↪ random_state = 42)

def train_val_split(df_list, train_sub, val_sub):

  train_list = []
  val_list = []
  for df in df_list:
    if df['subject'].unique() in train_sub:
      train_list.append(df)
    elif df['subject'].unique() in val_sub:
      val_list.append(df)

  return train_list, val_list
```

Es de notar que para este proyecto se ecoge un ratio de 80 % entrenamiento, 20 % validación.

1.4. Visualización de señales

Para poder comprender mejor la naturaleza de los datos, se grafican algunos canales EMG. En este caso se grafican los canales 1, 3 y 5, para el sujeto número 1 en su primera captura. Para cada canal se realizan 2 gráficos Amplitud vs Tiempo [ms]: uno de de ellos es un gráfico de dispersión en donde cada punto posee un color dependiendo del gesto correspondiente (el color púrpura indica que no se realiza ningún gesto), el otro simplemente es un gráfico de lineas.

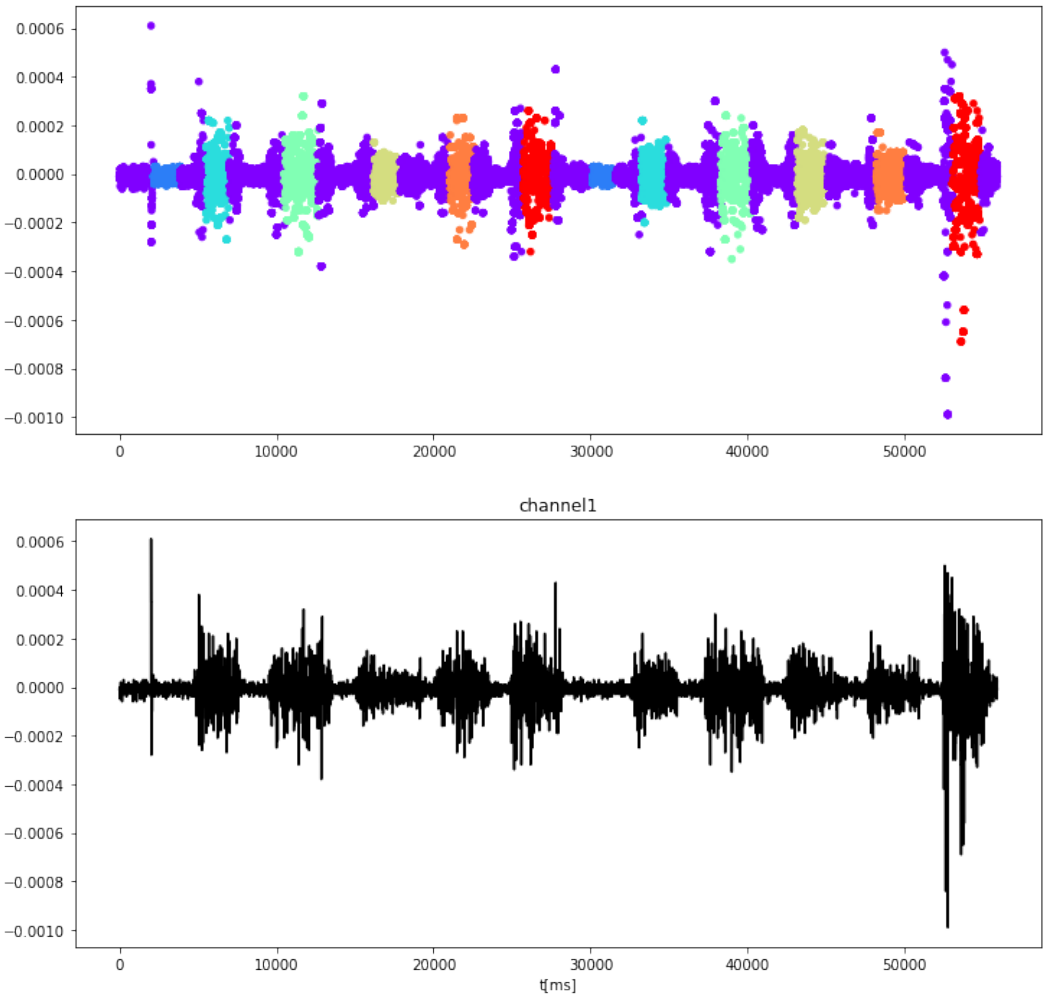

Figura 1: Señal EMG para el canal 1

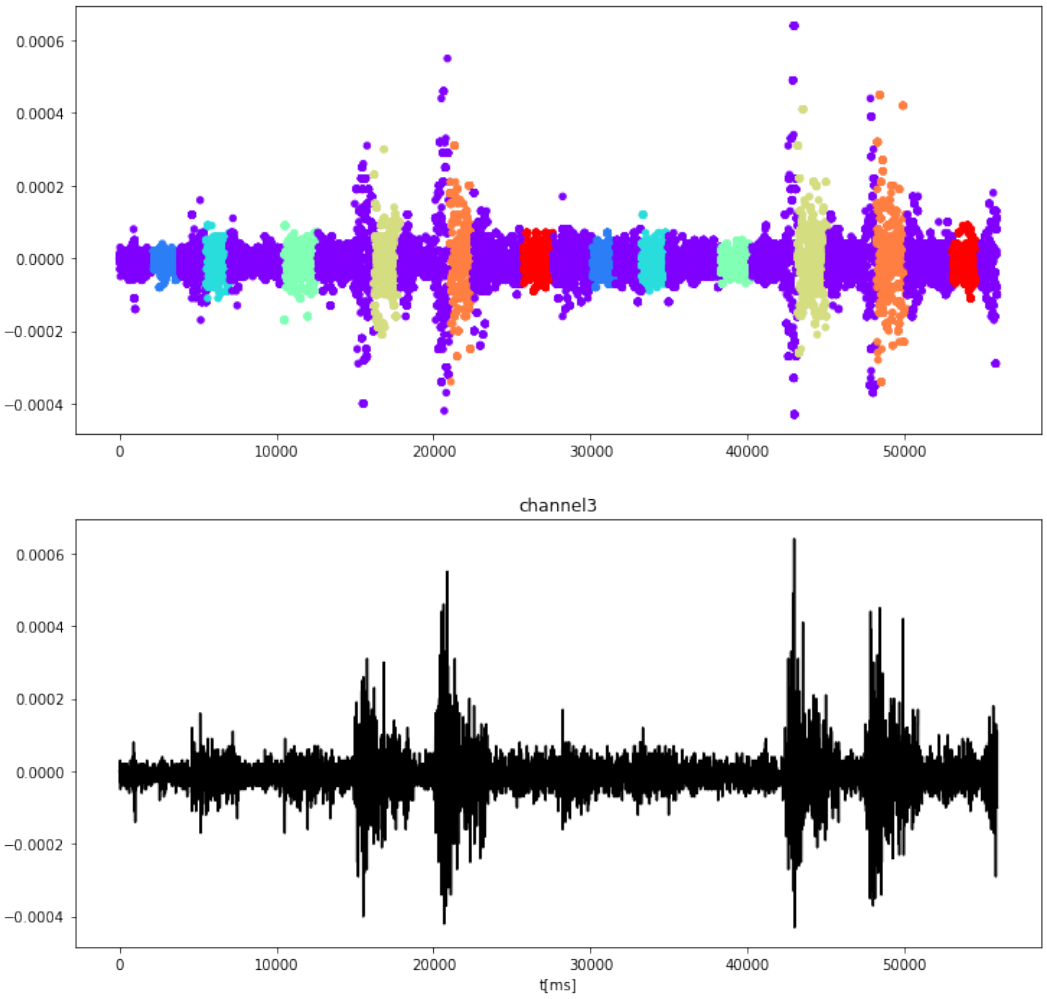

Figura 2: Señal EMG para el canal 3

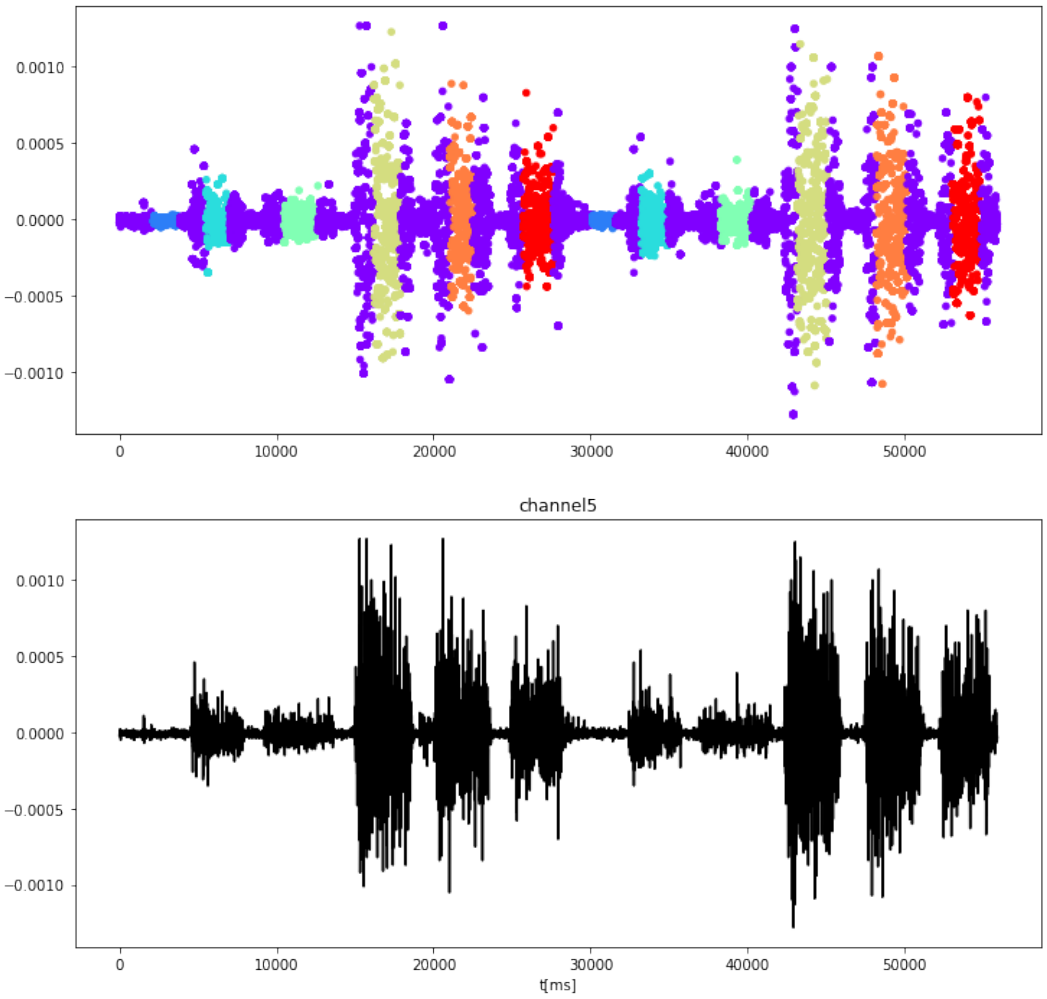

Figura 3: Señal EMG para el canal 5

De esta visualización de señales se puede extraer que cada gesto posee una señal más o menos bien definida en un cierto rango de amplitudes, también se puede observar a simple vista que cada gesto transcurre durante periodos de tiempo similares (indicando data balanceada o poco desbalanceada), también es interesante notar que en varios casos hay un período de tiempo anterior y posterior a la realización del gesto en que la señal toma la forma que tiene cuando se realiza el gesto (pero no se etiqueta como el gesto como tal). Otro aspecto interesante de notar es que en el canal 3 la señal por default (sin gesto) se asemeja a muchas de las señales etiquetadas, esto puede deberse a que la zona medida no presenta cambios significativos al momento de realizar la mayoría de gestos.

Para poder extraer un poco más de información a partir de las señales se calcula la transformada rápida de fourier para los canales anteriores, esto para poder ver como se comportan las señales en el dominio de frecuencia.

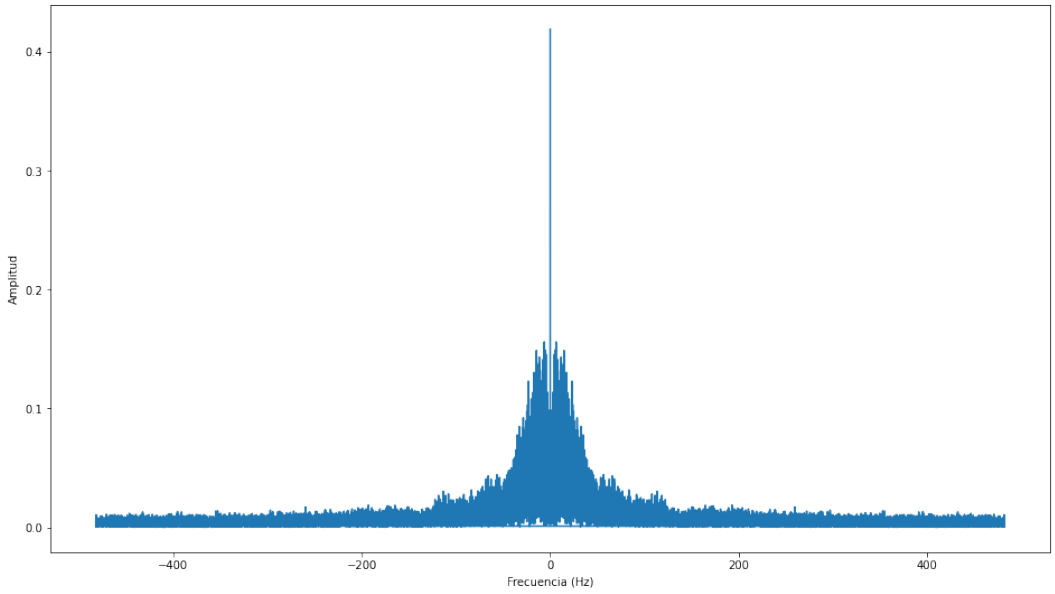

Figura 4: FFT para canal 1

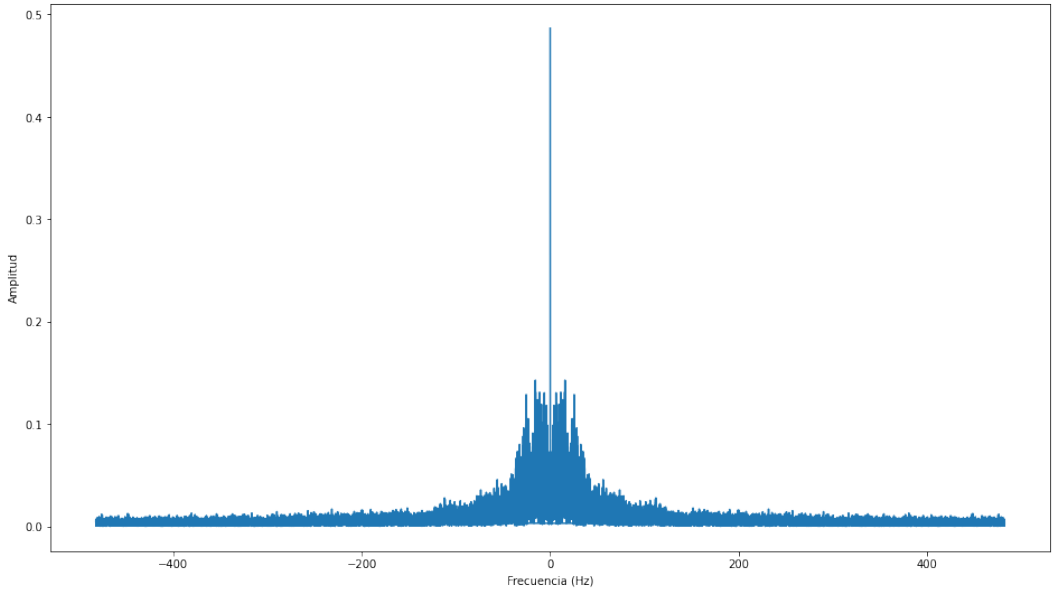

Figura 5: FFT para canal 3

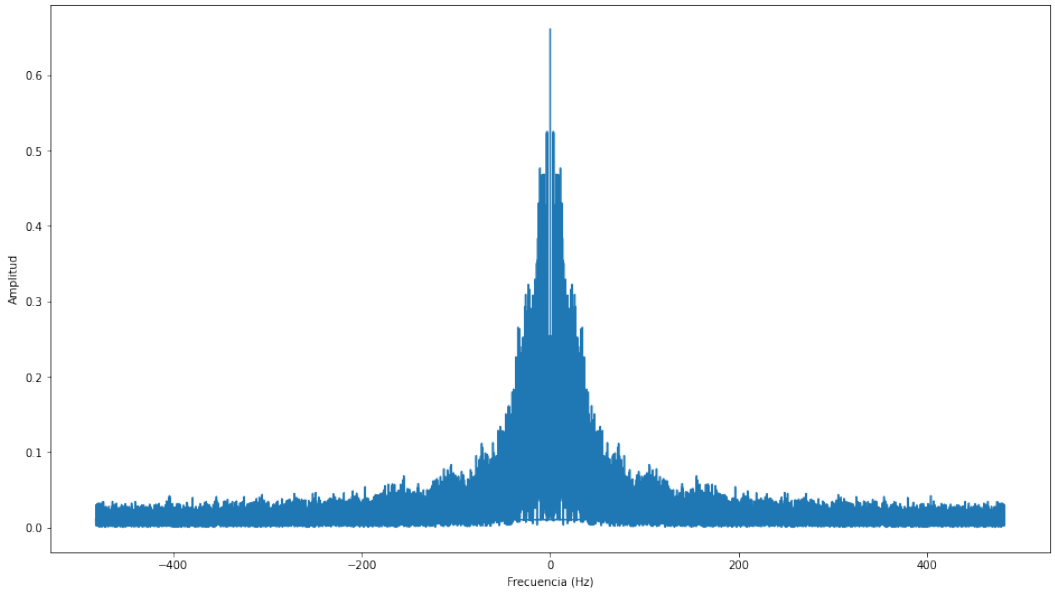

Figura 6: FFT para canal 5

De las transformadas de fourier se puede extraer que la mayor amplitud de cada canal se encuentra en frecuencias cercanas a 0 Hz, esto generalmente indica que la señal EMG no está centrada en 0 en el dominio temporal, para poder centrar las señales de los canales bastaría con restar el promedio de las amplitudes del canal a cada una de las amplitudes. Otro dato que se puede extraer es el rango de frecuencias, las cuáles pueden llegar hasta aprox. 450-500 Hz.

2. Procesamiento de datos

Se busca optimizar el conjunto para mejorar los resultados de clasificación. Por lo tanto para el procesamiento se utilizarán todos aquellos métodos que mejoren los resultados en entrenamientos preliminares. Estos entrenamientos preliminares consisten en entrenar un pequeña selección de clasificadores con el conjunto de entrenamiento/-validación después de aplicar algún método de pre-procesado, manteniendo aquellos cambios que, o mejoren el score de referencia, o no tengan un gran impacto en este pero puedan implicar tiempos menores de entrenamiento y/o una mejor generalización.

Antes de explicar los resultados para cada tipo de pre-procesado, es necesario explicar como se ajustan los datos para su uso en clasificación.

2.1. Generación de ventanas

Dado que existe una gran cantidad de datos, es conveniente analizar las señales de cada gesto por separado, en ventanas cortas de tiempo, de forma de tener un mejor reconocimiento de las características de las señales. Para ello se aplica la técnica del *Sliding Window* a los datos, a grandes rasgos esto se resume en obtener las señales por individuo, gesto y captura, y dividir la señal en ventanas de 800 muestras (aprox. 800 ms), con una intersección de 550 muestras entre ventanas creadas de forma consecutiva, esto último con el objetivo de aprovechar mejor los datos. El resultado de aplicar esta técnica son ventanas de tamaño $(800, n_{channels})$, cada una con un label asignado; es importante señalar que para el proyecto no se toman en cuenta ventanas con labels 0 y 7, dado que el label 0 no implica la realización de ningún gesto, y el label 7 corresponde a un gesto que no fue realizado por todos los sujetos, por lo que no es de interés.

A continuación se incluye el código utilizado para la generación de ventanas y sus respectivos labels.

Código 4: Creación de una lista de ventanas a partir de una lista de dataframes.

```
def create_windows(dataframe, window_size = 800, step = 250):
  windows = []
  start = 0
  while start + window_size < len(dataframe):
      window = dataframe[start:start+800]
      windows.append(window)
      start += step
  return windows

def emg_windows(df_list, window_size = 800, step = 250):
  classes = np.arange(1, 7)
  windows = []
  labels = []
  for df in df_list:
    for cl in classes:
      temp_window = create_windows(df[df['class']==cl].drop('class', axis=1),
      ↪ window_size, step)
```

```
    windows+=temp_window
    labels += [cl]*len(temp_window)
  return windows, labels
```

2.2. Generación de características a partir de ventanas

Para poder obtener la mayor cantidad de información posible de las ventanas creadas, resulta conveniente resumir la información de las señales de cada canal en medidas estadísticas (como promedio, varianza, rango, etc), para su posterior uso como características en el entrenamiento de clasificadores. De esta forma si se tiene una lista de N ventanas de dimensiones $(800, n_{channels})$, mediante el cálculo de características se transforman todos estos datos a un arreglo de dimensiones $(N, n_{features} \cdot n_{channels})$.

A continuación se incluye el código utilizado para la generación de características.

Código 5: Creación de un arreglo de características a partir de una lista de ventanas.

```
def window_to_features(windows_list, features):
  data = []
  for window in windows_list:
    temp = window.agg(features, axis=0).values.flatten('F') # arreglo 1d de largo
     ↪ n_channels*n_features
    data.append(temp)
  return np.array(data)
```

Como dato adicional, posteriormente las características se normalizan con un *MinMaxScaler* ajustado al conjunto de entrenamiento.

2.3. Metodología de los entrenamientos preliminares

Para los entrenamientos preliminares se utilizan clasificadores SVM lineal, K-NN y Random Forest. Estos clasificadores se entrenan en una búsqueda por grilla para encontrar los mejores hiper-parámetros, una vez encontrada la mejor versión de cada clasificador se calculan los accuracy de entrenamiento y validación para cada clasificador y se toman decisiones de acuerdo a estos.

Como último detalle, se incluyen los hiper-parámetros utilizados para los entrenamientos preliminares.

SVM Lineal:

- C : 0.0001, 0.001, 0.1, 1, 10

K-NN:

- $N_{neighbors}$: 5, 10, 15
- Algorithm: ball tree, kd tree

Random Forest:

- $N_{estimators}$: 150, 250

2.4. Elección de características

Lo ideal es tener suficientes características para resumir bien la información de cada señal, sin embargo puede ser perjudicial tener muchas características, pues puede significar mayores tiempos de entrenamiento y overfitting. Es por ello que se decide limitar la cantidad a un rango de entre 4 a 6 características por canal.
Dado que existen muchas medidas estadísticas utilizadas en el análisis de señales, se establecen las siguientes estadísticas mínimas:

- Waveform Length: Es el largo acumulado de la forma de la señal sobre el tiempo. Se calcula de la siguiente forma:

$$\sum_{n=1}^{N-1} |x_{n+1} - x_n|$$

- Mean Absolute Deviation: Se utiliza para detectar los niveles de contracción muscular.
- RMS: Representa la potencia promedio de la señal en unidades de amplitud (Volts generalmente).
- Mean Crossing Rate: La cantidad de veces que la amplitud de la señal cruza el valor promedio de amplitud.

Dadas estas características, se evalúa añadir las siguientes:

- Rango: Permite conocer el rango máximo de amplitud de la señal.
- Skew: Corresponde al grado de asimetría observado en una distribución de probabilidad que se desvía de una distribución normal simétrica.
- Kurtosis: Corresponde al nivel de outliers presentes en una distribución de datos.

A continuación se presentan los resultados de clasificación al añadir cada característica por separado.

Tabla 1: Resultados de accuracy en entrenamiento y validación al añadir características. Los resultados se presentan en la forma acc_{train}, acc_{val}

Clasificadores	Base	+Rango	+Skew	+Kurtosis
SVM Lineal	0.841, 0.926	0.858, 0.950	0.850, 0.949	0.844, 0.929
Random Forest	1.0, 1.0	1.0, 1.0	1.0, 1.0	1.0, 1.0
K-NN	0.966, 0.980	0.977, 0.987	0.954, 0.967	0.967, 0.982

Los resultados muestran que añadir rango y kurtosis por separado mejoran el rendimiento de los clasificadores SVM y K-NN en ambos conjuntos, por lo que ambos son candidatos a ser añadidos como características. Por último se evalúa el rendimiento añadiendo ambos a la vez.

Tabla 2: Resultados de accuracy en entrenamientoy validación al añadir rango y kurtosis como características.

Clasificadores	Base	+ Rango + Kurtosis
SVM Lineal	0.841, 0.926	0.861, 0.953
Random Forest	1.0, 1.0	1.0, 1.0
K-NN	0.966, 0.980	0.977, 0.989

Dado que añadir ambas características mejora el resultado de añadirlas por separado, se añadirán ambas características para futuras iteraciones.

2.5. Selección de canales

Se evalúa la posibilidad de eliminar un canal, esto ya que puede que no todos los canales entreguen información relevante acerca de los gestos. Eliminar un canal redundante puede resultar beneficioso, ya que el número de características se reduce significativamente, lo que puede proveer mejores tiempos de entrenamiento, menor riesgo de overfitting por número elevado de características y mejor capacidad de generalización.

Para encontrar posibles candidatos a eliminación se utilizan 2 métricas:

- **Correlación absoluta**: Mide que tanto se relacionan 2 variables, tomando valores entre 0 y 1. En este caso correlaciones altas entre canales pueden indicar redundancia, dado que la información entregada por un canal sería muy similar a la que puede entregar otro canal. Hay varios tipos de correlaciones, en este caso se utiliza la correlación de Pearson que mide relaciones lineales, así como las correlaciones de Spearman y Kendall que miden relaciones monotónicas.
- **Información mutua**: Mide la relación de dependencia entre 2 variables, específicamente que tanto se reduce la incertidumbre en el valor de una variable si es que se conoce el valor de la otra. En este caso se utiliza para determinar cuales son los canales menos relevantes al momento de determinar un gesto, osea, aquellos con menor información mutua.

Se calculan las correlaciones entre canales, así como la información mutua entre canales y labels. Este calculo se realiza con el dataset de validación, dejando de lado datos asociados a los labels 0 y 7, ya que no son de interés para este proyecto. Hay que mencionar que si bien estos resultados pueden variar de acuerdo a los sujetos que se usen para validación, los resultados presentados deberían ser representativos para la mayoría de subconjuntos.

Tabla 3: Pares de canales con correlaciones más altas

Canales	Pearson	Kendall	Spearman
Ch 3 - Ch2	0.406	0.390	0.531
Ch 4 - Ch3	0.429	0.342	0.448
Ch 2 - Ch 1	0.439	0.329	0.439

Tabla 4: Canales con información mutua más baja

Canal	Información Mutua
Canal 2	0.0735
Canal 3	0.1223
Canal 6	0.1253

De las correlaciones se puede extraer que los principales candidatos a ser eliminados son los canales 2 y 3, teniendo estos 2 de las correlaciones más altas entre sí, además de tener correlaciones altas (relativas al resto de canales) con otros canales. Aún así estas correlaciones suelen estar en el rango 0.4-0.5, que es a lo más una correlación media-baja.
Por otro lado la métrica de información mutua reafirma a los canales 2 y 3 como aquellos que entregan información menos relevante.

Dado que las métricas reiteran a los canales 2 y 3 como los más redundantes (en comparación a los demás canales), se decide comprobar como cambian los resultados de clasificación si se elimina alguno de estos canales.

Tabla 5: Comparación de accuracy al eliminar algún canal. Los resultados son de la forma acc_{train}, acc_{val}

Clasificadores	Sin eliminar canal	Sin canal 2	Sin canal 3
SVM Lineal	0.861, 0.953	0.842, 0.940	0.858, 0.950
Random Forest	1.0, 1.0	1.0, 1.0	1.0, 1.0
K-NN	0.977, 0.989	0.973, 0.984	0.974, 0.988

De los resultados se hace visible que eliminar los canales tiene un impacto negativo en el accuracy, sin embargo eliminar el canal 3 tiene un impacto menos notable en el accuracy, por lo que se podría decir que este es menos relevante que el canal 2. Ante estos resultados se consideran 2 opciones válidas: No eliminar ningún canal, o eliminar el canal 3. Si bien no eliminar ningún canal no tendría efectos negativos en el accuracy, eliminar el canal 3 reduciría la cantidad de características, lo que podría ayudar a mejorar los tiempos de entrenamiento y prevenir overfitting.

En este caso se opta por eliminar el canal 3 para futuras iteraciones.

2.6. Filtrado de señales

Adicional a la eliminación de canales, también se le pueden aplicar filtros a las señales emg, ya sea para eliminar ruido o resaltar un cierto rango de frecuencias de interés. Dado que por transformada de fourier se sabe que las frecuencias van desde 0 hasta al menos 400 hz, se aplican filtros pasaaltos y pasabajos en estos rangos de frecuencia y se evalúa el efecto que tienen en la clasificación.

Tabla 6: Comparación de accuracy para distintos filtros. Los resultados están en la forma acc_{train}, acc_{test}

Clasificadores	Sin filtro	Pasaaltos		Pasabajos	
		1 Hz	12 Hz	400 Hz	350 Hz
SVM Lineal	0.858, 0.950	0.857, 0.951	0.856, 0.936	0.848, 0.945	0.848, 0.946
Random Forest	1.0, 1.0	1.0, 1.0	1.0, 1.0	1.0, 1.0	1.0, 1.0
K-NN	0.974, 0.988	0.973, 0.991	0.964, 0.981	0.962, 0.979	0.961, 0.980

De la tabla anterior se puede apreciar que la tendencia es que el accuracy empeore al aplicar un filtro. Dados estos resultados no se puede afirmar que aplicar un filtro tenga algún efecto beneficioso, por lo que se opta por no aplicar ningún filtro para futuras iteraciones.

2.7. Tamaño y separación de ventanas

Actualmente las ventanas utilizadas en entrenamiento son de tamaño 800, con una separación de 250 muestras entre ventanas creadas de forma consecutiva. Se evalúa cambiar el tamaño y separación actual que poseen las ventanas: ventanas más grandes implicarían mayor información para cada ventana y una menor cantidad de ventanas; por otro lado una mayor separación implicaría una menor intersección entre datos de ventanas y reduciría la cantidad de estas. Para determinar si se aplicará algún cambio a la creación de ventanas se realiza un entrenamiento preliminar, alterando una variable a la vez. A continuación se muestran los resultados:

Tabla 7: Comparación de accuracy para distintas configuraciones de ventanas. Los resultados están en la forma acc_{train}, acc_{test}

Clasificadores	Estándar	Cambio en tamaño		Cambio en separación	
		750	950	150	350
SVM Lineal	0.858, 0.950	0.858, 0.951	0.860, 0.945	0.868, 0.951	0.847, 0.937
Random Forest	1.0, 1.0	1.0, 1.0	1.0, 1.0	1.0, 1.0	1.0, 1.0
K-NN	0.974, 0.988	0.972, 0.989	0.993, 0.996	0.990, 0.996	0.943, 0.970

Ante estos resultados es de notar que el cambio en la separación de ventanas afecta de forma consistente a los clasificadores (exceptuando Random Forest), en donde una menor separación mejora el rendimiento de los clasificadores, mientras que sucede lo contrario cuando aumenta. Por otro lado cambiar el tamaño de ventana sólo afecta de forma significativa al clasificador K-NN, en donde un mayor tamaño de ventana mejora significativamente los resultados tanto en el conjunto de entrenamiento como el de validación.

Dado que tanto aumentar el tamaño de las ventanas como disminuir la separación mejoran los resultados de accuracy en ambos conjuntos, se evalúa aplicar ambos métodos a la vez. A continuación se muestran los resultados de ampliar el tamaño de las ventanas y disminuir su separación a la vez:

Tabla 8: Accuracies obtenidos con ventanas de 950 y separación de 150.

SVM Lineal	0.868, 0.950
Random Forest	1.0
K-NN	0.990, 0.996

De la tabla anterior se puede notar que los resultados no son muy distintos a comparación de simplemente reducir la separación a 150 muestras. Dado lo anterior, para futuras iteraciones solamente se reducirá la separación entre ventanas, dado que de esta forma se logran mejoras consistentes tanto en SVM Lineal como en K-NN.

3. Entrenamiento

En esta sección se mencionan los clasificadores a utilizar y su método de entrenamiento. Además se presentan los resultados obtenidos en el conjunto de validación y en kaggle.

3.1. Clasificadores a utilizar

El entrenamiento sigue el modelo de los entrenamientos preliminares, realizando una búsqueda por grilla con *balanced accuracy* como métrica de validación. A diferencia de los entrenamientos preliminares, la cantidad de clasificadores e Hiperparámetros a utilizar es mayor. A continuación se enlistan los clasificadores y sus respectivos hiper-parámetros:

SVM Lineal:

- C : 0.0001, 0.001, 0.1, 1, 10

SVM Polinomial:

- C : 0.01, 0.1, 1
- γ: 0.1, 1, 10
- grado: 2, 3

SVM RBF:

- C : 0.01, 0.1, 1, 10
- γ: 0.1, 1, 10

K-NN

- $N_{neighbors}$: 5, 10, 15, 20
- Algorithm: ball tree, kd tree

AdaBoost

- $N_{estimators}$: 10, 30, 50, 70, 90, 110

Random Forest

- $N_{estimators}$: 150, 250, 350

Aparte de la búsqueda por grilla se entrena una red neuronal con 3 capas ocultas de 30 neuronas, función de activación relu, learning rate adaptativo y early stopping, el resto de características de la red son las que tiene por default un *MLPClassifier* de *Sklearn*.

3.2. Reducción de características

Para un entrenamiento eficaz se busca tener la mayor información posible reduciendo al mínimo necesario la cantidad de características. Para ello se realiza primero un entrenamiento preliminar solamente con SVM Lineal, AdaBoost y RandomForest,

dado que esos clasificadores poseen parámetros que permiten determinar las características más importantes. Una vez completado el entrenamiento preliminar se aplica el clasificador con mayor accuracy en la función *SelectFromModel* de sklearn, lo que posteriormente permite reducir el total de características a utilizar en el entrenamiento final.

3.3. Resultados de entrenamiento

Se realiza el entrenamiento preliminar, obteniendo los siguientes resultados en el conjunto de validación:

Tabla 9: Accuracy de validación para el entrenamiento previo.

Forest	SVM Lineal	AdaBoost
1.0	0.949	0.343

En este caso Random Forest obtiene el accuracy más alto, por lo que se utiliza para la selección de características. La aplicación de la función SelectFromModel reduce la cantidad de características de 42 a 18.

Se entrenan todos los clasificadores sobre el conjunto reducido, obteniendo los siguientes resultados:

Tabla 10: Accuracy de clasificadores para los conjuntos de entrenamiento y validación.

Clasificador	Acc. Entrenamiento	Acc. Validación
Forest	1.0	1.0
Red Neuronal	0.913	0.965
K-NN	0.932	0.962
SVM Polinomial	0.885	0.947
SVM RBF	0.833	0.932
SVM Lineal	0.846	0.961
AdaBoost	0.342	0.389

De los resultados el clasificador con mejores resultados es Random Forest (350 estimadores), seguido de la red neuronal y K-NN (20 neighbors, algoritmo ball tree). Un caso destacable es el de AdaBoost, el cual tiene un rendimiento mucho peor comparado al resto de clasificadores, probablemente debido a una búsqueda en grilla poco extensa, o a que simplemente el clasificador no es óptimo para este tipo de problema. Es de notar que todos los clasificadores tienen mejor accuracy en el conjunto de validación que en el de entrenamiento, es muy probable que esto se deba al uso de la búsqueda en grilla (early stopping en el caso de la red neuronal).

A continuación se muestran las matrices de confusión para el conjunto de validación de los 3 mejores clasificadores.

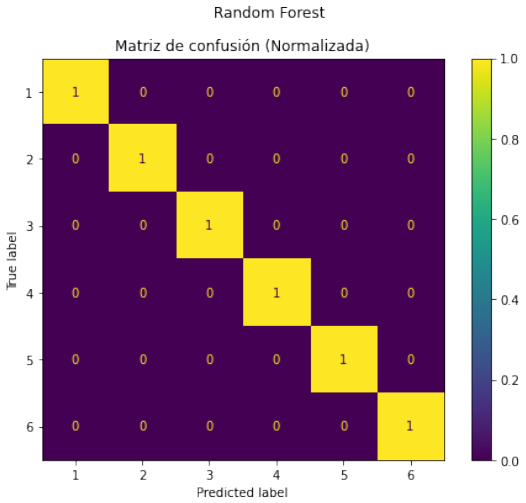

Figura 7: Matriz de confusión para Random Forest

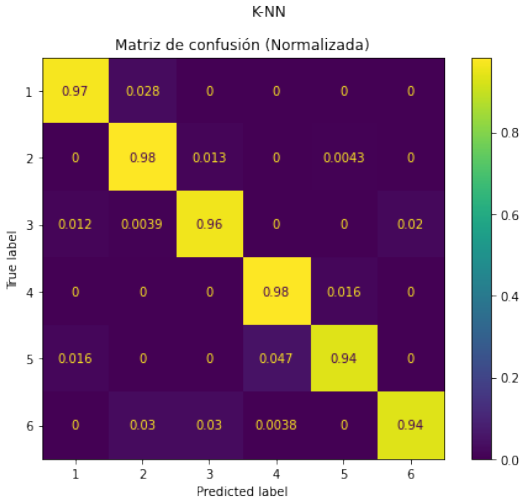

Figura 8: Matriz de confusión para K-NN

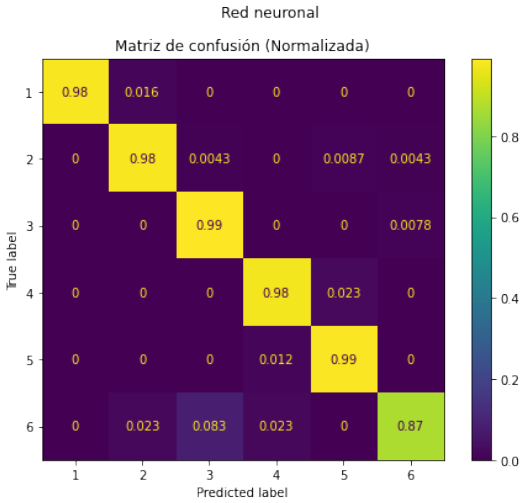

Figura 9: Matriz de confusión para red neuronal

En cuanto a las matrices de confusión para el conjunto de validación los 3 clasificadores obtuvieron buenos resultados en la mayoría de clases, teniendo algunos problemas la red neuronal con el gesto 6. Destaca el clasificador Random Forest por no poseer errores, lo cuál es un indicio de overfitting.

4. Resultados de Kaggle y análisis

En un principio se optaría por utilizar el mejor clasificador para la competencia de kaggle, pero dado que los resultados de Random Forest son inusualmente altos, se opta por utilizar los 3 mejores clasificadores. A continuación se muestran los resultados obtenidos:

Tabla 11: Accuracy de clasificadores en el conjunto de prueba.

Forest	K-NN	Red Neuronal
0.811	0.818	0.846

De los resultados es notable que el mejor de los 3 clasificadores probados en el conjunto de prueba es la red neuronal, obteniendo 0.846 en el leaderbord publico. Por otro lado K-NN y Random Forest tienen rendimientos más cercanos, siendo mejor K-NN por una diferencia de 0.007. Es de notar que a pesar de que los clasificadores lograron resultados superiores a 90 % tanto en los conjuntos de entrenamiento como de validación, ninguno logra llegar al 85 %, estando 2 de los 3 clasificadores en el rango 81-82 %.

Algo que importante de notar, es que a pesar de que Random Forest logró resultados perfectos en todas las pruebas previas al conjunto de prueba, es el clasificador con peor rendimiento de los 3 probados. Es muy probable que esto se deba a overfitting para los conjuntos de prueba y validación, posiblemente debido a un gran número de estimadores.

4.1. Problemas y posibles mejoras

Como se hizo notar en la sección anterior, el rendimiento en el conjunto de prueba deja que desear si se compara con el rendimiento en otros conjuntos. Existe la posibilidad de overfitting en el conjunto de validación debido a su uso para mejorar los hiper-parámetros. Ante esto se podría intentar utilizar un score distinto al momento de optimizar los hiper-parámetros, ya que en los entrenamientos se utilizó el accuracy balanceado, el cuál podría no ser la métrica más confiable, algunos scores que podrían utilizarse en su lugar serían el F1 y ROC AUC. También se podría probar una proporción distinta en el split de sujetos de entrenamiento y validación.

Otra posible mejora radica en las características utilizadas, pues si bien en este caso la cantidad no se consideró excesiva, si podrían existir otras medidas estadísticas que entreguen mejor información sobre el dataset. Lo anterior también aplica al preprocesado de las señales, ya que no se consideraron otros métodos para reducir el ruido de las señales EMG además del uso de filtros pasabajos/pasaaltos.

Conclusiones

En general el entrenamiento de clasificadores en base a datos de señales no es un problema trivial, y para ser realizado de forma efectiva es deseable poseer conocimientos previos en el área, esto pues para sacar el máximo provecho de las señales se requiere de un pre-procesado efectivo. Para este proyecto se consideró el accuracy de los clasificadores entrenados como métrica de referencia para elegir el tratado que darle a los datos, sin embargo este no siempre podría ser un buen criterio. En general se confió mucho en el accuracy como métrica de referencia, por lo que es importante considerar una mayor variedad de métricas y métodos de evaluación en futuros proyectos, esto tanto para el procesado de datos como para el refinamiento de clasificadores, para así poder tomar decisiones mejor fundadas al momento de intentar mejorar los resultados.

Si bien los resultados logrados dejan que desear, el trabajo realizado podría servir como una base para refinar y lograr mejores resultados.

Anexo

Código 6: utils.py

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import time
import sklearn.metrics as metrics
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import PredefinedSplit
from sklearn.feature_selection import SelectFromModel
from sklearn import svm
import seaborn as sns
from scipy.signal import butter,filtfilt
from scipy.fft import fft, fftfreq

def plot_freq(df, channel='channel3'):
    # Number of samples in normalized_tone
    N = len(df[channel]) #SAMPLE_RATE * DURATION

    sample_rate = N/(df['time'].max()/1000)
    yf = fft(np.array(df[channel]))
    xf = fftfreq(N, 1 / sample_rate)

    i_max = np.argmax(yf)
    x_max = xf[i_max]
    print(x_max)
    plt.figure(figsize=(16,9))
    plt.plot(xf, np.abs(yf))
    plt.ylabel('Amplitud')
    plt.xlabel('Frecuencia (Hz)')
    plt.show()

def plot_channel(dataframe, channel='channel1'):
  fig, axis = plt.subplots(2, figsize = (12,12))
  plt.title(channel)
  axis[0].scatter(dataframe['time'], dataframe[channel], s=20, c=dataframe['class'],
     ↪ cmap='rainbow')
  axis[1].plot(dataframe['time'], dataframe[channel], color='black')
  plt.xlabel('t[ms]')
  plt.show()

def read_data(path):
    '''
    path: Carpeta en donde se encuentran todas las carpetas con la data de los sujetos
     ↪ .
    '''
    d_read = lambda d : pd.read_csv(d, sep='\t', header = 0)
```

```
    path_train = path +'/subj{}'
    common_names = ['1.txt', '2.txt']

    # Juntar todos los datos
    #df_data = pd.DataFrame()
    data_list = []
    for i in range(1, 31):
        sub_n = str(i).zfill(2) # Número del sujeto
        folder = path_train.format(sub_n) # Ruta de la carpeta
        for file in common_names:
            df_file = d_read(f'{folder}/{file}')
            df_file['subject'] = i
            df_file['capture'] = int(file[0])

            data_list.append(df_file)
            #df_data = pd.concat([df_data, df_file], ignore_index=True)
    return data_list

def read_test_data(path_test):

    df_test = pd.read_csv(path_test, sep=",", header=0)

    # Almacenar ventanas 800x8 en un arreglo de numpy
    test = np.array(df_test.drop('Id', axis=1)).reshape(672, 8, 800).transpose((0, 2, 1)
     ↪ )

    return test

def butter_filter(data, cutoff, fs, order, btype = 'low'):
    nyq = 0.5 * fs
    normal_cutoff = np.array(cutoff) / nyq
    # Get the filter coefficients
    b, a = butter(order, normal_cutoff, btype=btype, analog=False)
    y = filtfilt(b, a, data)
    return y

def substract_mean(dataframe, columns):
  df_copy = dataframe.copy(True)
  for x in columns:
    df_copy[x] = dataframe[x] - dataframe[x].mean()
  return df_copy


def filtrar_df(df, fs=970, cutoffs = 350, btype='low', channels = [f'channel{i}' for i in
    ↪  range(1,9)]):
  df_copy = df.copy(True)
  for ch in channels:
        y = butter_filter(df_copy[ch], cutoffs, fs, order = 2, btype=btype)
        df_copy[ch] = y
  return df_copy


def create_windows(dataframe, window_size = 800, step = 250):
```

```
    '''
    Crea ventanas de tamaño (window_size, n_columns) a partir de un dataframe.
    El número de ventanas depende del largo del dataframe y el step utilizado.
    Las ventanas son dataframes, y se retorna una lista con estos.

    Inputs:
    - dataframe: Dataframe de pandas
    '''
    windows = []
    start = 0
    while start + window_size < len(dataframe):
        window = dataframe[start:start+800]
        windows.append(window)
        start += step
    return windows

def emg_windows(df_list, window_size = 800, step = 250):
    classes = np.arange(1, 7)
    windows = []
    labels = []
    for df in df_list:
      for cl in classes:
        temp_window = create_windows(df[df['class']==cl].drop('class', axis=1),
        ↪ window_size, step)
        windows+=temp_window
        labels += [cl]*len(temp_window)
    return windows, labels

def train_val_split(df_list, train_sub, val_sub):
    train_list = []
    val_list = []
    for df in df_list:
      if df['subject'].unique() in train_sub:
        train_list.append(df)
      elif df['subject'].unique() in val_sub:
        val_list.append(df)
    return train_list, val_list

def window_to_features(windows_list, features):
    data = []
    for window in windows_list:
      temp = window.agg(features, axis=0).values.flatten('F') # arreglo 1d de largo
      ↪ n_channels*n_features
      data.append(temp)
    return np.array(data)

def multitrain(grid_dict, x_cv, y_cv):
    '''
    grid_list : Diccionario con grillas listas para el entrenamiento
    x_cv: data de entrenamiento y validación concatenada
    y_cv: labels de entrenamiento y validación concatenados
    '''
    trained_dict = {}
```

```
149     for grid in grid_dict:
150         grid_dict[grid].fit(x_cv, y_cv)
151         grid_obj = grid_dict[grid]
152         trained_dict[grid] = [grid_obj.best_estimator_, grid_obj.best_params_]
153 
154     return trained_dict
155 
156 
157 def show_matrixes(y_true, y_predicted, model=''):
158 
159   acc = metrics.accuracy_score(y_true, y_predicted)
160   print(f'Accuracy = {round(acc, 3)}')
161   fig, axs = plt.subplots(1, 1)
162   fig.set_size_inches(8, 6)
163   axs.set_title('Matriz de confusión (Normalizada)')
164   metrics.ConfusionMatrixDisplay.from_predictions(y_true, y_predicted, normalize='
        ↪ true', ax=axs)
165 
166   if model!='':
167     fig.suptitle(model)
168   fig.show()
169   return acc
170 
171 
172 class TrainWrapper():
173     '''
174     Clase que reune métodos para el procesamiento de datos para su uso en el
        ↪ entrenamiento.
175     '''
176     def __init__(self, df_list):
177         '''
178         Constructor: Recibe una lista de dataframes procesados para ser utilizados en el
        ↪  proyecto.
179         '''
180         self.dfs = df_list
181         self.df_train = None
182         self.df_val = None
183 
184         self.train_windows = None
185         self.val_windows = None
186 
187         self.y_train = None
188         self.y_val = None
189 
190         self.x_train = None
191         self.x_val = None
192         self.scaler = None
193 
194         self.x_tv = None
195         self.y_tv = None
196         self.cv = None
197 
198     def split(self, train_sub, val_sub):
```

```
        '''
        Recibe listas con los sujetos de entrenamiento y validación.
        Crea los splits de entrenamiento y validación en la lista inicial de dataframes
        '''
        self.df_train, self.df_val = train_val_split(self.dfs, train_sub, val_sub)

    def make_windows(self, window_size = 800, step = 250):
        '''
        Se crean las ventanas de entrenamiento y validación a partir de las listas respectivas.
        '''
        if self.df_train!=None and self.df_val!=None:
            self.train_windows, self.y_train = emg_windows(self.df_train, window_size, step)
            self.val_windows, self.y_val = emg_windows(self.df_val, window_size, step)

        else:
            print('Primero debes crear un split de entrenamiento/validación (método split)')

    def compute_features(self, feature_list, scaler = MinMaxScaler()):
        '''
        Convierte las ventanas en data apta para entrenamiento a partir de
        la lista de caracteristicas a computar.

        feature_list: Lista de caracteristicas (debe ser apta para un pandas.agg)
        scaler: El scaler a usar en los datos
        '''
        if self.train_windows!=None and self.val_windows!=None:
            to_drop = ['subject', 'capture', 'time']
            self.x_train = [x.drop(to_drop, axis=1) for x in self.train_windows]
            self.x_val = [x.drop(to_drop, axis=1) for x in self.val_windows]

            self.x_train = window_to_features(self.x_train, feature_list)
            self.x_val = window_to_features(self.x_val, feature_list)

            #scal = scaler()
            scaler.fit(self.x_train)
            self.scaler = scaler
            self.x_train = scaler.transform(self.x_train)
            self.x_val = scaler.transform(self.x_val)
        else:
            print('Primero debes crear ventanas de entrenamiento/validación (método make_windows)')

    def make_test_folds(self):
        self.x_tv = np.concatenate([self.x_train, self.x_val])
        self.y_tv = np.concatenate([self.y_train, self.y_val])
        test_fold = np.concatenate([
                                np.full(self.x_train.shape[0], -1),
                                np.zeros(self.x_val.shape[0])])
        self.cv = PredefinedSplit(test_fold)
```

Código 7: Contenido de Testing_IC.ipynb

```
# -*- coding: utf-8 -*-
"""Testing_IC.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1nT5et2VHLxUpk-aZja44VsChttCebTEa
"""

import utils
import pandas as pd

df_list = utils.read_data('train')

"""## Split entrenamiento validación"""

from sklearn.model_selection import train_test_split
import numpy as np

train_sub, val_sub = train_test_split(np.arange(1, 31), train_size = 0.8,
    ↪ random_state = 42)

"""## Gráficos"""

subj = df_list[0]
subj.head()

"""### Amplitud vs Tiempo"""

utils.plot_channel(subj, 'channel1')

utils.plot_channel(subj, 'channel3')

utils.plot_channel(subj, 'channel5')

"""### Amplitud vs Frecuencia"""

utils.plot_freq(subj, 'channel1')

utils.plot_freq(subj, 'channel3')

utils.plot_freq(subj, 'channel5')

"""## Correlaciones"""

val_list = []
for df in df_list:
    if df['subject'].unique() in val_sub:
      val_list.append(df)

df_val = pd.concat(val_list, ignore_index=True)
```

```
51
52 non_features = ['class', 'subject', 'capture', 'time']
53 short_corr = lambda d,m: d.drop(non_features, axis=1).corr(method=m).abs()
54 corr_methods = ['pearson', 'kendall', 'spearman']
55 reduced_data  = df_val[(df_val['class']!=0) & (df_val['class']!=1)] # Se quitan las
       ↪ clases que no interesan
56
57 reduced_corr = {}
58 for method in corr_methods:
59   reduced_corr[method] = short_corr(reduced_data, method)
60
61 for key in reduced_corr:
62   top = reduced_corr[key].where(np.tril(np.ones(reduced_corr[key].shape), -1).astype
       ↪ (bool)).stack() # Se toma la triangular para evitar repetir pares
63   top = top.sort_values(ascending = False)
64   print(f'Canales más correlacionados según: {key}')
65   print(top[:5])
66
67 from sklearn.feature_selection import mutual_info_regression
68 mi = mutual_info_regression((reduced_data).drop(non_features, axis=1),
       ↪ reduced_data['class'])
69
70 print('Mutual information')
71 for i in range(len(mi)):
72   print(f'Canal{i+1} : {round(mi[i], 4)}')
73
74 """## Features"""
75
76 def data_range(x):
77   return x.max()-x.min()
78
79 def rms(x):
80   z = x*x
81   sum = z.sum()
82   result = np.sqrt(sum/len(x))
83
84   return result
85
86 # Zero crossing rate
87 def zcr(x):
88   x = np.array(x)
89   n = len(x)
90   zc = ((x[:-1] * x[1:]) < 0).sum()
91   return zc/n
92
93 def mcr(x):
94   x = np.array(x)
95   z = x-np.mean(x)
96   return zcr(z)
97
98 #waveform length
99 def wl(data):
100    return np.sum(np.abs(np.diff(data,axis=0)), axis=0)
```

```
101
102 features = [wl, 'mad', mcr, rms]
103
104 """## Grids"""
105
106 from sklearn.ensemble import RandomForestClassifier
107 from sklearn.neighbors import KNeighborsClassifier
108
109 # pre_grids
110 param_pregrid_linear = [{
111                  'C':[0.0001, 0.001, 0.1, 1, 10],
112                  'kernel': ['linear']
113                }]
114
115 param_pregrid_forest = [{
116     'n_estimators':[150, 250],
117     'max_depth': [None],
118     'criterion' : ['entropy']
119                }]
120
121 param_pregrid_knn = [{
122     'n_neighbors':[5, 10, 15],
123     'algorithm': ['ball_tree', 'kd_tree']
124 }]
125
126 from sklearn import svm
127 from sklearn.model_selection import GridSearchCV
128
129 common_pregrid = {'scoring':'balanced_accuracy', 'refit':True, 'verbose':1}
130
131 pregrid_dict = {
132     'linear':lambda x :  GridSearchCV(estimator= svm.SVC(), param_grid =
      ↪ param_pregrid_linear,
133                                 cv = x, **common_pregrid),
134
135     'forest': lambda x : GridSearchCV(estimator= RandomForestClassifier(),
136                 param_grid = param_pregrid_forest, cv = x , **common_pregrid),
137
138     'knn' : lambda x : GridSearchCV(estimator = KNeighborsClassifier(),
139                     param_grid = param_pregrid_knn, cv = x, **common_pregrid)
140 }
141
142 """## Simplificación de entrenamiento"""
143
144 def train_with_wrapper(df_list, train_sub, val_sub, features, grid_dict):
145   wrapper = utils.TrainWrapper(df_list)
146   wrapper.split(train_sub, val_sub) # Split train/val
147   wrapper.make_windows() # Crear ventanas
148   wrapper.compute_features(features)
149   wrapper.make_test_folds() # Crear x_tv, y_tv y cv
150
151   train_dict = {}
152   for key in grid_dict:
```

```
    train_dict[key] = grid_dict[key](wrapper.cv) # Setea cross-validation fold

  wrapper_trained = utils.multitrain(train_dict, wrapper.x_tv, wrapper.y_tv)

  return wrapper_trained, wrapper

preparams = [train_sub, val_sub, features, pregrid_dict]

"""## Prueba 1 """

channels = [f'channel{i}' for i in range(1,9)]

df_list1 = [utils.substract_mean(x, channels) for x in df_list]

test1_trained, test1_wrapper = train_with_wrapper(df_list1, *preparams)

for x in test1_trained:
  print(f'{x}: {test1_trained[x][1]}')

import sklearn.metrics as metrics
pre_accuracies = {}
for x in test1_trained:
  cl = test1_trained[x][0]

  pre_predictv = cl.predict(test1_wrapper.x_val)
  pre_predict = cl.predict(test1_wrapper.x_train)
  pre_accuracies[x] = (metrics.accuracy_score(test1_wrapper.y_train, pre_predict),
                       metrics.accuracy_score(test1_wrapper.y_val, pre_predictv))

pre_accuracies

"""## Prueba 1.5 (Elegir caracteristicas extra)"""

extra_features = ['kurtosis', 'skew', data_range]

pre_accsf = {}
for f in extra_features:
  nparams = [train_sub, val_sub, features + [f], pregrid_dict]
  testf_trained, testf = train_with_wrapper(df_list1, *nparams)

  pre_accsf[f] = {}
  for x in testf_trained:
    cl = testf_trained[x][0]

    pre_predictv = cl.predict(testf.x_val)
    pre_predict = cl.predict(testf.x_train)
    pre_accsf[f][x] = (metrics.accuracy_score(testf.y_train, pre_predict),
                       metrics.accuracy_score(testf.y_val, pre_predictv))

pre_accuracies

for key in pre_accsf:
  print(key)
```

```
  print(pre_accsf[key])

nparams = [train_sub, val_sub, features + [data_range, 'kurtosis'], pregrid_dict]
testf_trained, testf = train_with_wrapper(df_list1, *nparams)

pre_accsf['skew + range'] = {}

for x in testf_trained:
    cl = testf_trained[x][0]

    pre_predictv = cl.predict(testf.x_val)
    pre_predict = cl.predict(testf.x_train)
    pre_accsf['skew + range'][x] = (metrics.accuracy_score(testf.y_train, pre_predict)
    ↪ ,
                    metrics.accuracy_score(testf.y_val, pre_predictv))

for key in pre_accsf:
  print(key)
  print(pre_accsf[key])

"""## Caracteristicas definitivas"""

features += [data_range, 'kurtosis']
preparams = [train_sub, val_sub, features, pregrid_dict]

"""## Prueba 2 (Sin canal 2 o 3 + P1)"""

pre_accs2 = {'channel2':{}, 'channel3':{}}
for ch in ['channel2', 'channel3']:
    df_list2 = [x.drop([ch], axis=1) for x in df_list1]
    test2_trained, test2_wrapper = train_with_wrapper(df_list2, *preparams)
    for x in test2_trained:
      cl = test2_trained[x][0]

      pre_predictv = cl.predict(test2_wrapper.x_val)
      pre_predict = cl.predict(test2_wrapper.x_train)

      pre_accs2[ch][x] = (metrics.accuracy_score(test2_wrapper.y_train, pre_predict)
      ↪ ,
                      metrics.accuracy_score(test2_wrapper.y_val, pre_predictv))

for key in pre_accs2:
  print(key)
  print(pre_accs2[key])

"""## Prueba 3 (Filtros + P2)"""

channels_p3 = [x for x in channels if x!='channel3']

df_list3 = [x.drop(['channel3'], axis=1) for x in df_list1]

filters = [ {'cutoffs' : 350, 'btype':'low'}, {'cutoffs' : 400, 'btype':'low'},
          {'cutoffs' : 1, 'btype':'high'}, {'cutoffs' : 12, 'btype':'high'}]
```

```
257
258 pre_accs3 = {}
259 for f in filters:
260   f['channels'] = channels_p3
261   df_listf = [utils.filtrar_df(df_list3[i], **f) for i in range(len(df_list3))]
262   test3_trained, test3_wrapper = train_with_wrapper(df_listf, *preparams)
263
264   key = f"{f['cutoffs']}-{f['btype']}"
265   pre_accs3[key] = {}
266
267   for x in test3_trained:
268     cl = test3_trained[x][0]
269     pre_predictv = cl.predict(test3_wrapper.x_val)
270     pre_predict = cl.predict(test3_wrapper.x_train)
271
272     pre_accs3[key][x] = (metrics.accuracy_score(test3_wrapper.y_train, pre_predict
      ↪ ),
273                          metrics.accuracy_score(test3_wrapper.y_val, pre_predictv))
274
275 for key in pre_accs3:
276   print(key)
277   print(pre_accs3[key])
278
279 """## Prueba 4 (Ventanas)"""
280
281 win_step = [(800, 250), (800, 150), (800, 350), (750, 250), (950, 250)] # pares (
      ↪ window_size, step)
282
283 df_listv = [x.drop(['channel3'], axis=1) for x in df_list1]
284
285 wrapper = utils.TrainWrapper(df_listv)
286 wrapper.split(train_sub, val_sub)
287
288 ws_results = {}
289 train_dict = {}
290
291 for ws in win_step:
292   wrapper.make_windows(*ws)
293   wrapper.compute_features(features)
294   wrapper.make_test_folds()
295
296   for key in pregrid_dict:
297     train_dict[key] = pregrid_dict[key](wrapper.cv)
298
299   trained = utils.multitrain(train_dict, wrapper.x_tv, wrapper.y_tv)
300   key = str(ws)
301   ws_results[key] = {}
302
303   for x in trained:
304     cl = trained[x][0]
305
306     pre_predictv = cl.predict(wrapper.x_val)
307     pre_predict = cl.predict(wrapper.x_train)
```

```
    ws_results[key][x] = (metrics.accuracy_score(wrapper.y_train, pre_predict),
                          metrics.accuracy_score(wrapper.y_val, pre_predictv))

for key in ws_results:
  print(key)
  print(ws_results[key])

wrapper.make_windows(*(950, 150))
wrapper.compute_features(features)
wrapper.make_test_folds()

for key in pregrid_dict:
     train_dict[key] = pregrid_dict[key](wrapper.cv)

trained = utils.multitrain(train_dict, wrapper.x_tv, wrapper.y_tv)
key = '(950, 150)'
ws_results[key] = {}

for x in trained:
    cl = trained[x][0]

    pre_predictv = cl.predict(wrapper.x_val)
    pre_predict = cl.predict(wrapper.x_train)
    ws_results[key][x] = (metrics.accuracy_score(wrapper.y_train, pre_predict),
                          metrics.accuracy_score(wrapper.y_val, pre_predictv))

for key in ws_results:
  print(key)
  print(ws_results[key])
```

Código 8: Contenido de Train_IC.ipynb

```
# -*- coding: utf-8 -*-
"""Train_IC.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1tIA_SUVcoyLZHrAa8Heup-ie4qRtF0de
"""

"""## Procesado previo"""

import utils
import pandas as pd
from sklearn.model_selection import train_test_split
import numpy as np

df_list = utils.read_data('train')
train_sub, val_sub = train_test_split(np.arange(1, 31), train_size = 0.8,
    ↪ random_state = 42)

```

```
def data_range(x):
  return x.max()-x.min()

def rms(x):
  z = x*x
  sum = z.sum()
  result = np.sqrt(sum/len(x))

  return result

# Zero crossing rate
def zcr(x):
  x = np.array(x)
  n = len(x)
  zc = ((x[:-1] * x[1:]) < 0).sum()
  return zc/n

def mcr(x):
  x = np.array(x)
  z = x-np.mean(x)
  return zcr(z)

#waveform length
def wl(data):
    return np.sum(np.abs(np.diff(data,axis=0)), axis=0)

features = [wl, 'mad', mcr, rms, data_range, 'kurtosis']

# Sustraer promedio
channels = [f'channel{i}' for i in range(1,9)]
df_list1 = [utils.substract_mean(x, channels) for x in df_list]

# Quitar canal 3
df_list1 = [x.drop(['channel3'], axis=1) for x in df_list1]

"""## Simplificación de entrenamiento"""

def train_with_wrapper(df_list, train_sub, val_sub, features, grid_dict):
  wrapper = utils.TrainWrapper(df_list)
  wrapper.split(train_sub, val_sub) # Split train/val
  wrapper.make_windows() # Crear ventanas
  wrapper.compute_features(features)
  wrapper.make_test_folds() # Crear x_tv, y_tv y cv

  for key in grid_dict:
    grid_dict[key].cv = wrapper.cv # Setea cross-validation fold

  wrapper_trained = utils.multitrain(grid_dict, wrapper.x_tv, wrapper.y_tv)

  return wrapper_trained, wrapper

"""### Grids"""

```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

param_grids = {
    'linear' : [{
        'C':[0.001, 0.1, 1, 10],
        'kernel': ['linear']
        }],

     'poly' : [{
          'C': [1e-2, 1e-1, 1],
          'gamma':[1e-1, 1, 1e1],
          'degree':[2, 3],
          'kernel': ['poly']
                }],
    'rbf': [{
          'C': [0.01, 0.1, 1.0, 10.0],
          'gamma': [1e-1, 1, 10],
          'kernel': ['rbf']
                }],
    'knn': [{
        'n_neighbors':[5, 10, 15, 20],
        'algorithm': ['ball_tree', 'kd_tree']}],

    'boost': [{
        'n_estimators': np.arange(10, 111, 20)}],

    'forest': [{
          'n_estimators': np.arange(150, 351, 100),
          'criterion' : ['entropy']
                    }]
}

param_grids.keys()

estimators = {
    'linear': SVC(), 'poly': SVC(), 'rbf': SVC(),
     'knn': KNeighborsClassifier(), 'boost': AdaBoostClassifier(),
      'forest': RandomForestClassifier()
}

"""## Entrenamiento"""

wrapper = utils.TrainWrapper(df_list1)
wrapper.split(train_sub, val_sub) # Split train/val
wrapper.make_windows(step = 150) # Crear ventanas
wrapper.compute_features(features)
wrapper.make_test_folds() # Crear x_tv, y_tv y cv

```

```
126 common = {'scoring':'balanced_accuracy', 'refit':True, 'verbose':1}
127
128 pre_estimators = ['linear', 'boost', 'forest']
129 pre_grid = {}
130 for p in pre_estimators:
131   pre_grid[p] = GridSearchCV(estimator = estimators[p], param_grid =
      ↪ param_grids[p],
132                              cv = wrapper.cv, **common)
133
134 wrapper_trained = utils.multitrain(pre_grid, wrapper.x_tv, wrapper.y_tv)
135
136 wrapper_trained
137
138 import sklearn.metrics as metrics
139
140 pre_accs = {}
141
142 for x in wrapper_trained:
143     cl = wrapper_trained[x][0]
144     pre_predictv = cl.predict(wrapper.x_val)
145     pre_predict = cl.predict(wrapper.x_train)
146
147     pre_accs[x] = (metrics.accuracy_score(wrapper.y_train, pre_predict),
148                    metrics.accuracy_score(wrapper.y_val, pre_predictv))
149
150 pre_accs
151
152 from sklearn.feature_selection import SelectFromModel
153
154 selector = SelectFromModel(wrapper_trained['forest'][0]).fit(wrapper.x_train,
      ↪ wrapper.y_train)
155 x_trainr = selector.transform(wrapper.x_train)
156 x_valr = selector.transform(wrapper.x_val)
157 x_tvr = np.concatenate([x_trainr, x_valr])
158
159 print(f'{wrapper.x_train.shape[-1]}, {x_tvr.shape[-1]}')
160
161 grid = {}
162 for p in estimators.keys():
163     grid[p] = GridSearchCV(estimator = estimators[p], param_grid = param_grids[p],
164                            cv = wrapper.cv, **common)
165
166 trained = utils.multitrain(grid, x_tvr, wrapper.y_tv)
167
168 print('Hiper-parámetros')
169 for key in trained:
170   print(f'{key}: {trained[key][0]}')
171
172 accs = {}
173
174 for x in trained:
175     cl = trained[x][0]
176     predictv = cl.predict(x_valr)
```

```
    predict = cl.predict(x_trainr)

    accs[x] = (metrics.accuracy_score(wrapper.y_train, predict),
                metrics.accuracy_score(wrapper.y_val, predictv))

frac = len(wrapper.x_val)/(len(wrapper.x_train)+ len(wrapper.x_val))
frac

ann = MLPClassifier((30, 30, 30), shuffle = False, random_state = 42,
    ↪ early_stopping=True,
          validation_fraction = frac, learning_rate='adaptive')

ann.fit(x_tvr, wrapper.y_tv)

for i in range(1,7):
  print(f'label={i}')
  print((np.array(wrapper.y_train) == i).sum())
  print((np.array(wrapper.y_val) == i).sum())

predictv = ann.predict(x_valr)
predict = ann.predict(x_trainr)

accs['ann'] = (metrics.accuracy_score(wrapper.y_train, predict),
                metrics.accuracy_score(wrapper.y_val, predictv))

accs

"""## Matrices de confusión"""

c = utils.show_matrixes(wrapper.y_val, trained['forest'][0].predict(x_valr), 'Random
    ↪ Forest')

c =  utils.show_matrixes(wrapper.y_val, trained['knn'][0].predict(x_valr), 'K-NN')

c = utils.show_matrixes(wrapper.y_val, ann.predict(x_valr), 'Red neuronal')

"""## Predicción para conjunto de prueba"""

path_test = 'windows_test.csv'
test_data = utils.read_test_data(path_test)

test_list = [pd.DataFrame(np.delete(test_data[i], 2, axis=1)) for i in range(len(
    ↪ test_data))] # Sin canal 3

x_test = utils.window_to_features(test_list, features)

x_tests = wrapper.scaler.transform(x_test)
x_testr = selector.transform(x_tests)

x_testr.shape

#y_predict = trained['forest'][0].predict(x_testr)
#y_predict = trained['knn'][0].predict(x_testr)
```

```
227 y_predict = ann.predict(x_testr)
228
229 y_predict
230
231 y_kaggle = pd.DataFrame(np.arange(0, 672), columns=['Id'])
232 y_kaggle['Category'] = y_predict
233 y_kaggle.to_csv('predict.csv', index=False)
```