Inteligencia Artifical Avansada para la Ciencia de Datos I

Álvaro Morán Errejón — A01638034
Héctor Manuel Cárdenas Yáñez — A01634615
Isaí Ambrocio — A01625101
Siddhartha López Valenzuela — A00227694

September 4th, 2023

1 Introduction

This report intends to show a solution the problem of "Activivty Recognition in Senior Citizens". The Human Activity Recognition 70+ (HAR70+) dataset is a professionally-annotated dataset containing 18 fit-to-frail older-adult subjects (70-95 years old) wearing two 3-axial accelerometers for around 40 minutes during a semi-structured free-living protocol. The sensors were attached to the right thigh and lower back Each subject's recordings are provided in a separate .csv file. The purpose was to train machine learning models for human activity recognition on professionally-annotated accelerometer data of fit-to-frail older adults.

2 Libraries

A list of all the libraries used to perform this activity are shown in the image below:

3 Data Analysis

3.1 Loading Data

For staters, all the data was uploaded to Google Colab and was roughly concatenated into a huge DataFrame containing all the information so that it could be used for further analysis.

```
[2] df501 = pd.read_csv("/content/501.csv")
    df502 = pd.read_csv("/content/502.csv")
    df503 = pd.read_csv("/content/503.csv")
    df504 = pd.read_csv("/content/504.csv")
    df505 = pd.read_csv("/content/505.csv")
    df506 = pd.read_csv("/content/506.csv")
    df507 = pd.read_csv("/content/507.csv")
    df508 = pd.read_csv("/content/508.csv")
    df509 = pd.read_csv("/content/509.csv")
    df510 = pd.read_csv("/content/510.csv")
    df511 = pd.read_csv("/content/511.csv")
    df512 = pd.read_csv("/content/512.csv")
    df513 = pd.read_csv("/content/513.csv")
    df514 = pd.read_csv("/content/514.csv")
    df515 = pd.read_csv("/content/515.csv")

[3] dataframes = [df501, df502, df503, df504, df505, df506, df507, df508, df509, df510, df511, df512, df513,

[4] df = pd.concat(dataframes)
```

3.2 Exploratory Analysis

As you can see below we stated by simply undestanding the size of the sample, then print a summary of the data in a table and understanding the type of data in three: object, float and integer.

```
[5] df.size

    14660480

[6] df.describe()
```

	back_x	back_y	back_z	thigh_x	thigh_y	thigh_z	label
count	1.832560e+06	1.832560e+06	1.832560e+06	1.832560e+06	1.832560e+06	1.832560e+06	1.832560e+06
mean	-8.683814e-01	-3.178814e-02	2.244210e-02	-6.763959e-01	8.185066e-03	-3.858819e-01	3.940692e+00
std	2.756643e-01	1.556768e-01	4.279549e-01	5.596829e-01	2.707317e-01	5.087015e-01	2.912512e+00
min	-4.333252e+00	-2.031006e+00	-2.204834e+00	-7.942139e+00	-5.142578e+00	-7.593750e+00	1.000000e+00
25%	-9.909670e-01	-1.093750e-01	-2.692870e-01	-9.855960e-01	-1.132810e-01	-9.770510e-01	1.000000e+00
50%	-9.377440e-01	-1.855500e-02	-9.399400e-02	-9.357910e-01	-1.464800e-02	-1.906740e-01	3.000000e+00
75%	-8.344730e-01	5.761700e-02	3.078610e-01	-7.763700e-02	1.230470e-01	-3.174000e-03	7.000000e+00
max	3.630370e-01	1.576660e+00	1.179199e+00	3.395264e+00	5.725098e+00	3.953369e+00	8.000000e+00

```
[7] df.dtypes

    timestamp     object
    back_x       float64
    back_y       float64
    back_z       float64
    thigh_x      float64
    thigh_y      float64
    thigh_z      float64
    label          int64
    dtype: object
```

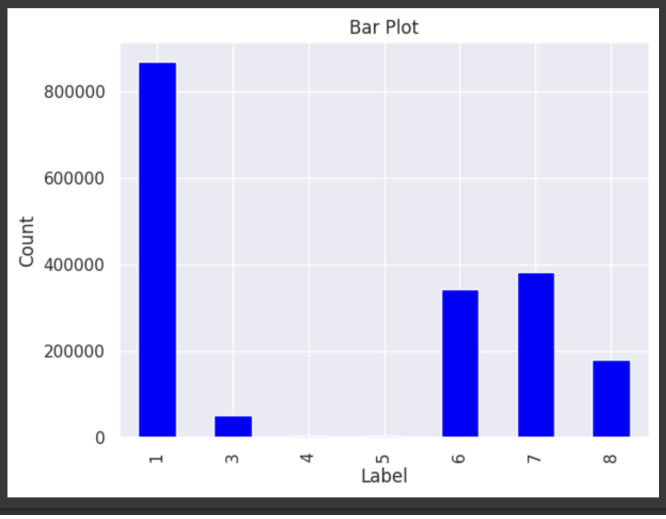
As we can see there is no missng information in any of the columns, which in turn will save us a step for creating the model. There we can also see that by uiing the function .unique() we are able to see the labels, which coincide with thoseis Kaggle. It is divided as follows

1. walking
3. shuffling
4. ascending stairs
5. descending stairs
6. standing
7. sitting
8. lying

```
[8] df.isnull().sum()

    timestamp    0
    back_x       0
    back_y       0
    back_z       0
    thigh_x      0
    thigh_y      0
    thigh_z      0
    label        0
    dtype: int64

[9] df['label'].unique()

    array([6, 3, 1, 7, 8, 5, 4])
```

It is also important to highlight that we manage to understand how the data is distributed within each label. As we can see, the data is unfairly distributed among the lables. Therefore we must implement a balancing method in order to advance with the perdiction method. Balancing data is crucial to prevent biased models, enhance generalization, ensure fairness and equity and, most importantly improve the performance metrics of the method. By addressing data imbalances we are making the model more accurate and equitable predictions, essentialy making it more reliable and useful in real-world applications while mitigating the negatve impact of class imbalances.

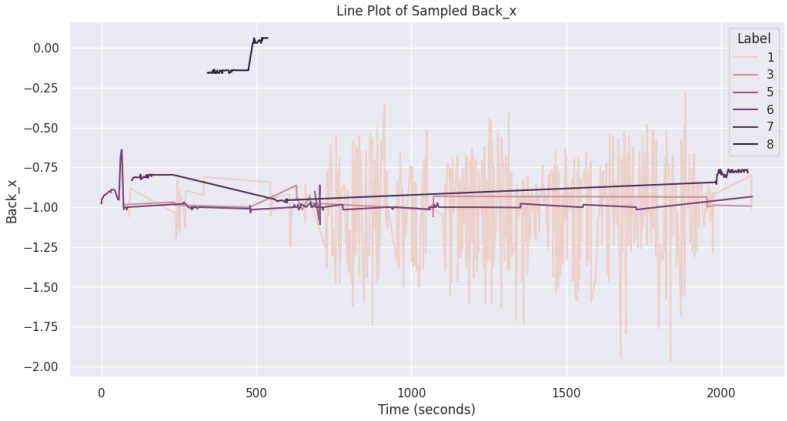
```
df['label'].value_counts().sort_index().plot(kind='bar', color='blue')
plt.xlabel('Label')
plt.ylabel('Count')
plt.title('Bar Plot')
plt.show()
```



Next we analized how the variables function in regard with the tieme. The code below graphs the data in labels and shows how is acts as the time goes by. It is worth noting that this is merely a small sample of the Data Frame, since running this code with the whole amount of data would take too long to display. However, we areable to see the how each individual label acts in comparison to the rest. The amouont of time and movemente clearly displays the relevance of some labels over others. Image:

```
[ ] ###SOLO CORRER ESTE CODIGO SI EL DATASET ESTA PEQUEÑO (QUE NO ESTEN LOS 15 CSV CONCATENADOS). DE LO CONT
    df['timestamp'] = pd.to_datetime(df['timestamp'])

    # muestra de datos usada
    sampled_df = df.sample(frac=0.01)  # Adjust the fraction as needed

    # cambiar a segundos el timestamp de inicio a fin
    sampled_df['timestamp_seconds'] = (sampled_df['timestamp'] - sampled_df['timestamp'].min()).dt.total_sec

    # line plots para cada columna
    variables_to_plot = ['back_x', 'back_y', 'back_z', 'thigh_x', 'thigh_y', 'thigh_z']

    plt.figure(figsize=(12, 6))

    for variable in variables_to_plot:
        plt.figure(figsize=(12, 6))
        sns.lineplot(data=sampled_df, x='timestamp_seconds', y=variable, hue='label')
        plt.xlabel('Time (seconds)')
        plt.ylabel(f'{variable.capitalize()}')
        plt.title(f'Line Plot of Sampled {variable.capitalize()}')
        plt.legend(title='Label')
        plt.show()
```

Image:


Line Plot of Sampled Back_x
Label
1
3
5
6
7
8
Back_x
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
−2.00
0
500
1000
1500
2000
Time (seconds)

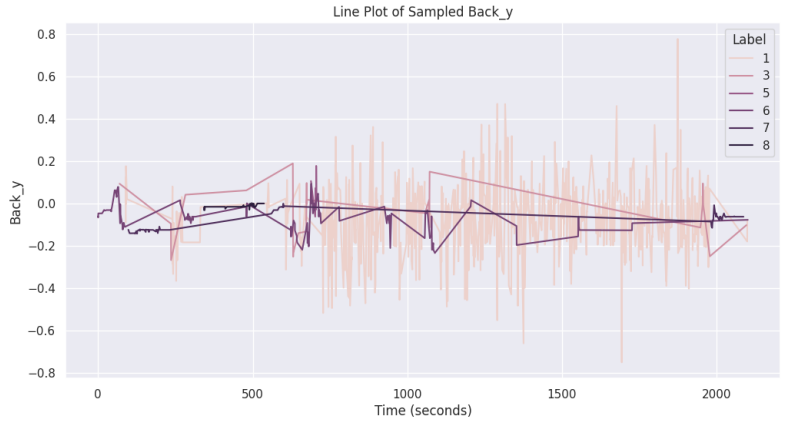
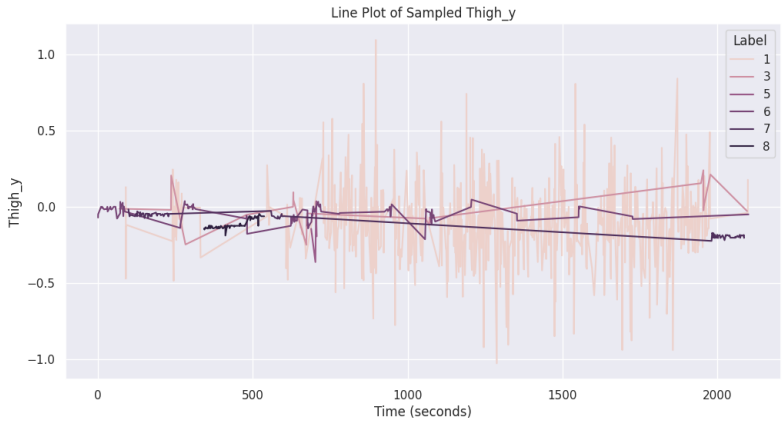

Line Plot of Sampled Back_y
Label
1
3
5
6
7
8
Back_y
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
500
1000
1500
2000
Time (seconds)

Image:


Line Plot of Sampled Back_z
Label
1
3
5
6
7
8
Back_z
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
500
1000
1500
2000
Time (seconds)
Line Plot of Sampled Thigh_x
Thigh_x
0
−1
−2
−3
−4
Time (seconds)

Image:

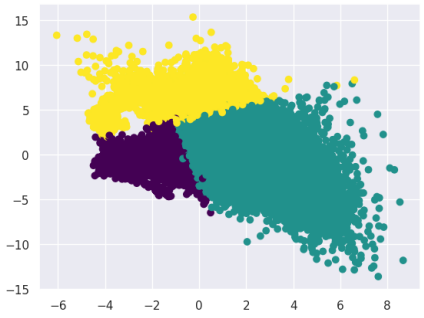


Finally, we performed a Data Clusering in order to further understand how the data works. As we can see the data is very close together, mostly divided in three mayor labels, just as we saw prevosly, however, it is worth noting the the data is closely tied together, there aren't any particular outliers that we need to take into consideration.

Data Clustering

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA


clusters = ['back_x', 'back_y', 'back_z', 'thigh_x', 'thigh_y', 'thigh_z']
df_clusters = df[clusters]
scaler = StandardScaler()
scaled_data = scaler.fit_transform(df_clusters)

# hacer el elbow method o cualquier otro metodo para determinar num_clusters, falta!!!
num_clusters = 3
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
cluster_labels = kmeans.fit_predict(scaled_data)

pca = PCA(n_components=2)
pca_data = pca.fit_transform(scaled_data)

plt.scatter(pca_data[:, 0], pca_data[:, 1], c=cluster_labels, cmap='viridis')
plt.show()
```

Image:



4 Data Processing

4.1 Unbalanced Analysis

First, we start by analyzing the model as it is, with the data unbalanced. The following code starts by setting up a stratified k-fold cross-validation, splitting the dataset into three folds and shuffling the data. It employs a linear Support Vector Classifier (SVC) for training and prediction. During each fold, it trains the model on a training subset and evaluates its performance on a validation subset, storing the true and predicted labels. After the cross-validation, it concatenates the true and predicted labels from all folds and generates a comprehensive classification report, presenting key classification metrics such as precision, recall, f1-score, and support for each class. This report provides a consolidated view of the model's performance across all folds, offering insights into its generalization capabilities and predictive accuracy on the given dataset.

```
[ ]  import random
     import numpy as np
     import matplotlib.pyplot as plt
     from sklearn.svm import SVC
     from sklearn.model_selection import StratifiedKFold
     from sklearn.metrics import classification_report

[ ]  #df.dropna(inplace=True)
     #df.isnull().sum()

[ ]  sampled_df = df.sample(frac=0.1)
     x = sampled_df.iloc[:, 0:6]
     y = sampled_df.iloc[:, 6]

[ ]  print("----- Imbalanced sample -----")
     kf = StratifiedKFold(n_splits=3, shuffle=True)
     clf = SVC(kernel='linear')
     cv_y_test = []
     cv_y_pred = []

     for train_index, test_index in kf.split(x, y):
         # Training phase
         x_train = x.iloc[train_index, :]
         y_train = y.iloc[train_index]
         clf.fit(x_train, y_train)

         # Test phase
         x_test = x.iloc[test_index, :]
         y_test = y.iloc[test_index]
         y_pred = clf.predict(x_test)

         cv_y_test.append(y_test)
         cv_y_pred.append(y_pred)

     print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

As we can see by the results, the model as it is, with unbalanced data, returns a poorly trained model. The precision for two labels is approximately 98 percent, which is very good. However, the model is deficient in predicting a precise answer for the remaining labels. Some even have 0 percent precision. This highlights the importance of creating a balancing method in order to create an optimized and accurate model.

```
----- Imbalanced sample -----
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           1       0.68      0.99      0.81     87078
           3       0.00      0.00      0.00      5036
           4       0.00      0.00      0.00       366
           5       0.00      0.00      0.00       447
           6       0.00      0.00      0.00     34266
           7       0.98      0.98      0.98     38249
           8       0.98      0.98      0.98     17814

    accuracy                           0.77    183256
   macro avg       0.38      0.42      0.40    183256
weighted avg       0.62      0.77      0.68    183256

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
  _warn_prf(average, modifier, msg_start, len(result))
```

4.2 Balancing Methods

The following code begins by randomly sampling 10 percent of the rows from an original dataset and segregates features and labels accordingly. It then focuses on addressing class imbalance in a classification task. The first approach, "Upsampling," involves resampling the minority classes to equal the majority class in terms of sample count. The code accomplishes this by duplicating samples from the minority class, achieving a more balanced dataset. The second strategy involves utilizing a weighted loss function in conjunction with the Support Vector Classifier (SVC). By setting the class weights to 'balanced,' the algorithm automatically adjusts these weights based on the class distribution in the training data, effectively mitigating the imbalanced class issue. Both strategies aim to improve classification performance by enhancing the model's ability to handle disparate class frequencies. The code concludes by presenting a detailed classification report, offering insights into the model's performance across various evaluation metrics.

```
    # Test phase
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred), zero_division='warn'))

##### balancear datos usando weighted funtion #####
print("----- Weighted loss function -----")

clf = SVC(kernel='linear', class_weight='balanced')
kf = StratifiedKFold(n_splits=5, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    # Training phase
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    clf.fit(x_train, y_train)

    # Test phase
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred),zero_division='warn'))
```

```
sampled_df = df.sample(frac=0.1)
x = sampled_df.iloc[:, 0:6]
y = sampled_df.iloc[:, 6]
import warnings  # Import the warnings module
warnings.filterwarnings("ignore")

##### balancear datos usando Upsampling #####
print("----- Upsampling -----")
clf = SVC(kernel='linear')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    # Training phase
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]

    # Find unique class labels in the current fold
    unique_classes = y_train.unique()

    # Calculate the number of samples in the majority class (assuming class 7 is the majority class)
    majority_class_label = 7
    n_majority = (y_train == majority_class_label).sum()

    # Upsample each minority class if there are at least two unique classes
    for class_label in unique_classes:
        if class_label != majority_class_label:
            x_class = x_train[y_train == class_label]
            y_class = y_train[y_train == class_label]
            n_class = len(y_class)

            if n_class > 0:
                # Randomly select with replacement from the minority class to match the majority class
                ind = random.choices(range(n_class), k=n_majority)
                x_sub = pd.concat([x_class.iloc[ind], x_train[y_train == majority_class_label]], axis=0)
                y_sub = pd.concat([y_class.iloc[ind], y_train[y_train == majority_class_label]], axis=0)

                clf.fit(x_sub, y_sub)
```

The result displays that although the methods did show some progress and optimized the unbalane defficiency, it's still not the best solution for dealing the problem. The precision stil wrong, having some labels with 0 precision.

```
----- Upsampling -----
              precision    recall  f1-score   support

           1       0.00      0.00      0.00     86928
           3       0.00      0.00      0.00      5060
           4       0.00      0.66      0.01       357
           5       0.00      0.33      0.01       468
           6       0.00      0.00      0.00     34418
           7       0.68      0.99      0.81     38138
           8       0.00      0.00      0.00     17887

    accuracy                           0.21    183256
   macro avg       0.10      0.28      0.12    183256
weighted avg       0.14      0.21      0.17    183256

----- Weighted loss function -----
              precision    recall  f1-score   support

           1       0.72      0.19      0.30     86928
           3       0.05      0.26      0.09      5060
           4       0.01      0.58      0.02       357
           5       0.01      0.33      0.02       468
           6       0.41      0.55      0.47     34418
           7       0.98      0.98      0.98     38138
           8       0.97      0.98      0.98     17887

    accuracy                           0.50    183256
   macro avg       0.45      0.55      0.41    183256
weighted avg       0.72      0.50      0.53    183256
```

4.3 Methods

The next code conducts a comprehensive evaluation of multiple machine learning classifiers (linear SVM, RBF SVM, KNN, Decision Tree, Linear Discriminant Analysis, Random Forest, Gradient Boosting, and Polynomial SVM) on the dataset. The dataset is initially sampled, and features and labels are extracted. Each classifier is trained and evaluated using 3-fold cross-validation. The evaluation involves training on a subset of the data and testing on another, followed by generating a classification report detailing precision, recall, F1-score, and support metrics for each class. The main goal is to assess and compare the performance of these classifiers on the dataset.

Linear SVM

```
sampled_df = df.sample(frac=0.5)
x = sampled_df.iloc[:, 0:6]
y = sampled_df.iloc[:, 6]

import warnings  # Import the warnings module
warnings.filterwarnings("ignore")


# Linear SVM
print('----- Linear-SVM -----')
kf = KFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []
print(x)
for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]  # Use DataFrame indexing
    y_train = y.iloc[train_index]  # Use DataFrame indexing
    x_test = x.iloc[test_index]    # Use DataFrame indexing
    y_test = y.iloc[test_index]    # Use DataFrame indexing

    clf = SVC(kernel='linear')
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

RBF SVM

```
# RBF SVM
print('----- RBF-SVM -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf = SVC(kernel='rbf')
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))

sampled_df = df.sample(frac=0.1)
x = sampled_df.iloc[:, 0:6]
y = sampled_df.iloc[:, 6]
```

KNN

```
# KNN
print('----- KNN -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf = KNeighborsClassifier(n_neighbors=3)
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

Decision Tree

```
# Decision tree
print('----- Decision tree -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf = DecisionTreeClassifier()
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

Linear Discriminant Analysis

```
# Linear Discriminant Analysis
print('----- Linear Discriminant Analysis -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf = LinearDiscriminantAnalysis()
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

Random Forest

```
# Random Forest
print('----- Random Forest -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf = RandomForestClassifier(n_estimators=100, random_state=42)
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

Gradient Boosting

```
# Gradient Boosting
print('----- Gradient Boosting -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf = GradientBoostingClassifier(n_estimators=100, random_state=42)
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

SVM with Kernel Polynomial

```python
# SVM con kernel polinómico
print('----- Polynomial SVM -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf = SVC(kernel='poly', degree=3)  # Ajusta el grado del polinomio según sea necesario
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

print(classification_report(np.concatenate(cv_y_test), np.concatenate(cv_y_pred)))
```

The result display that the best model is the Random Forest. It shows the best precision overall in all the labels. The worst being of 0.53 and the best of 1.00. However, the recall is still pretty deficient on labels: 3, 4, and 5. Linear

SVM -

```
----- Linear-SVM -----
          back_x    back_y    back_z   thigh_x   thigh_y   thigh_z
79265  -0.905762 -0.112305 -0.433105 -0.982422 -0.019531 -0.113281
69748  -0.942627  0.128662 -0.107910 -1.022705  0.048584 -0.207275
2614   -1.031006 -0.126465  0.007080 -0.003662  0.148682 -1.002686
92664  -0.871338 -0.260498 -0.416016 -0.924805 -0.051758 -0.216309
99071  -0.804688 -0.079834  0.515625 -0.205078 -0.199707 -1.044678
...          ...       ...       ...       ...       ...       ...
86756  -0.772217  0.018555  0.471680 -0.975098 -0.038086 -0.363525
98750  -0.899658  0.060791  0.409424  0.015869  0.154297 -0.979980
43300  -0.971191  0.022461 -0.167236 -1.028076  0.106201 -0.246338
133449 -0.895996  0.003418 -0.181396 -0.720459 -0.742188 -0.139160
54268  -0.949951 -0.156250  0.203125  0.070068 -0.054199 -0.989502

[91628 rows x 6 columns]
              precision    recall  f1-score   support

           1       0.68      0.99      0.81     43638
           3       0.00      0.00      0.00      2503
           4       0.00      0.00      0.00       181
           5       0.00      0.00      0.00       244
           6       0.00      0.00      0.00     17293
           7       0.98      0.98      0.98     18922
           8       0.98      0.97      0.98      8847

    accuracy                           0.77     91628
   macro avg       0.38      0.42      0.39     91628
weighted avg       0.62      0.77      0.68     91628
```

RBF-SVM and KNN -

```
----- RBF-SVM -----
              precision    recall  f1-score   support

           1       0.88      0.87      0.88     43638
           3       0.00      0.00      0.00      2503
           4       0.00      0.00      0.00       181
           5       0.00      0.00      0.00       244
           6       0.68      0.81      0.74     17293
           7       0.99      0.98      0.99     18922
           8       0.98      0.99      0.98      8847

    accuracy                           0.87     91628
   macro avg       0.50      0.52      0.51     91628
weighted avg       0.85      0.87      0.86     91628

----- KNN -----
              precision    recall  f1-score   support

           1       0.92      0.93      0.93     86915
           3       0.28      0.11      0.16      5026
           4       0.53      0.12      0.19       401
           5       0.46      0.08      0.14       486
           6       0.83      0.90      0.87     34282
           7       1.00      1.00      1.00     38275
           8       1.00      1.00      1.00     17871

    accuracy                           0.92    183256
   macro avg       0.72      0.59      0.61    183256
weighted avg       0.91      0.92      0.91    183256
```

Decision Tree and Linear Discriminant -

```
----- Decision tree -----
              precision    recall  f1-score   support

           1       0.91      0.90      0.90     86915
           3       0.14      0.16      0.15      5026
           4       0.10      0.11      0.11       401
           5       0.05      0.06      0.06       486
           6       0.81      0.82      0.82     34282
           7       1.00      1.00      1.00     38275
           8       1.00      1.00      1.00     17871

    accuracy                           0.89    183256
   macro avg       0.57      0.58      0.58    183256
weighted avg       0.89      0.89      0.89    183256

----- Linear Discriminant Analysis -----
              precision    recall  f1-score   support

           1       0.68      0.93      0.79     86915
           3       0.00      0.00      0.00      5026
           4       0.00      0.00      0.00       401
           5       0.00      0.00      0.00       486
           6       0.34      0.06      0.10     34282
           7       0.92      0.95      0.94     38275
           8       0.90      0.95      0.93     17871

    accuracy                           0.74    183256
   macro avg       0.41      0.41      0.39    183256
weighted avg       0.67      0.74      0.68    183256
```

Random Forest and Gradient Boosting -

```
----- Random Forest -----
              precision    recall  f1-score   support

           1       0.91      0.96      0.94     86915
           3       0.53      0.03      0.06      5026
           4       0.64      0.02      0.04       401
           5       0.62      0.01      0.02       486
           6       0.87      0.90      0.88     34282
           7       1.00      1.00      1.00     38275
           8       1.00      1.00      1.00     17871

    accuracy                           0.93    183256
   macro avg       0.80      0.56      0.56    183256
weighted avg       0.92      0.93      0.92    183256

----- Gradient Boosting -----
              precision    recall  f1-score   support

           1       0.91      0.94      0.92     86915
           3       0.44      0.00      0.00      5026
           4       0.03      0.01      0.01       401
           5       0.01      0.00      0.00       486
           6       0.81      0.88      0.84     34282
           7       1.00      1.00      1.00     38275
           8       1.00      1.00      1.00     17871

    accuracy                           0.92    183256
   macro avg       0.60      0.55      0.54    183256
weighted avg       0.90      0.92      0.90    183256
```

Polynomial SVM

```
----- Polynomial SVM -----
              precision    recall  f1-score   support

           1       0.83      0.88      0.86     86915
           3       0.00      0.00      0.00      5026
           4       0.00      0.00      0.00       401
           5       0.00      0.00      0.00       486
           6       0.66      0.67      0.67     34282
           7       0.99      0.99      0.99     38275
           8       0.99      0.99      0.99     17871

    accuracy                           0.85    183256
   macro avg       0.50      0.51      0.50    183256
weighted avg       0.82      0.85      0.83    183256
```

4.4 Random Forest and Balanced Random Forest Model

Considering the results of all the models, we decided to use two: Random Forest, which gave us the best precisión, and Balanced Random Forest, which displayed the highest recall from all the models. The code focuses on classification using these two methods. It begins by sampling the dataset and extracting features and labels. It then trains and evaluates a Random Forest classifier and a Balanced Random Forest classifier using 3-fold stratified cross-validation, presenting classification reports for both. Afterwards, it creates an ensemble using a Voting Classifier that combines both classifiers and evaluates the ensemble's performance using the same cross-validation approach, generating a classification report. The goal is to compare the individual and combined performances of Random Forest and Balanced Random Forest, exploring the potential benefits of an ensemble approach for classification tasks.Random Forest -

```
[ ] # Sampled data (adjust the fraction as needed)
    sampled_df = df.sample(frac=0.5)
    x = sampled_df.iloc[:, 0:6]
    y = sampled_df.iloc[:, 6]

    # Random Forest
    print('----- Random Forest -----')
    kf = StratifiedKFold(n_splits=3, shuffle=True)
    cv_y_test_rf = []
    cv_y_pred_rf = []

    for train_index, test_index in kf.split(x, y):
        x_train = x.iloc[train_index]
        y_train = y.iloc[train_index]
        x_test = x.iloc[test_index]
        y_test = y.iloc[test_index]

        clf_rf = RandomForestClassifier(n_estimators=100, random_state=42)
        clf_rf.fit(x_train, y_train)
        y_pred_rf = clf_rf.predict(x_test)

        cv_y_test_rf.append(y_test)
        cv_y_pred_rf.append(y_pred_rf)

    print("Random Forest Classification Report:")
    print(classification_report(np.concatenate(cv_y_test_rf), np.concatenate(cv_y_pred_rf)))
```

Balanced Random Forest -

```
# Balanced Random Forest
print('----- Balanced Random Forest -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test_brf = []
cv_y_pred_brf = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf_brf = BalancedRandomForestClassifier(n_estimators=100, random_state=42)
    clf_brf.fit(x_train, y_train)
    y_pred_brf = clf_brf.predict(x_test)

    cv_y_test_brf.append(y_test)
    cv_y_pred_brf.append(y_pred_brf)

print("Balanced Random Forest Classification Report:")
print(classification_report(np.concatenate(cv_y_test_brf), np.concatenate(cv_y_pred_brf)))
```

Ensemble both classifiers

```
# Ensemble both classifiers
voting_clf = VotingClassifier(estimators=[
    ('random_forest', clf_rf),
    ('balanced_random_forest', clf_brf)
], voting='soft')

print('----- Ensemble: Random Forest + Balanced Random Forest -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test_ensemble = []
cv_y_pred_ensemble = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    voting_clf.fit(x_train, y_train)
    y_pred_ensemble = voting_clf.predict(x_test)

    cv_y_test_ensemble.append(y_test)
    cv_y_pred_ensemble.append(y_pred_ensemble)

print("Ensemble Classification Report:")
print(classification_report(np.concatenate(cv_y_test_ensemble), np.concatenate(cv_y_pred_ensemble)))
```

As we can see both methods individually have great results for precision and recall respectively. By ensembling a method with both we get a new method in the middle. It doesn't have the precision from the Random Forest nor the recall of the Balanced Random Forest, instead it has an in between.

Random Forest and Balanced Random Forest Results -

```
----- Random Forest -----
Random Forest Classification Report:
              precision    recall  f1-score   support

           1       0.92      0.97      0.94    434767
           3       0.62      0.06      0.12     25457
           4       0.79      0.08      0.14      1909
           5       0.74      0.03      0.06      2238
           6       0.89      0.90      0.90    171852
           7       1.00      1.00      1.00    190835
           8       1.00      1.00      1.00     89222

    accuracy                           0.94    916280
   macro avg       0.85      0.58      0.59    916280
weighted avg       0.93      0.94      0.92    916280

----- Balanced Random Forest -----
Balanced Random Forest Classification Report:
              precision    recall  f1-score   support

           1       0.98      0.61      0.75    434767
           3       0.14      0.58      0.23     25457
           4       0.03      0.72      0.06      1909
           5       0.02      0.58      0.04      2238
           6       0.87      0.83      0.85    171852
           7       1.00      1.00      1.00    190835
           8       1.00      1.00      1.00     89222

    accuracy                           0.77    916280
   macro avg       0.58      0.76      0.56    916280
weighted avg       0.94      0.77      0.83    916280
```

Random Forest + Balanced Random Forest Results

```
----- Ensemble: Random Forest + Balanced Random Forest -----
Ensemble Classification Report:
              precision    recall  f1-score   support

           1       0.94      0.94      0.94    434767
           3       0.32      0.23      0.26     25457
           4       0.30      0.31      0.31      1909
           5       0.31      0.19      0.24      2238
           6       0.87      0.91      0.89    171852
           7       1.00      1.00      1.00    190835
           8       1.00      1.00      1.00     89222

    accuracy                           0.93    916280
   macro avg       0.68      0.65      0.66    916280
weighted avg       0.92      0.93      0.93    916280
```

4.5 Optimal Selection of Hyperparameters

Next, we decided to search for the optimal number of hyperparameters. The following code conducts a comprehensive evaluation of the same machine learning classifiers (Random Forest, Balanced Random Forest, and an ensemble of both using a Voting Classifier) just as before. Then the hyperparameters are optimized through randomized search. The ensemble is evaluated using the best hyperparameters and another classification report is generated. This process enables a thorough assessment of individual classifiers and an ensemble, demonstrating how hyperparameter tuning can enhance the performance of an ensemble combining diverse classifiers.Random Forest -

```
sampled_df = df.sample(frac=0.5)
x = sampled_df.iloc[:, 0:6]
y = sampled_df.iloc[:, 6]


# Random Forest
print('----- Random Forest -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test_rf = []
cv_y_pred_rf = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf_rf = RandomForestClassifier(n_estimators=100, random_state=42)
    clf_rf.fit(x_train, y_train)
    y_pred_rf = clf_rf.predict(x_test)

    cv_y_test_rf.append(y_test)
    cv_y_pred_rf.append(y_pred_rf)

print("Random Forest Classification Report:")
print(classification_report(np.concatenate(cv_y_test_rf), np.concatenate(cv_y_pred_rf)))
```

Balanced Random Forest -

```
# Balanced Random Forest
print('----- Balanced Random Forest -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test_brf = []
cv_y_pred_brf = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    clf_brf = BalancedRandomForestClassifier(n_estimators=100, random_state=42)
    clf_brf.fit(x_train, y_train)
    y_pred_brf = clf_brf.predict(x_test)

    cv_y_test_brf.append(y_test)
    cv_y_pred_brf.append(y_pred_brf)

print("Balanced Random Forest Classification Report:")
print(classification_report(np.concatenate(cv_y_test_brf), np.concatenate(cv_y_pred_brf)))
```

Hyperparameters part 1 -

```
# Ensemble both classifiers
voting_clf = VotingClassifier(estimators=[
    ('random_forest', clf_rf),
    ('balanced_random_forest', clf_brf)
], voting='soft')  # You can use 'soft' voting for probability-based ensemble

# Suppress warnings using a context manager
with warnings.catch_warnings():
    warnings.filterwarnings("ignore")  # Ignore all warnings within this context

    # Define the hyperparameter grid for tuning
    param_dist = {
        'random_forest__n_estimators': [50, 100, 150],
        'balanced_random_forest__n_estimators': [50, 100, 150],
        'balanced_random_forest__sampling_strategy': ['auto', 0.5, 0.75],
    }

    # Perform randomized search for hyperparameter tuning
    random_search = RandomizedSearchCV(estimator=voting_clf, param_distributions=param_dist, n_iter=10, cv=5, scoring='accuracy', random_sta
    random_search.fit(x, y)

# Get the best hyperparameters
best_params = random_search.best_params_
print("Best Hyperparameters:", best_params)
best_random_forest_estimators = best_params['random_forest__n_estimators']
best_balanced_rf_estimators = best_params['balanced_random_forest__n_estimators']
best_sampling_strategy = best_params['balanced_random_forest__sampling_strategy']

# Create the ensemble model with the best hyperparameters
best_voting_clf = VotingClassifier(estimators=[
    ('random_forest', RandomForestClassifier(n_estimators=best_random_forest_estimators)),
    ('balanced_random_forest', BalancedRandomForestClassifier(n_estimators=best_balanced_rf_estimators, sampling_strategy=best_sampling_stra
], voting='soft')  # You can use 'soft' voting for probability-based ensemble
```

Hyperparameters part 2

```
# Fit the ensemble model with the best hyperparameters
best_voting_clf.fit(x, y)

print('----- Ensemble with Best Hyperparameters -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)
cv_y_test_best_ensemble = []
cv_y_pred_best_ensemble = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    best_voting_clf.fit(x_train, y_train)
    y_pred_best_ensemble = best_voting_clf.predict(x_test)

    cv_y_test_best_ensemble.append(y_test)
    cv_y_pred_best_ensemble.append(y_pred_best_ensemble)

print("Ensemble Classification Report with Best Hyperparameters:")
print(classification_report(np.concatenate(cv_y_test_best_ensemble), np.concatenate(cv_y_pred_best_ensemble)))
```

By getting the optimal number of hyperparameter we can now test the model and models results for precision, recall and f1

Random Forest and Balanced Random Forest Results -

```
----- Random Forest -----
Random Forest Classification Report:
              precision    recall  f1-score   support

           1       0.92      0.97      0.94    434901
           3       0.59      0.06      0.11     25245
           4       0.81      0.07      0.14      1804
           5       0.74      0.04      0.07      2257
           6       0.89      0.90      0.90    171418
           7       1.00      1.00      1.00    191203
           8       1.00      1.00      1.00     89452

    accuracy                           0.94    916280
   macro avg       0.85      0.58      0.59    916280
weighted avg       0.93      0.94      0.93    916280

----- Balanced Random Forest -----
Balanced Random Forest Classification Report:
              precision    recall  f1-score   support

           1       0.98      0.60      0.75    434901
           3       0.14      0.58      0.23     25245
           4       0.03      0.73      0.06      1804
           5       0.02      0.58      0.04      2257
           6       0.87      0.83      0.85    171418
           7       1.00      1.00      1.00    191203
           8       1.00      1.00      1.00     89452

    accuracy                           0.77    916280
   macro avg       0.58      0.76      0.56    916280
weighted avg       0.94      0.77      0.83    916280
```

Hyperparameters

```
Best Hyperparameters: {'random_forest__n_estimators': 150, 'balanced_random_forest__sampling_strategy': '
----- Ensemble with Best Hyperparameters -----
Ensemble Classification Report with Best Hyperparameters:
              precision    recall  f1-score   support

           1       0.94      0.94      0.94    434901
           3       0.32      0.23      0.27     25245
           4       0.29      0.34      0.31      1804
           5       0.32      0.17      0.23      2257
           6       0.87      0.91      0.89    171418
           7       1.00      1.00      1.00    191203
           8       1.00      1.00      1.00     89452

    accuracy                           0.93    916280
   macro avg       0.68      0.66      0.66    916280
weighted avg       0.92      0.93      0.93    916280
```

4.6 Model tuning with hyperparameters

Lastly, we adjust the method with the hyperparameter from the previous example. This should allow for the optimization of the model's performance metrics like accuracy, precision, recall, and F1-score.Model with Hyperparameters

```
import random
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
from sklearn.metrics import classification_report
from imblearn.ensemble import BalancedRandomForestClassifier
from sklearn.ensemble import VotingClassifier
import warnings  # Import the warnings module
warnings.filterwarnings("ignore")


x = df.iloc[:, 0:6]
y = df.iloc[:, 6]

# Define the hyperparameters for the ensemble model
best_random_forest_estimators = 150  #
best_balanced_rf_estimators = 100  #
best_sampling_strategy = 'auto'  #

# Ensemble both classifiers with the defined hyperparameters
voting_clf = VotingClassifier(estimators=[
    ('random_forest', RandomForestClassifier(n_estimators=best_random_forest_estimators)),
    ('balanced_random_forest', BalancedRandomForestClassifier(n_estimators=best_balanced_rf_estimators, s
], voting='soft')  # You can use 'soft' voting for probability-based ensemble

# Fit the ensemble model with the defined hyperparameters using the entire dataset
voting_clf.fit(x, y)

print('----- Ensemble with Defined Hyperparameters using All Data -----')
kf = StratifiedKFold(n_splits=3, shuffle=True)

# Perform cross-validation and generate predictions
cv_y_test_ensemble = []
cv_y_pred_ensemble = []

for train_index, test_index in kf.split(x, y):
    x_train = x.iloc[train_index]
    y_train = y.iloc[train_index]
    x_test = x.iloc[test_index]
    y_test = y.iloc[test_index]

    voting_clf.fit(x_train, y_train)
    y_pred_ensemble = voting_clf.predict(x_test)

    cv_y_test_ensemble.append(y_test)
    cv_y_pred_ensemble.append(y_pred_ensemble)

# Concatenate the results from all folds
cv_y_test_all_folds = np.concatenate(cv_y_test_ensemble)
cv_y_pred_all_folds = np.concatenate(cv_y_pred_ensemble)

# Generate a classification report for cross-validation results
print("Ensemble Classification Report with Defined Hyperparameters (Cross-Validation):")
print(classification_report(cv_y_test_all_folds, cv_y_pred_all_folds))
```

The result display that the model has improved greatly from its initial predictions. Now, the precision has gotten exponentially better in the most unbalanced labels while also improving on the models recall. This means that the models prediction will increase and it's a more oprimize and funcional modelResults

```
Ensemble Classification Report with Defined Hyperparameters (Entire Dataset):
              precision    recall  f1-score   support

           1       0.98      0.97      0.98    869690
           3       0.78      0.72      0.75     50892
           4       0.70      0.80      0.75      3726
           5       0.80      0.74      0.77      4522
           6       0.94      0.96      0.95    343198
           7       1.00      1.00      1.00    381770
           8       1.00      1.00      1.00    178762

    accuracy                           0.97   1832560
   macro avg       0.89      0.88      0.88   1832560
weighted avg       0.97      0.97      0.97   1832560
```

5 Flask

The following code saves the model in a json, so that it can be passed to a Flask server with an interface to run the program and make predictions.

```
[ ] from flask import Flask, request, jsonify
    import joblib

    app = Flask(__name__)

    # Cargar el modelo de clasificación previamente entrenado
    modelo = joblib.load('modelo_clasificacion.pkl')

    @app.route('/')
    def hello_world():
        return '¡Servidor de clasificación en funcionamiento!'

    @app.route('/predict', methods=['POST'])
    def predict():
        try:
            # Obtener los datos de entrada del usuario
            data = request.get_json()

            # Realizar una predicción con el modelo cargado
            input_features = data['features']
            prediction = modelo.predict([input_features])

            # Devolver la predicción como respuesta JSON
            response = {'prediction': prediction.tolist()}
            return jsonify(response)

        except Exception as e:
            return jsonify({'error': str(e)})

    if __name__ == '__main__':
        app.run(debug=True)
```