Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Proyecto final

Integrantes: Bastián Garcés
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla Z.
Ayudantes: Juan Pablo Cáceres B.
Rudy García
Rodrigo Salas O.
Sebastian Solanich
Pablo Troncoso P.

Fecha de entrega: 27 Julio de 2021
Santiago, Chile

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Introducción

En este informe se detallará el proyecto final del curso EL4106 Inteligencia Computacional, proyecto que tenía como finalidad hacer que el estudiante se enfrentara a un problema más real, permitiendo así que se apliquen los conocimientos adquiridos durante el transcurso del semestre. El objetivo del proyecto era implementar un clasificador de actividades físicas, donde para ello se utilizará una versión modificada de la base de datos *Human Activity Recognition Using Smartphones Data Set*. La base de datos recién mencionada consistía en 9 señales medidas con una IMU de un samsung Galaxy S II, donde las actividades físicas para las que se calcularon las señales corresponden a: Caminar, Subir escaleras caminando, bajar escaleras caminando, estar sentado, estar parado y estar acostado. Las señales fueron tomadas a 50[Hz] y se obtuvieron 128 muestras. Finalmente el conjunto de datos original fue dividido en un nuevo conjunto que será el que se utilizará para el proyecto, donde dicho conjunto contiene 2800 muestras de entrenamiento, 2000 de validación y 2000 de prueba.

Las bibliotecas que permitirán desarrollar el proyecto son: *Sklearn*, *Numpy*, *Pandas*, *Matplotlib*, *Seaborn*, *Torch*, *Sys*, *Scipy*, *Csv* y *Random*, y el entorno de trabajo será *Colaboratory*.

El presente informe contendrá diversas secciones que irán detallando el código implementado, dichas secciones son:

- Lectura de DataFrames.
- Comparativa de algunas señales, esto permitirá determinar si algunas señales pueden ser o no eliminadas
- Creación de DataFrames a partir de características.
- Normalización de características.
- Clustering
- Random Forest
- Red Neuronal
- Resultados

2. Desarrollo

Como se mencionó durante la introducción, en esta sección se irán mostrando los diferentes códigos implementados, así como también se irá explicando que es lo que hace cada uno.

2.1. Lectura de DataFrames

Para darle lectura a los DataFrames primero se debía crear un código que permitiese subir dichos DataFrames al entorno de trabajo, dicho código se puede observar en el código 1, donde la última línea permite descomprimir el archivo.

Código 1: Código para subir archivos.

```
#Lectura del archivo
from google.colab import files
uploaded = files.upload()

#Descomprimimos el zip
!unzip Dataset_proyecto.zip
```

Luego se tiene otra porción de código que es idéntica a la anterior, pero recibe otros archivos, es en estos archivos donde los clasificadores implementados (Random Forest y Red Neuronal) escribirán la predicción hecha sobre el conjunto de prueba. El código que realiza la actividad anterior se puede observar en el código 2.

Código 2: Código para subir archivos en los que se escribirán los resultados.

```
#Archivos csv donde se guardaran los resultados
from google.colab import files
uploaded = files.upload()

#Descomprimimos el zip
!unzip Resultados.zip
```

Una vez subidos todos los archivos simplemente se les procedió a dar lectura utilizando la función *read_csv()* de *Pandas*, el código 3 es el encargado de realizar lo antes mencionado.

Código 3: Lectura de los distintos DataFrames.

```
#Lectura de los datos de entrenamiento
Entrena1 = pd.read_csv('body_acc_x_entrenamiento.csv')
Entrena2 = pd.read_csv('body_acc_y_entrenamiento.csv')
Entrena3 = pd.read_csv('body_acc_z_entrenamiento.csv')
Entrena4 = pd.read_csv('body_gyro_x_entrenamiento.csv')
Entrena5 = pd.read_csv('body_gyro_y_entrenamiento.csv')
Entrena6 = pd.read_csv('body_gyro_z_entrenamiento.csv')
Entrena7 = pd.read_csv('total_acc_x_entrenamiento.csv')
Entrena8 = pd.read_csv('total_acc_y_entrenamiento.csv')
Entrena9 = pd.read_csv('total_acc_z_entrenamiento.csv')

#Lectura de los datos de validacion
Vali1 = pd.read_csv('body_acc_x_validacion.csv')
```

```
Vali2 = pd.read_csv('body_acc_y_validacion.csv')
Vali3 = pd.read_csv('body_acc_z_validacion.csv')
Vali4 = pd.read_csv('body_gyro_x_validacion.csv')
Vali5 = pd.read_csv('body_gyro_y_validacion.csv')
Vali6 = pd.read_csv('body_gyro_z_validacion.csv')
Vali7 = pd.read_csv('total_acc_x_validacion.csv')
Vali8 = pd.read_csv('total_acc_y_validacion.csv')
Vali9 = pd.read_csv('total_acc_z_validacion.csv')

#Lectura de los datos de prueba
Prueba1 = pd.read_csv('body_acc_x_prueba.csv')
Prueba2 = pd.read_csv('body_acc_y_prueba.csv')
Prueba3 = pd.read_csv('body_acc_z_prueba.csv')
Prueba4 = pd.read_csv('body_gyro_x_prueba.csv')
Prueba5 = pd.read_csv('body_gyro_y_prueba.csv')
Prueba6 = pd.read_csv('body_gyro_z_prueba.csv')
Prueba7 = pd.read_csv('total_acc_x_prueba.csv')
Prueba8 = pd.read_csv('total_acc_y_prueba.csv')
Prueba9 = pd.read_csv('total_acc_z_prueba.csv')


#Labels de los distintos conjuntos
labels_entrena = pd.read_csv('y_entrenamiento.csv')
labels_entrena = labels_entrena['Category']-1

labels_vali = pd.read_csv('y_validacion.csv')
labels_vali = labels_vali['Category']-1
```

Se debe mencionar el hecho de que a los labels de conjunto de entrenamiento y validación se les resto 1, esto se hizo con la finalidad de que los clasificadores implementados trabajasen con datos cuya clasificación vaya de 0 a 5, si no se les hubiese restado 1 a los labels los clasificadores habrían trabajado con etiquetas que van del 1 al 6, lo que podría generar errores, principalmente en la red neuronal.

2.2. Comparativa de señales

Antes de pasar como tal a mostrar el código implementado para comparar señales se debe mencionar el motivo por el cual esto se hizo. Como es sabido se cuentan con 9 señales 3 señales que miden la aceleración total de los ejes X, Y, Z, otras 3 que miden el giroscopio sobre los mismos 3 ejes y finalmente otras 3 señales que fueron obtenidas de aplicar un filtrado pasabajo a las señales de aceleración total en los distintos ejes, es por esto último que se comparan de forma visual las diferentes características, pues existen 3 señales que fueron obtenidas de otra señal, por lo que podría darse el caso de que estas señales obtenidas mediante el uso de un filtro no estén aportando más información, debido a esto durante los próximos parrafos se irán mostrando las diferentes señales sobre una misma muestra, para así lograr ver si estas son muy parecidas entre sí, en cuyo caso se eliminarán las señales obtenidas mediante un filtrado.

El código que permite realizar la comparativa previamente mencionada consiste en graficar señales, para una muestra aleatoria de los diferentes conjuntos. En el código 4 se observa la comparación entre

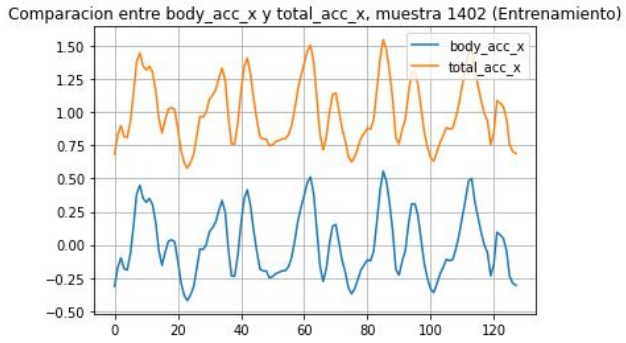
las señales de aceleración en el eje X, donde por cada conjunto se seleccionaron 2 muestras aleatorias.

Código 4: Comparativa de las señales de aceleración en el eje x.

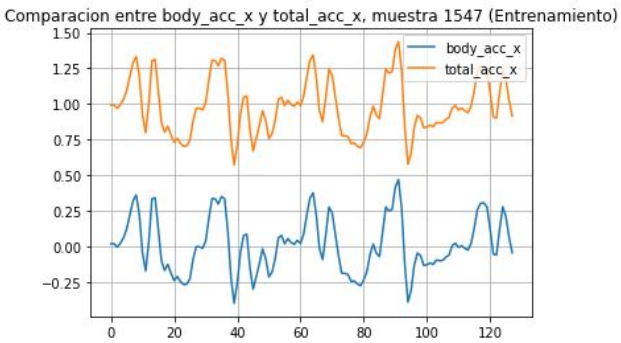
```
a = random.randint(0, 2799)
plt.plot(x,Entrena1.iloc[a],label='body_acc_x')
plt.plot(x,Entrena7.iloc[a],label='total_acc_x')
plt.title('Comparacion entre body_acc_x y total_acc_x, muestra '+str(a)+ '(Entrenamiento)')
plt.legend(loc='upper right')
plt.grid()
plt.show()

b = random.randint(0, 2799)
plt.plot(x,Entrena1.iloc[b],label='body_acc_x')
plt.plot(x,Entrena7.iloc[b],label='total_acc_x')
plt.title('Comparacion entre body_acc_x y total_acc_x, muestra '+str(b)+ '(Entrenamiento)')
plt.legend(loc='upper right')
plt.grid()
plt.show()

c = random.randint(0, 1999)
plt.plot(x,Vali1.iloc[c],label='body_acc_x')
plt.plot(x,Vali7.iloc[c],label='total_acc_x')
plt.title('Comparacion entre body_acc_x y total_acc_x, muestra '+str(c)+ '(Validacion)')
plt.legend(loc='upper right')
plt.grid()
plt.show()

d = random.randint(0, 1999)
plt.plot(x,Vali1.iloc[d],label='body_acc_x')
plt.plot(x,Vali7.iloc[d],label='total_acc_x')
plt.title('Comparacion entre body_acc_x y total_acc_x, muestra '+str(d)+ '(Validacion)')
plt.legend(loc='upper right')
plt.grid()
plt.show()

e = random.randint(0, 1999)
plt.plot(x,Prueba1.iloc[e],label='body_acc_x')
plt.plot(x,Prueba7.iloc[e],label='total_acc_x')
plt.title('Comparacion entre body_acc_x y total_acc_x, muestra '+str(e)+ '(Prueba)')
plt.legend(loc='upper right')
plt.grid()
plt.show()

f = random.randint(0, 1999)
plt.plot(x,Prueba1.iloc[f],label='body_acc_x')
plt.plot(x,Prueba7.iloc[f],label='total_acc_x')
plt.title('Comparacion entre body_acc_x y total_acc_x, muestra '+str(f)+ '(Prueba)')
plt.legend(loc='upper right')
plt.grid()
plt.show()
```

Luego las figuras obtenidas sobre las características de aceleración en el eje x se logran observar a continuación:

- **Conjunto de Entrenamiento:** Si se utilizan las señales body_acc_x y total_acc_x del conjunto de entrenamiento y se compara una muestra de cada uno se obtienen las figuras 1.a y 1.b.



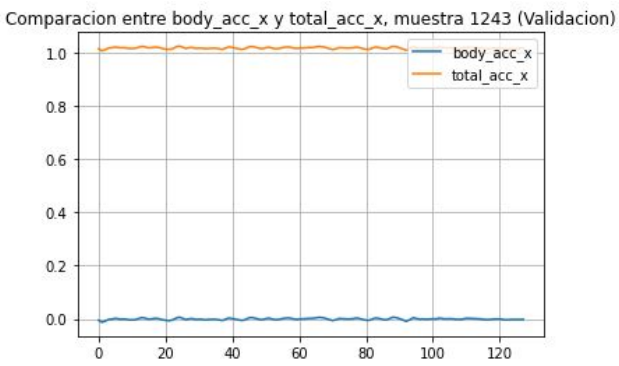
(a) Señales de aceleración en muestra 1402.



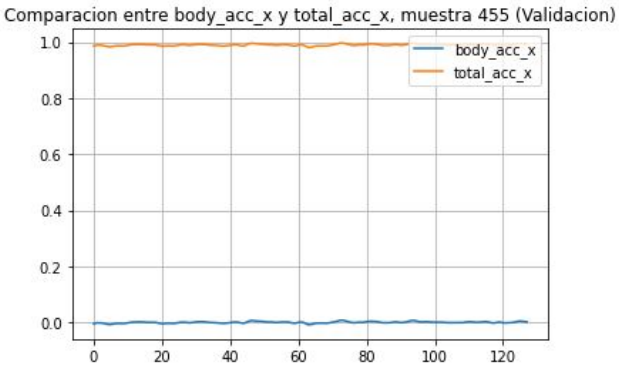
(b) Señales de aceleración en muestra 1547.

Figura 1: Señales de aceleración en el eje X sobre el conjunto de entrenamiento.

- **Conjunto de Validación:** Si se hace lo mismo que en el conjunto de entrenamiento se obtienen las figuras 2.a y 2.b.



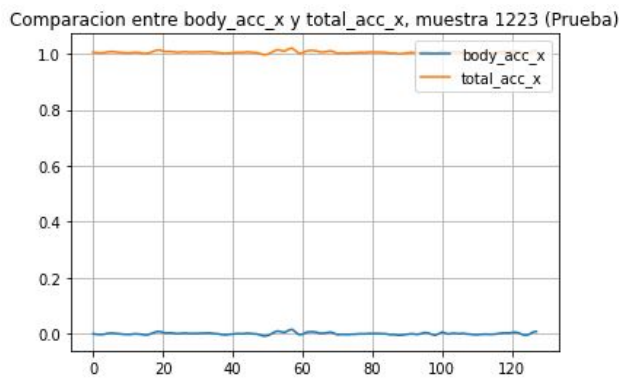
(a) Señales de aceleración en muestra 1243.



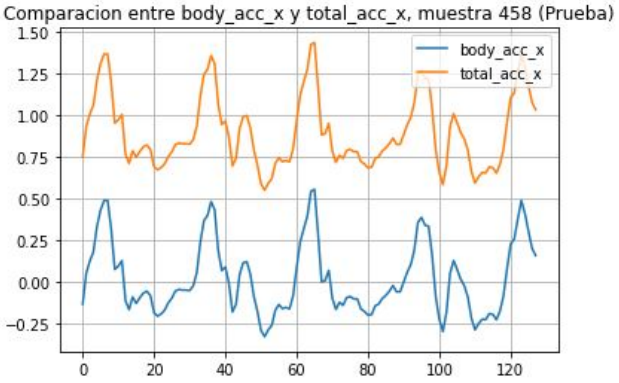
(b) Señales de aceleración en muestra 455.

Figura 2: Señales de aceleración en el eje X sobre el conjunto de validación.

- **Conjunto de Prueba:** Finalmente si se vuelve a repetir lo anterior sobre el conjunto de prueba se obtienen las figuras 3.a y 3.b.



(a) Señales de aceleración en muestra 1223.



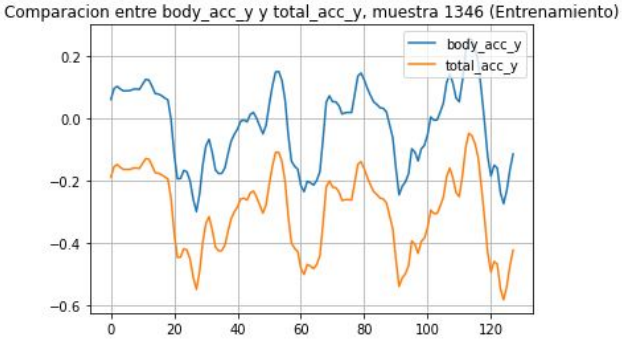
(b) Señales de aceleración en muestra 458.

Figura 3: Señales de aceleración en el eje X sobre el conjunto de prueba.

De las figuras mostradas anteriormente se logra observar que en términos generales la forma de las señales de aceleración sobre el eje x son iguales, basándose en este argumento se decidió descartar la señal body_acc_x, pues como se mencionó anteriormente esta señal simplemente es un filtrado pasabajo de la señal total_acc_x.
Se debe destacar que en este reporte solo se mostraron estos casos, sin embargo cuando se estuvo trabajando durante el proyecto se graficaron en repetidas ocasiones distintas muestras de las señales, donde siempre se observaba que las señales que se estaban comparando eran prácticamente idénticas. Repitiendo el procedimiento anterior para los pares de señales body_acc_y, total_acc_y y body_acc_z, total_acc_z se obtiene el mismo resultado. A continuación se exhibirá la comparativa en las señales mencionadas anteriormente:

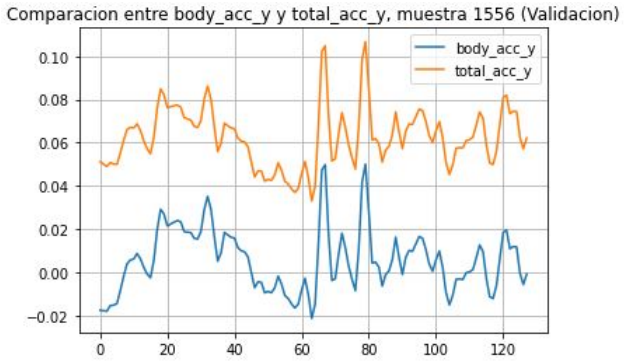
- **Señales body_acc_y y total_acc_y:** En las figuras 4.a y 4.b se observan las señales sobre el conjunto de entrenamiento, en las figuras 5.a y 5.b se observan las señales sobre el conjunto de validación, finalmente en las figuras 6.a y 6.b se observan las señales sobre el conjunto de prueba.

(a) Señales de aceleración en muestra 2292.

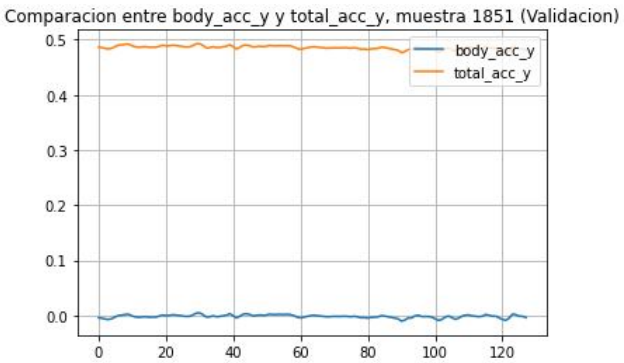


(b) Señales de aceleración en muestra 1346.

Figura 4: Señales de aceleración en el eje Y sobre el conjunto de entrenamiento.

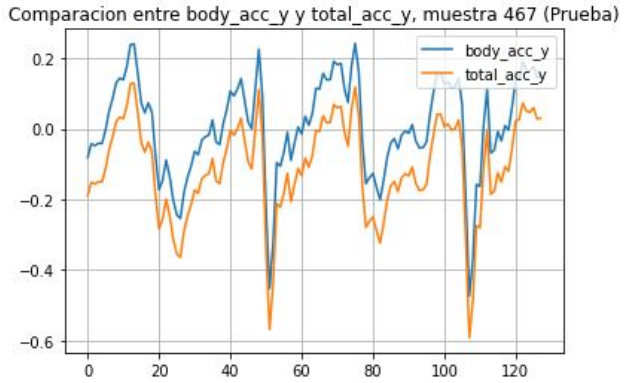


(a) Señales de aceleración en muestra 1556.

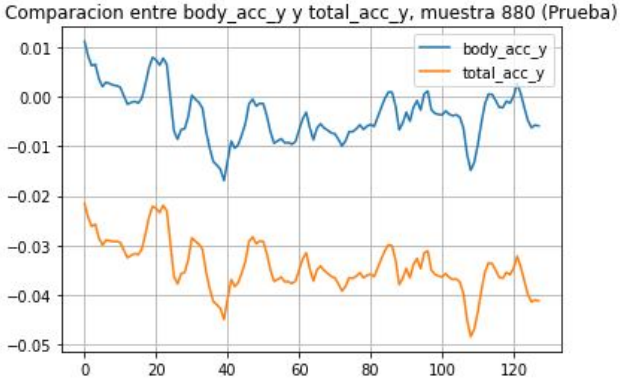


(b) Señales de aceleración en muestra 1851.

Figura 5: Señales de aceleración en el eje Y sobre el conjunto de validación.



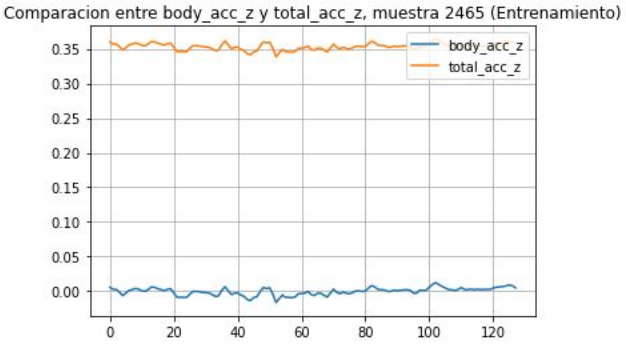
(a) Señales de aceleración en muestra 467.



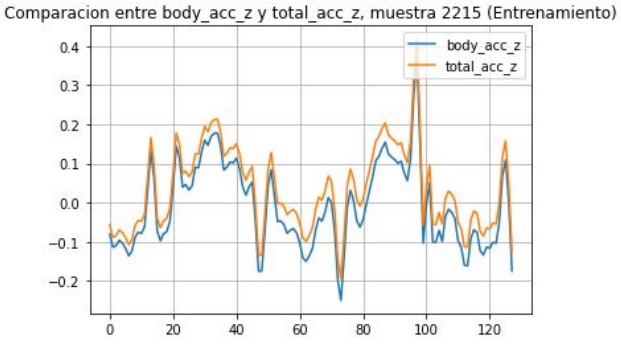
(b) Señales de aceleración en muestra 880.

Figura 6: Señales de aceleración en el eje Y sobre el conjunto de prueba.

- **Señales body_acc_z y total_acc_z:** En las figuras 7.a y 7.b se observan las señales sobre el conjunto de entrenamiento, en las figuras 8.a y 8.b se observan las señales sobre el conjunto de validación, finalmente en las figuras 9.a y 9.b se observan las señales sobre el conjunto de prueba.

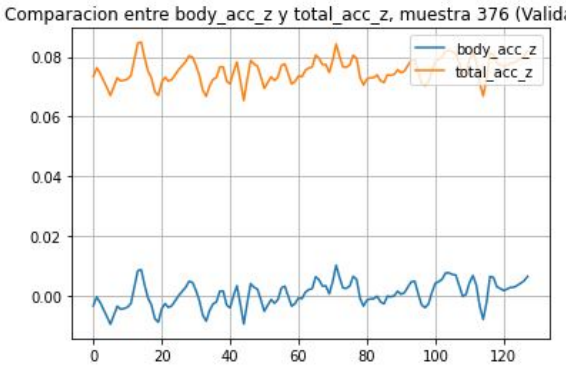


(a) Señales de aceleración en muestra 2465.

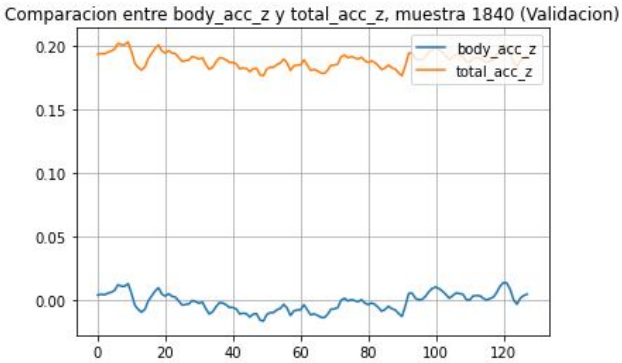


(b) Señales de aceleración en muestra 2215.

Figura 7: Señales de aceleración en el eje Z sobre el conjunto de entrenamiento.

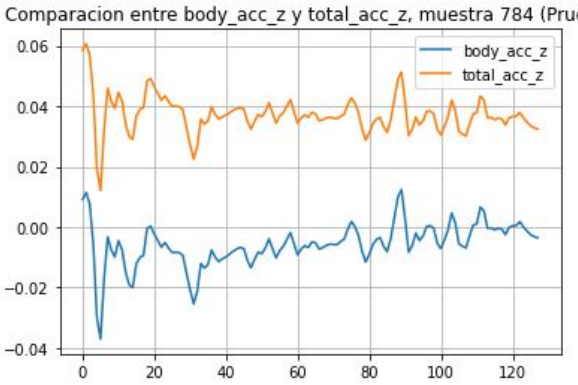


(a) Señales de aceleración en muestra 376.

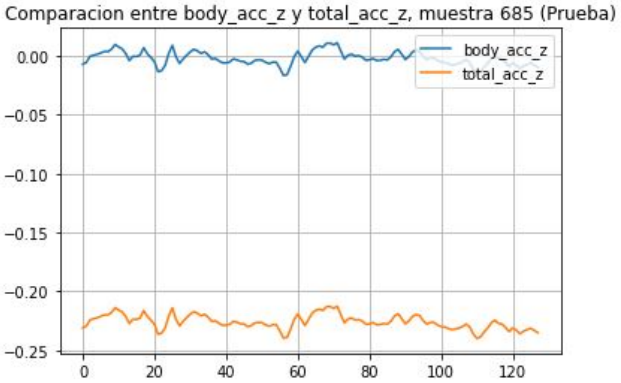


(b) Señales de aceleración en muestra 1840.

Figura 8: Señales de aceleración en el eje Z sobre el conjunto de validación.



(a) Señales de aceleración en muestra 784.



(b) Señales de aceleración en muestra 685.

Figura 9: Señales de aceleración en el eje Z sobre el conjunto de prueba.

Luego en base a todas las figuras previamente mostradas también se eliminaron las señales body_acc_y y body_acc_z, pues al igual que para el caso de las señales de aceleración en el eje x, solo son un filtrado pasabajo de las señales total_acc_y y total_acc_z.
Se debe mencionar que también se hizo una comparativa de las 3 señales de un eje (por ejemplo sobre el eje x se graficaron las señales body_acc_x, total_acc_x y body_gyro_x), sin embargo, la señal que indicaba el giroscopio difería mucho de las otras dos en cuanto a su forma, motivo por el cual esta no se eliminó ni tampoco se mostraron dichos graficos pues no aportaban a la explicación de este informe.

2.3. Creación de DataFrames a partir de características

Para la creación de los DataFrames se seleccionaron 7 características que en base a las distintas pruebas realizadas permiten que los clasificadores entreguen un mejor accuracy, las características elegidas corresponden al promedio, desviación estándar, máximo, mínimo, rango (Peak to Peak), Rango Intercuartil (Interquartil Range o iqr) y entropía. El código 5 es el código que realiza el cálculo de las características previamente mencionadas.

Código 5: Creación del DataFrame de entrenamiento.

```
#En este vector se guardan las señales con las que se va a trabajar
aux_Entrena = [Entrena4, Entrena5, Entrena6, Entrena7, Entrena8, Entrena9]
#Calculo de caracteristicas

#Promedio
M = []
for j in range(len(aux_Entrena)):
  N = []
  for i in range(Entrena1.shape[0]):
      N.append(aux_Entrena[j].iloc[i].mean())
  M.append(N)

#Desviacion estandar
M1 = []
for j in range(len(aux_Entrena)):
  N = []
  for i in range(Entrena1.shape[0]):
      N.append(aux_Entrena[j].iloc[i].std())
  M1.append(N)

#Maximo
M2 = []
for j in range(len(aux_Entrena)):
  N = []
  for i in range(Entrena1.shape[0]):
      N.append(aux_Entrena[j].iloc[i].max())
  M2.append(N)

#Minimo
M3 = []
for j in range(len(aux_Entrena)):
```

```
  N = []
  for i in range(Entrena1.shape[0]):
      N.append(aux_Entrena[j].iloc[i].min())
  M3.append(N)

#Peak to Peak
M4 = []
for j in range(len(aux_Entrena)):
  N = []
  for i in range(Entrena1.shape[0]):
      N.append(aux_Entrena[j].iloc[i].max()-aux_Entrena[j].iloc[i].min())
  M4.append(N)

#interquartile range
M7 = []
for j in range(len(aux_Entrena)):
  N = []
  for i in range(Entrena1.shape[0]):
      N.append(scipy.stats.iqr(aux_Entrena[j].iloc[i].to_numpy()))
  M7.append(N)

#Entropy
M8 = []
for j in range(len(aux_Entrena)):
  N = []
  for i in range(Entrena1.shape[0]):
    hist = np.histogram(aux_Entrena[j].iloc[i], bins=50)
    hist_norm = hist[0]/128
    N.append(scipy.stats.entropy(hist_norm, base=2)/50)
  M8.append(N)

prom = np.array(M)
prom = pd.DataFrame(prom)

std = np.array(M1)
std = pd.DataFrame(std)

max = np.array(M2)
max = pd.DataFrame(max)

min = np.array(M3)
min = pd.DataFrame(min)

ptp = np.array(M4)
ptp = pd.DataFrame(ptp)

iqr = np.array(M7)
iqr = pd.DataFrame(iqr)

ent = np.array(M8)
ent = pd.DataFrame(ent)
```

```
83
84 Entrena = pd.concat([prom, std, max, min, ptp, iqr, ent], axis=0)
85
86 nombres = ['Prom_bgx', 'Prom_bgy', 'Prom_bgz', 'Prom_tax', 'Prom_tay', 'Prom_taz',
87         'Std_bgx', 'Std_bgy', 'Std_bgz', 'Std_tax', 'Std_tay', 'Std_taz',
88         'Max_bgx', 'Max_bgy', 'Max_bgz', 'Max_tax', 'Max_tay', 'Max_taz',
89         'Min_bgx', 'Min_bgy', 'Min_bgz', 'Min_tax', 'Min_tay', 'Min_taz',
90         'Ptp_bgx', 'Ptp_bgy', 'Ptp_bgz', 'Ptp_tax', 'Ptp_tay', 'Ptp_taz',
91         'Iqr_bgx', 'Iqr_bgy', 'Iqr_bgz', 'Iqr_tax', 'Iqr_tay', 'Iqr_taz',
92         'Ent_bgx', 'Ent_bgy', 'Ent_bgz', 'Ent_tax', 'Ent_tay', 'Ent_taz']
93
94 Entrena = Entrena.transpose()
95 Entrena.columns = nombres
96 Entrena
```

En el principio del código anterior se observa la creación de un vector llamado aux_Entrena, dicho vector facilitará la obtención de características y contiene los dataframes con las señales que se van a utilizar. Luego el cálculo de características simplemente consiste en iterar sobre el vector previamente mencionado y sobre las filas de los DataFrames, de esta forma se obtuvieron todas las características mencionadas con anterioridad. Todas las características son fáciles de interpretar, pues las funciones encargadas de obtener la característica como tal simplemente recibe una fila de los distintos DataFrames con señales y entrega un valor, sin embargo la característica de Entropía es diferente, pues la función necesita recibir un vector de probabilidades, es por este motivo que a cada fila de las distintas señales se les calcula un histograma, luego se normalizan los histogramas por la cantidad de datos, pues de esta forma se logra obtener la probabilidad de los distintos bins, luego el vector de probabilidades se le entrega a la función *entropy()* de *scipy* y así se obtiene el resultado, sin embargo se debe mencionar que este cálculo de entropía arroja la entropía de los bins y no de los datos como tal. Se intentó implementar un código que obtuviese la entropía de los datos, sin embargo se necesitaban muchas iteraciones y el código tardaba mucho en ejecutarse, es por este motivo que se eligió calcular la entropía de los bins, pues de esta forma el tiempo de procesamiento era mucho menor.

La parte final del código anterior simplemente sirve para "armar" el DataFrame de entrenamiento pues se concatenan los DataFrames que contienen las características y se le da un nombre a cada columna del DataFrame de entrenamiento resultante.

La creación de los DataFrames de validación y prueba es análogo, motivo por el cual dicho código no fue explicado durante este informe.

2.4. Normalización de características

En el código 6 se logra observar la forma en que se normalizaron las características.

Código 6: Normalización de características.

```
1 #Normalizacion de caracteristicas
2 scaler = sklearn.preprocessing.StandardScaler()
3
4 scaler.fit(Entrena)
5
```

```
S_Entrena = scaler.transform(Entrena)
S_Vali = scaler.transform(Vali)
S_Prueba = scaler.transform(Prueba)

#Concatenamos conjunto de entrenamiento y validacion para que lo use el grid
Entrena_Vali = np.concatenate([S_Entrena, S_Vali])
Entrena_Vali_class = np.concatenate([labels_entrena, labels_vali])

#Vemos que indices del conjunto Entrena_Vali pertenecen al conjunto de entrenamiento
index = np.zeros(Entrena.shape[0]+Vali.shape[0])
for i in range(Entrena.shape[0]):
  index[i] = -1

#Definimos el predifinedSplit
ps = sklearn.model_selection.PredefinedSplit(index)
```

En el código anterior se logra observar que se utilizó la función *StandardScaler()* de *sklearn* para normalizar las características, donde la función *fit()* entrena al scaler con el conjunto de entrenamiento y luego mediante el uso de la función *transform()* se normalizan todos los conjuntos.

Por otra parte se logra observar que se concatenaron los conjuntos de entrenamiento y validación normalizados, luego se definió un vector cuyas celdas con valor -1 indican si el índice de dicha celda apunta a un elemento del conjunto de entrenamiento, las celdas con valor 0 apuntan a elementos del conjunto de validación. Lo anterior se hace con la finalidad de que cuando se utilice una grilla para buscar los mejores hiperparámetros del Random Forest se tenga en consideración a que conjunto pertenece cada muestra del conjunto entrenamiento y validación concatenados. Es importante mencionar que los labels de los conjuntos también fueron concatenados.

2.5. Clustering

El clustering implementado se logra observar en el código 7.

Código 7: Clustering.

```
#Creación del PCA
pca_Entrena = PCA(n_components=2).fit_transform(S_Entrena)

#DBSCAN
dbscan_cluster = DBSCAN().fit(pca_Entrena)
pred_cluster = np.array(dbscan_cluster.labels_).astype(int)
n_clusters = set(pred_cluster)

#Grafico de clustering
plt.figure(figsize=(10, 8))
for i in range(0,len(pred_cluster)):
  if pred_cluster[i] == 0:
    plt.scatter(pca_Entrena[i,0],pca_Entrena[i,1], label='Muestras PCA',color='m')
  elif pred_cluster[i] == 1:
    plt.scatter(pca_Entrena[i,0],pca_Entrena[i,1], label='Muestras PCA',color='y')
  elif pred_cluster[i] == 2:
```

```
    plt.scatter(pca_Entrena[i,0],pca_Entrena[i,1], label='Muestras PCA',color='orange')
  elif pred_cluster[i] == 3:
    plt.scatter(pca_Entrena[i,0],pca_Entrena[i,1], label='Muestras PCA',color='r')
  elif pred_cluster[i] == 4:
    plt.scatter(pca_Entrena[i,0],pca_Entrena[i,1], label='Muestras PCA',color='green')
  else:
    plt.scatter(pca_Entrena[i,0],pca_Entrena[i,1], label='Muestras PCA',color='b')
plt.xlabel('Columna 0')
plt.ylabel('Columna 1')
plt.title('Clustering sobre los datos de entrenamiento')
plt.grid()
plt.show()
```

El código anterior utiliza la función *pca()*, esta función permite llevar, en este caso, al conjunto de entrenamiento a dos dimensiones, luego mediante el uso de la función *DBSCAN()* se realiza el clustering de los datos, la idea de dicho clustering es lograr que se identifiquen 6 clusters, uno por cada clase que se quiere predecir. Luego simplemente se utiliza la función *scatter()* para graficar el clustering realizado.

El clustering obtenido tras utilizar el código anterior se logra observar en la figura 10 donde no se logran observar 6 clusters como se deseaba, sin embargo esta fue la mejor situación que se logró obtener, si se agregaban más características aparecían menos clusters o estos se mezclaban, algo similar ocurría si se utilizaban menos características.

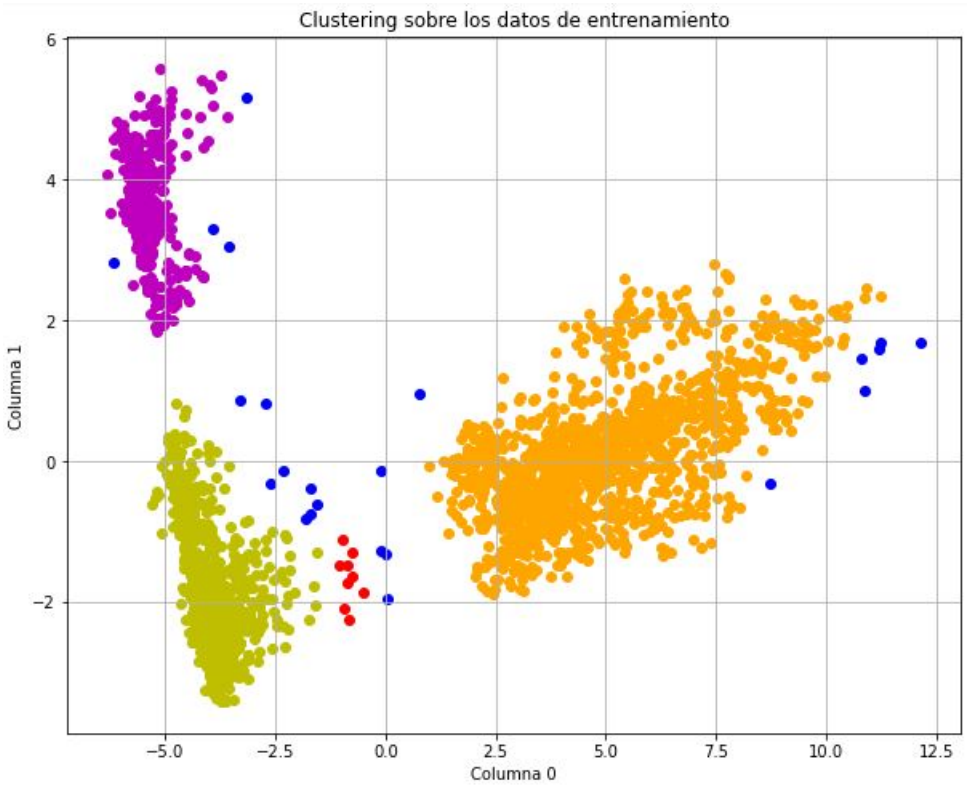


Figura 10: Clustering obtenido.

2.6. Random Forest

Como se ha mencionado anteriormente uno de los clasificadores implementados corresponde a un Random Forest, donde el código 8 es el que se encarga de implementar dicho clasificador.

Código 8: Random Forest.

```
#Random Forest que trabaja con todas las caracteristicas

#Mediante un random forest
Forest = sklearn.ensemble.RandomForestClassifier()
#Definimos una grilla para seleccionar los hiperparametros
GridForest = sklearn.model_selection.GridSearchCV(Forest, param_grid = {'n_estimators' : [150, 175, 200, 225, 250],
                                                                        'max_depth' : [15]}, cv=ps)

GridForest.fit(Entrena_Vali, Entrena_Vali_class)

param_Forest = GridForest.best_params_
print('Los mejores parametros antes de reducir caracteristicas son: '+str(param_Forest))
#Obtenemos el mejor estimador
estim_Forest = GridForest.best_estimator_

#Vemos la clasificacion del estimador sobre el conjunto de validacion
predi_vali_Forest = estim_Forest.predict(S_Vali)

#Accuracy del clasificador
accuracy_Forest = sklearn.metrics.accuracy_score(labels_vali, predi_vali_Forest)
print('El accuracy sin reducir caracteristicas es (Validacion): '+str(accuracy_Forest))

predi_test_Forest = estim_Forest.predict(S_Prueba)

print('Los labels predichos sobre el conjunto de prueba al no reducir caracteristicas son:')
print(predi_test_Forest)

#Obtenemos la matriz de confusión en el conjunto de validacion
Confu = sklearn.metrics.confusion_matrix(labels_vali, predi_vali_Forest, normalize='true')

print('')
print('-----------------------------------------------')
print('')
#-------------------------------------------------------------------------------
#Hacemos el modelo con el que buscaremos reducir caracteristicas
model_Forest = sklearn.feature_selection.SelectFromModel(estim_Forest)

#Entrenamos el model con el conjunto de entrenamiento
model_Forest.fit(S_Entrena, labels_entrena)

#Obtenemos las caracteristicas seleccionadas
caracte_Forest = model_Forest.get_support(indices=False)

```

```
#Recorremos la lista de caracteristicas seleccionadas y las añadimos a una lista donde se consideran
#solo a las caracteristicas seleccionadas
select_carac_Forest = []
for i in range(len(caracte_Forest)):
  if caracte_Forest[i] == True:
    select_carac_Forest.append(nombres[i])
print('Se seleccionaron '+str(len(select_carac_Forest))+' caracteristicas')
print('Las caracteristicas seleccionadas son: '+str(select_carac_Forest))

#Obtenemos los distintos conjuntos con las caracteristicas reducidas
Entrena_Vali_redu_Forest = model_Forest.transform(Entrena_Vali) #<--- Entrenamiento y validacion concatenados
Vali_redu_Forest = model_Forest.transform(S_Vali) #<----- Conjunto de validacion
Test_redu_Forest = model_Forest.transform(S_Prueba) #<----- Conjunto de prueba

#-------------------------------------------------------------------------

#RandomForest que trabaja con las caracteristicas reducidas
Forest1 = sklearn.ensemble.RandomForestClassifier()

#Definimos una grilla para seleccionar los hiperparametros
GridForest1 = sklearn.model_selection.GridSearchCV(Forest1, param_grid = {'n_estimators' : [150, 175, 200, 225, 250],
                                                                          'max_depth' : [15]}, cv=ps)

GridForest1.fit(Entrena_Vali_redu_Forest, Entrena_Vali_class)

param_Forest1 = GridForest1.best_params_
print('Los mejores parametros al reducir caracteristicas son: '+str(param_Forest1))
#Obtenemos el mejor estimador
estim_Forest1 = GridForest1.best_estimator_

#Vemos la clasificacion del estimador sobre el conjunto de validacion
predi_vali_Forest1 = estim_Forest1.predict(Vali_redu_Forest)

#Accuracy del clasificador
accuracy_Forest1 = sklearn.metrics.accuracy_score(labels_vali, predi_vali_Forest1)
print('El accuracy al reducir caracteristicas es (validacion): '+str(accuracy_Forest1))

predi_test_Forest1 = estim_Forest1.predict(Test_redu_Forest)

print('Los labels predichos sobre el conjunto de prueba al reducir caracteristicas son:')
print(predi_test_Forest1)


#Obtenemos la matriz de confusión en el conjunto de validacion
Confu1 = sklearn.metrics.confusion_matrix(labels_vali, predi_vali_Forest1, normalize='true')
```

El código anterior está dividido en 3 porciones, a continuación se detallará que es lo que busca hacer cada una.

La primera porción de código se encarga de entrenar y obtener la predicción que genera un Random Forest que utiliza todas las características. Primero se define el clasificador Random Forest mediante el uso de la función *RandomForestClassifier()*, luego se define una grilla, este grilla se encargará de elegir los mejores hiperparámetros para el clasificador, en este punto se debe destacar el hecho de que la profundidad máxima del Random Forest (el hiperparámetro "max_depth") se fijó en un valor de 15, esto se hizo pues si se le daba a la grilla la opción de elegir la profundidad del clasificador se producían casos en que se elegía una profundidad muy pequeña, lo que provocaba que el accuracy decayera demasiado. Luego se entrena la grilla mediante el uso de la función *fit()*, una vez entrenada la grilla se obtuvieron los mejores parámetros del clasificador, esto se hace mediante el uso de la función *best_params_*, cabe destacar que esto se hizo solo para que el usuario que esté utilizando el código sepa cuales son los hiperparámetros seleccionados. Posteriormente se eligió el mejor estimador que se desprendió de la grilla, dicho estimador será aquel que tiene los mejores hiperparámetros, luego solo se obtuvo la predicción que realiza el Random Forest obtenido sobre el conjunto de validación y prueba, esto se hace utilizando la función *predict()*. Finalmente se obtuvo el accuracy y matriz de confusión del estimador sobre el conjunto de validación, esto se hace mediante el uso de la función *accuracy_score()* y *confusion_matrix()*, respectivamente.

La segunda porción de código es la que se encarga de reducir características. Primero se define la forma con la que se reducirán características, esto se hace mediante el uso de la función *SelectFromModel()*, dicha función recibe como parámetro el mejor estimador que se desprendió de la grilla obtenida durante la primera porción del código y se encarga de implementar una estrategía de selección de características llamada *Wrapper*. Una vez se definió el modelo con el que se reducirán características se entrena y se obtienen las características seleccionadas mediante el uso de la función *get_support()*, esta función retorna un vector con valores booleanos, donde si se tiene un valor booleano True en la posición i del vector significa que la característica i seguirá estando presente en el conjunto de características reducidas. Lo que sigue del código es una iteración sobre el vector obtenido por la función antes mencionada, esta iteración permitirá que el usuario tenga conocimiento de las características que fueron seleccionadas. Luego para terminar esta porción de código los conjuntos de entrenamiento, validación y prueba son reducidos mediante el uso de la función *transform()*.

La última porción de código es idéntica a la primera, solo que acá se trabajan con los conjuntos reducidos, por este motivo es que dicha porción de código no será explicada, pues es análoga a la primera porción de código previamente explicada.

Los datos que entrega el código anterior se pueden observar en el código 9, sin embargo se debe mencionar que este código tiene variaciones, no siempre se obtienen los mismos hiperparámetros o el mismo conjunto de características reducido.

Código 9: Datos que se desprenden del Random Forest.

```
Los mejores parametros antes de reducir caracteristicas son: {'max_depth': 15, 'n_estimators': 175}
El accuracy sin reducir caracteristicas es (Validacion): 1.0
Los labels predichos sobre el conjunto de prueba al no reducir caracteristicas son:
[3 3 1 ... 3 4 4]

-------------------------------------------------

Se seleccionaron 11 caracteristicas
```

```
Las caracteristicas seleccionadas son: ['Prom_tax', 'Prom_tay', 'Std_tax', 'Max_tax', 'Max_tay', '
    ↪ Min_tax', 'Min_tay', 'Ptp_tax', 'Ptp_tay', 'Iqr_tax', 'Iqr_tay']
Los mejores parametros al reducir caracteristicas son: {'max_depth': 15, 'n_estimators': 175}
El accuracy al reducir caracteristicas es (validacion): 0.9995
Los labels predichos sobre el conjunto de prueba al reducir caracteristicas son:
[3 3 0 ... 3 4 4]
```

Las matrices de confusión obtenidas sobre el conjunto de validación para la situación antes mostrada se observan en la figura 11 y 12, donde la primera de las figuras corresponde a la matriz de confusión obtenida al utilizar todas las características y la segunda es la matriz de confusión obtenida al utilizar el conjunto de características reducido.

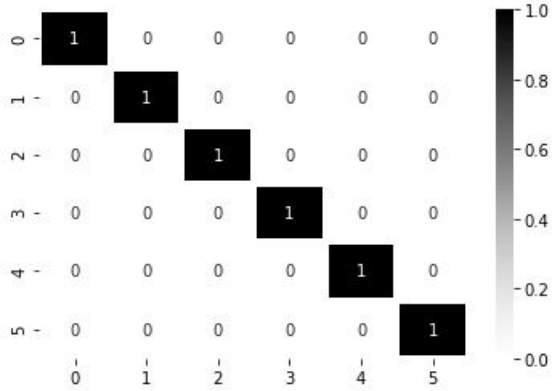


Figura 11: Matriz de confusión del Random Forest que utiliza todas las características.

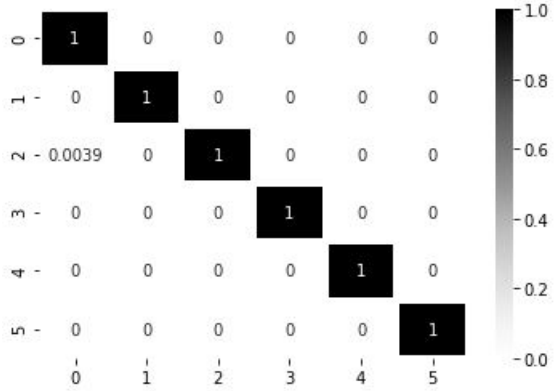


Figura 12: Matriz de confusión del Random Forest que utiliza el conjunto de características reducido.

Finalmente la predicción realizada sobre el conjunto de prueba se guardó en un archivo csv que fue subido al entorno de trabajo en el archivo .zip llamado “Resultados”, el archivo csv en el que se guardan los resultados se llama “ResultadosRF.csv”. El código que se encarga de guardar la predicción sobre el conjunto de prueba se observa en el código 10.

Código 10: Predicción realizada sobre el conjunto de prueba.

```
#Creacion del csv para subir guardar los resultados de la prediccion realizada
id = []
for i in range(len(predi_test_Forest1)+1):
  if i == 0:
    id.append('Id')
  else:
    id.append(i-1)

#Accuracy en validacion es mayor o igual con todas las características
if accuracy_Forest >= accuracy_Forest1:
  Resultados_Test_Forest = np.array(['Category'])
  Resultados_Test_Forest = np.append(Resultados_Test_Forest, predi_test_Forest+1)

  final = np.array([id,Resultados_Test_Forest])
  final = np.transpose(final)

  #Aca guardamos los resultados de la predicción hecha al reducir caracteristicas
  myFile = open('ResultadosRF.csv', 'w')
  with myFile:
     writer = csv.writer(myFile)
     writer.writerows(final)
  print('Los labels de prueba se obtuvieron con todas las caracteristicas')

#Accuracy en validacion es mayor con las características reducidas
else:
  Resultados_Test_Forest1 = np.array(['Category'])
  Resultados_Test_Forest1 = np.append(Resultados_Test_Forest1, predi_test_Forest1+1)

  final = np.array([id,Resultados_Test_Forest1])
  final = np.transpose(final)

  #Aca guardamos los resultados de la predicción hecha al reducir caracteristicas
  myFile = open('ResultadosRF.csv', 'w')
  with myFile:
     writer = csv.writer(myFile)
     writer.writerows(final)
  print('Los labels de prueba se obtuvieron con las caracteristicas reducidas')

res = pd.read_csv('ResultadosRF.csv')
res
```

El código anterior guarda la predicción realizada sobre el conjunto de prueba dependiendo de si el accuracy obtenido en el conjunto de validación es mayor al considerar todas las características o si es mayor al considerar el conjunto de características reducido. Se debe mencionar el hecho de que en la mayoría de las ocasiones el resultado que se guarda en el archivo csv corresponde a la predicción que se obtuvo al utilizar todas las características, pues en ese caso es cuando se obtiene un accuracy en validación más alto, sin embargo existen ocasiones en que esto no es así. Por otro lado se debe mencionar el hecho de que a los labels predichos por el clasificador se les sumó 1, esto se hace para que a la hora de subir el archivo con las predicciones a la plataforma Kaggle estén en el rango de 1 a 6.

2.7. Red Neuronal

El segundo clasificador implementado consiste en una red neuronal, a continuación se irán mostrando las diferentes porciones de código que se utilizan para implementar dicho clasificador.

El código 11 corresponde al código implementado para crear los dataloaders de entrenamiento y validación que utilizará la red neuronal, mientras que el código 12 se encarga de hacer esta misma tarea pero en el conjunto de prueba.

Código 11: Creación de dataloaders de entrenamiento y validación.

```
#Dataloader de entrenamiento y validación
dataset_Entrena_Vali_Concat = []

for i in range(Entrena1.shape[0]):
  M = []
  for j in range(len(aux_Entrena)):
    M.append(aux_Entrena[j].iloc[i].astype(np.float32))
  dataset_Entrena_Vali_Concat.append({"signal": np.array(M), "label": labels_entrena[i]})

for i in range(Vali1.shape[0]):
  M = []
  for j in range(len(aux_Vali)):
    M.append(aux_Vali[j].iloc[i].astype(np.float32))
  dataset_Entrena_Vali_Concat.append({"signal": np.array(M), "label": labels_vali[i]})

#85 % sera parte del conjunto de entrenamiento y el 15 % restante sera de validacion
n_train = int(len(dataset_Entrena_Vali_Concat)*0.85)
n_val = len(dataset_Entrena_Vali_Concat) - n_train

dataset_train, dataset_vali = random_split(dataset_Entrena_Vali_Concat, [n_train, n_val])

dataloader_train = torch.utils.data.DataLoader(dataset_train, batch_size=128, shuffle=True,
    ↪ num_workers=0)
dataloader_vali = torch.utils.data.DataLoader(dataset_vali, batch_size=128, shuffle=True,
    ↪ num_workers=0)
```

Código 12: Creación de dataloader de prueba.

```
#Dataloader de prueba
dataset_Prueba = []

for i in range(Prueba1.shape[0]):
  M = []
  for j in range(len(aux_Prueba)):
    M.append(aux_Prueba[j].iloc[i].astype(np.float32))
  dataset_Prueba.append({"signal": np.array(M)})

dataloader_test = torch.utils.data.DataLoader(dataset_Prueba, batch_size=1, shuffle=False,
    ↪ num_workers=0)
```

Los códigos anteriores funcionan de forma prácticamente idéntica, motivo por el cual solo se explicará el código 11. El código recién mencionado se encarga de crear los dataloaders de entrenamiento y validación, esto se logra mediante el uso de iteraciones. Primero se itera sobre las señales del conjunto de entrenamiento de la forma que sigue: Se toma la primera fila de la señal 1 del conjunto de entrenamiento y se guarda en un vector, luego se toma la misma fila de la señal 2 y se guarda en el mismo vector, esto se repite para las 6 señales, una vez se realiza esto para las 6 señales se guarda el vector que contiene la primera fila de las 6 señales en un diccionario con la llave "signal" mientras que en la llave "label" se guarda el label correcto de dicha fila, esto se repite para todas las filas de todas las señales del conjunto de entrenamiento. Luego se repite lo mismo para el conjunto de validación, acá se debe mencionar que los diccionarios creados para este conjunto se concatenan con los diccionarios obtenidos en el conjunto de entrenamiento, más adelante se explicará el motivo de esto.

Una vez se tienen todos los diccionarios concatenados se divide el vector que contiene los diccionarios en 2 partes, la primera parte contiene el 85 % de los diccionarios creados y la segunda parte tiene el 15 % restante, donde los diccionarios que tiene cada parte fueron elegidos al azar mediante el uso de *random_split()*. Luego el dataloader de entrenamiento se crea con el conjunto que tiene el 85 % de los diccionarios y el de validación se crea con el conjunto que tiene el 15 % restante, esto se hizo para que la red tenga una mayor cantidad de datos de entrenamiento, esto permite que la red arroje accuracy's más altos ya que la red estará más entrenada.

El código 13 consiste en la creación de la red a utilizar, dicha red tiene 2 capas convolucionales de 40 neuronas, dichas capas utilizan un kernel de tamaño 9, luego de dichas capas convolucionales se tiene una capa lineal.

Código 13: Creación de la red.

```
#Creamos el modelo de la red
model = nn.Sequential(
        nn.Conv1d(6, 40, 9),
        nn.ReLU(),
        nn.Conv1d(40, 40, 9),
        nn.Flatten(),
        nn.Linear(4480, 6))

device = torch.device('cuda')

model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

El entrenamiento de la red antes mostrada se logra observar en el código 14.

Código 14: Entrenamiento de la red.

```
prom_loss_train = [] #Promedio de loss del entrenamiento para cada epoca
prom_loss_vali = [] #Promedio del loss de validacion para cada epoca
check = 10
epocas = range(check-1,1000,check)
```

```
prom_loss_train_def = [] #Promedio de loss cada 10 epocas (Entrenamiento)
prom_loss_vali_def = [] #Promedio loss cada 10 epocas (Validacion)
for epoch in range(1000):
    try:
      model.train()

      # Train on the current epoch
      loss_epoch_train = [] #Acá se guarda el loss de cada dato de la epoca
      for i, data in enumerate(dataloader_train, 0):
          inputs = data['signal'].to(device)
          labels = data["label"].to(device)

          # zero the parameter gradients
          optimizer.zero_grad()

          # forward + backward + optimize
          outputs = model(inputs)

          loss_train = criterion(outputs, labels) #calculo del loss de entrenamiento por dato
          loss_epoch_train.append(loss_train)     #guardamos el loss del dato en un vector
          loss_train.backward()
          optimizer.step()
      prom_train = sum(loss_epoch_train)/len(loss_epoch_train) #Calculamos el loss promedio de
      ↪ la epoca
      prom_loss_train.append(prom_train) #Guardamos el promedio en un vector

      # Compute validation loss and accuracy for current epoch
      model.eval()

      with torch.no_grad():
        loss_epoch_vali = [] #Acá se guarda el loss de cada dato de la epoca
        for i, data in enumerate(dataloader_vali, 0):
            inputs = data["signal"].to(device)
            labels = data["label"].to(device)

            outputs = model(inputs)
            # Calcular loss de validación
            loss_vali = criterion(outputs, labels) #calculo del loss de validación por dato
            loss_epoch_vali.append(loss_vali)      #guargamos el loss del dato en un vector
        prom_vali = sum(loss_epoch_vali)/len(loss_epoch_vali) #Calculamos el loss promedio de la
      ↪ epoca
        prom_loss_vali.append(prom_vali)

      # Imprimir: numero de época, loss de entrenamiento y loss de validación
      # Se debe usar sys.stdout.write() para que la línea de texto se sobreescriba en vez de imprimirse
      ↪  línea por línea
      # No se debe guardar checkpoints en cada época (guardarlos cada 10 épocas)

      sys.stdout.write("\rÉpoca %d, loss entrenamiento %f, loss validacion %f" % (epoch,
      ↪ prom_train, prom_vali))

```

```
52      #Hacemos el checkpoint cada 10 epocas y revisamos si debemos interrumpir el entrenamiento
53      if epoch in epocas:
54        torch.save(model, 'red'+str(epocas.index(epoch))+'.pt')
55        s_train = 0 #Suma de los loss de las 10 epocas anteriores (entrenamiento)
56        s_vali = 0 #Suma de los loss de las 10 epocas anteriores (validacion)
57        for i in range(epoch-check,epoch):
58          s_train = prom_loss_train[i]+s_train
59          s_vali = prom_loss_vali[i]+s_vali
60        prome_train = s_train/check #Loss promedio de las 10 epocas (Entrenamiento)
61        prome_vali = s_vali/check #Loss promedio de las 10 epocas (Validacion)
62        prom_loss_train_def.append(prome_train)
63        prom_loss_vali_def.append(prome_vali)
64        #Si el loss de validacion actual es mayor que el de la vez pasada termino
65        #El entrenamiento
66        if prom_loss_vali_def[len(prom_loss_vali_def)-1] > prom_loss_vali_def[len(
      ↪ prom_loss_vali_def)-2]:
67          print('\nEl loss de validacion empezo a aumentar en la epoca '+str(epoch))
68          print('Por lo tanto nos quedaremos con la red de la epoca '+str(epoch-check))
69          model = torch.load('red'+str(epocas.index(epoch)-1)+'.pt') #Nos quedamos con la red
      ↪ anterior
70          break
71
72    except KeyboardInterrupt:
73      print("\nEntrenamiento interrumpido")
74      break
75
76  print('\nEntrenamiento finalizado')
77
78  #Grafico del loss por epoca
79  graf = len(prom_loss_train)-check #Esto es para graficar hasta el momento en que el loss fue
      ↪ minimo
80  plt.plot(range(graf), prom_loss_train[:graf], label='Entrenamiento')
81  plt.plot(range(graf), prom_loss_vali[:graf], label='Validacion')
82  plt.grid()
83  plt.legend(loc='upper right')
84  plt.xlabel('Numero de epocas')
85  plt.ylabel('Error')
86  plt.title('Loss por epoca')
```

El código anterior opera como sigue: primero se define un par de vectores que se encargan de almacenar el loss promedio por época, un vector guarda el loss promedio del entrenamiento en la época y el otro el loss promedio del conjunto de validación en la época. Luego se define una variable llamada check, variable que fue fijada en 10, dicha variable se encarga de indicar en que epoca se debe observar si el loss de validación ha comenzado a aumentar, en cuyo caso se debe detener el entrenamiento, luego se define un vector que corresponde a las épocas en que el algoritmo tendrá que chequear si se debe detener el entrenamiento, este vector va desde 9 a 1000 dando pasos de 10, es decir, en la época 9 (correspondiente a la época 10 pues *Python* considera la época 0 como la época 1) se revisa por primera vez si se debe detener el entrenamiento, luego se vuelve a revisar en la época 19, 29 y así sucesivamente, hasta que la condición de "detención" del entrenamiento se cumpla. Finalmente se tiene otro par de vectores, estos almacenan el loss promedio cada 10 épocas,

debido a lo anterior guardarán un valor cada 10 épocas, para clarificar se enunciará un ejemplo. Supongamos que el código va iterando en la época 79, al ser una época en que se debe revisar si se debe detener el entrenamiento se calcula el loss promedio de las 10 épocas anteriores, es decir, los loss promedio almacenados en los vectores "prom_loss_train" y "prom_loss_vali" desde la celda 69 a la 79 se promedian y el valor obtenido se guarda en los vectores "prom_loss_train'_def" y "prom_loss_vali_def" según corresponda.

Con lo anterior explicado se procederá a explicar como funciona el código que se encarga de realizar el entrenamiento como tal. Al momento de comenzar a entrenar en cada época se define otro vector, vector que se encargará de guardar el loss del dato que se está revisando, luego, una vez se han revisado todos los datos, se calcula el promedio de todos los loss del conjunto de entrenamiento (loss que están guardados en el vector "loss_epoch_train") y el valor obtenido se añade al vector "prom_loss_train". Luego al momento de evaluar el modelo implementado en el conjunto de validación se procede a repetir exactamente lo mismo, se define un vector en el que se guardará el loss de cada dato y luego se calcula el promedio de ese vector y el resultado se añadirá al vector "prom_loss_vali". Una vez se entrenó el modelo y se evaluó en el conjunto de validación simplemente se muestran en pantalla la época actual, el loss promedio de la época en el conjunto de entrenamiento y el loss promedio de la época en el conjunto de validación, donde se utilizó la función *sys.stdout.write()* para que la línea se vaya sobrescribiendo y no se muestre una para cada época. Finalmente se tiene una condición, esta corresponde a que si el código está iterando en una época múltiplo de 10 se debe chequear si detener el entrenamiento, dentro de esta condición se guarda el modelo que está implementado hasta el momento, para esto se hace uso de la función *torch.save()*, la finalidad de guardar los modelos implementados es la de dejar un "checkpoint" de la red implementada al momento de revisar si detener el entrenamiento para así recuperarla en caso de que sea necesario, luego simplemente se calcula el promedio de los promedios de los loss de entrenamiento y validación y se añaden a un vector, llegado a ese punto de la ejecución del código se debe revisar si el loss promedio de validación actual es mayor que el de las 10 épocas anteriores, si esto pasa se debe detener el entrenamiento, motivo por el cual se recupera el modelo guardado en las 10 épocas anteriores (momento en que el loss fue mínimo) y se rompe el bucle, de forma tal que el entrenamiento se detenga. Finalmente se muestra el gráfico de loss por época para el conjunto de validación y entrenamiento, dicho gráfico se logra observar en la figura 13.

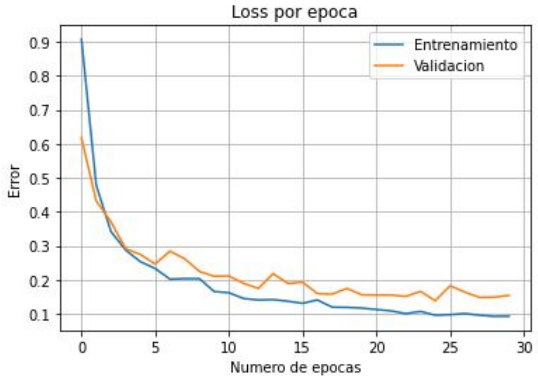


Figura 13: Loss de Entrenamiento y validación.

Finalmente se tienen dos porciones de código que son prácticamente idénticas, estas porciones

de código se encargan de obtener la predicción que genera la red sobre el conjunto de validación y sobre el conjunto de prueba. En el código 15 se logra observar el código utilizado para obtener la clasificación que realiza la red sobre el conjunto de validación, donde para el ejemplo mostrado se obtuvo un accuracy en validación de 0.9430555555555555 y la matriz de confusión que se observa en la figura 14. Por otro lado en el código 16 se observa la forma en que se obtiene la clasificación que realiza la red en el conjunto de prueba.

Código 15: Predicción de la red sobre el conjunto de validación.

```
predict_vali = []
class_vali = []
for i, data in enumerate(dataloader_vali, 0):
  inputs = data["signal"].to(device)
  labels = data["label"].to(device)

  #Obtenemos las salidas
  outputs = model(inputs)
  #Obtenemos las predicciones
  pred = outputs.cpu().argmax(axis=1)
  #A continuacion obtendremos la prediccion hecha para cada elemento y la
  #guardaremos en un vector, lo mismo hacemos con los labels
  for i in range(len(pred)):
    predict = pred[i].numpy().item()
    clas = labels[i].cpu().numpy().item()
    predict_vali.append(predict)
    class_vali.append(clas)

accuracy_vali = sklearn.metrics.accuracy_score(class_vali, predict_vali, normalize=True)
print('El accuracy en el conjunto de validacion es: '+str(accuracy_vali))

#Obtenemos la matriz de confusión en el conjunto de validacion
Confu2 = sklearn.metrics.confusion_matrix(class_vali, predict_vali, normalize='true')
print('Matriz de confusion al reducir caracteristicas (Validacion)')
sns.heatmap(Confu2, linewidths=3, annot=True, cmap="Greys")
```

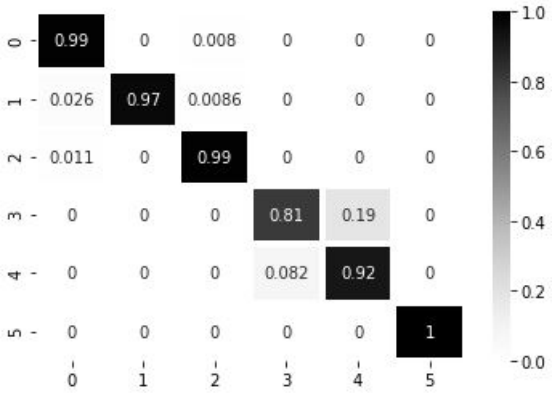


Figura 14: Matriz de confusión de la red sobre el conjunto de validación utilizado.

Código 16: Predicción de la red sobre el conjunto de prueba.

```
predict_Red = []

for i, data in enumerate(dataloader_test, 0):
  inputs = data["signal"].to(device)

  #Obtenemos las salidas
  outputs = model(inputs)
  #Obtenemos las predicciones
  pred = outputs.cpu().argmax(axis=1)
  #A continuacion obtendremos la prediccion hecha para cada elemento y la
  #guardaremos en un vector, lo mismo hacemos con los labels
  for i in range(len(pred)):
    predict = pred[i].numpy().item()
    predict_Red.append(predict)

predict_Red = np.array(predict_Red)
Resultados_Red = np.array(['Category'])
Resultados_Red = np.append(Resultados_Red, predict_Red+1)

final1 = np.array([id,Resultados_Red])
final1 = np.transpose(final1)

myFile = open('ResultadosRN.csv', 'w')
with myFile:
    writer = csv.writer(myFile)
    writer.writerows(final1)

res2 = pd.read_csv('ResultadosRN.csv')
res2
```

2.8. Resultados

En esta subseccion se presentarán los resultados obtenidos sobre el conjunto de pureba ante distintas configuraciones de los clasificadores. Se debe mencionar el hecho de que no fue posible obtener las matrices de confusión de los clasificadores sobre el conjunto de prueba, pues no se disponía de los labels correctos del conjunto de prueba.

- **Random Forest:** A continuación se expondrán 3 resultados obtenidos para el clasificador Random Forest.

  1. **Primer caso:** El primer caso consiste en el Random Forest que fue mostrado en el código 9, es decir, un Random Forest con hiperparámetros "max_depth" igual a 15 y "n_estimators" igual a 175. Dicho caso obtuvo un accuracy de 0.93833 en el conjunto de prueba, cabe destacar que la predicción realizada fue al utilizar todas las características y que el accuracy mencionado fue el obtenido en el leaderboard público de la plataforma Kaggle.
  2. **Segundo caso:** Este caso consiste en un Random Forest con hiperparámetros "max_depth" igual a 15 y "n_estimators" igual a 100, en este caso se consideró el conjunto de características reducido y se obtuvo un accuracy de 0.91833.

3. **Tercer caso:** El último caso consiste en un Random Forest con hiperparámetros "max_depth" igual a 15 y "n_estimators" igual a 150, en este caso también se consideró el conjunto de características reducido pero se tenía otra configuración de características, las características con las que se trabajaba era con: Promedio, máximo, mínimo, rango, desviación estandar y otra característica denominada media cuadrática, además se seleccionaron características mediante el uso de una estrategía de tipo filtro, donde se rescataron 11 características. Ante esta configuración se obtuvo un accuracy en el conjunto de prueba de 0.85333.

- **Red Neuronal:** Ante esta situación se obtuvieron los siguientes resultados:

1. **Primer caso:** Este caso corresponde al obtenido al utilizar la red mostrada durante este informe, en cuyo caso se obtuvo un accuracy del 0.94500 en el conjunto de prueba.
2. **Segundo caso:** El segundo caso consiste en una red con dos capas convolucionales de 50 neuronas y kernel de tamaño 5, seguidas de dos capas lineales, una de 6000 neuronas y la otra de 100, cabe destacar que en este caso no se tenían las señales del conjunto de entrenamiento y validación concatenadas. En este caso se obtuvo un accuracy de 0.88833.
3. **Tercer caso:** El último caso consiste en una red que trabajaba con características, y que por tanto no implementaba convoluciones, las características que se utilizaron eran promedio, desviación estándar, máximo, mínimo, rango, Skew, Kurtosis y Rango intercuartil. La red consistía en una capa lineal de 80 neuronas seguida de otra capa lineal con 100 neuronas, en este caso se obtuvo un accuracy en el conjunto de prueba de 0.89833.

3. Conclusión

A modo de conclusión se puede dar por completado el proyecto final del curso EL4106 Inteligencia Computacional, donde se logró implementar dos clasificadores que entregan un accuracy bastante alto sobre el conjunto de prueba.

A modo de aprendizaje se destaca fuertemente el uso de las señales de entrenamiento y validación concatenadas para entrenar la red, esto fue altamente útil pues permitió aumentar el accuracy en el conjunto de prueba, lo cual tiene sentido pues la red estará más "experimentada" a la hora de entregar una clasificación.

A modo de mejora se destaca el hecho de que no se hizo preprocesamiento de señales, esto se hizo principalmente porque no se vio necesario, además que en cierto punto del proyecto se implementó un filtrado pasabajo en las señales pero esto no mejoraba el accuracy, sin embargo esto se probó cuando aún no se concatenaban las señales de entrenamiento y validación para que la red neuronal entrenase con más datos, por lo cual es probable que si se hubiese implementado un filtrado pasabajo y se hubiesen tenido las señales concatenadas si se habría logrado una leve mejoría en el accuracy.