notebook

December 22, 2024

1 Proyecto de Investigación

1.1 Pablo Chantada Saborido (pablo.chantada@udc.es)

1.2 Claudia Vidal Otero (claudia.votero@udc.es)

1.3 Aldana Smyna Medina Lostaunau (aldana.medina@udc.es)

2 Preparación de los datos

Antes de abordar el problema tenemos que entender con que tipo de datos estamos tratando, para ello realizamos un leve estudio del nuestro conjunto de datos. El *dataset* que utilizaremos está diseñado para clasificar seis actividades de la vida diaria realizadas por 30 individuos. Estas actividades son: **WALKING, WALKINGUPSTAIRS, WALKINGDOWNSTAIRS, SITTING, STANDING, LAYING**.

2.0.1 Información general del *dataset*

- **Número de ejemplos:** 10,299.
- **Número de variables:** 561.
- **Tipos de datos:**
  - **Enteros:** identificadores.
  - **Reales:** Medidas numéricas.
  - **Categóricos:** Etiquetas de las actividades realizadas.
- **Valores nulos:** El *dataset* incluye valores faltantes que deberán manejarse.

Descripción de las variables

1. **Identificador del sujeto:** Variable entera que indica el participante que realizó la actividad.
2. **Mediciones inerciales:**
   - **Aceleración triaxial (x, y, z):** Capturada por el acelerómetro, tanto la aceleración total como la componente filtrada (movimiento del cuerpo).

- **Velocidad angular triaxial (x, y, z):** Capturada por el giroscopio.

3. **Características derivadas:**
   - Calculadas a partir de ventanas deslizantes de las señales.

   - Variables en el dominio del tiempo y la frecuencia, que incluyen medidas estadísticas como medias, desviaciones estándar y correlaciones entre los ejes.

4. **Etiqueta de la actividad:** Variable categórica que representa la actividad realizada (*WALKING*, *SITTING*, etc.).

[ ]:
```
ENV["PYTHONWARNINGS"] = "ignore"
using PyCall
@pyimport warnings
warnings.filterwarnings("ignore")
using ScikitLearn
using ScikitLearn: fit!
using DataFrames
using CSV
using Random
using Plots
using DataFrames
using Statistics
using Test
using Distributions
using MLJ: predict
using Printf
using StatsBase: countmap
using Statistics: mean, std

@sk_import feature_selection: (mutual_info_classif,)
@sk_import linear_model: (LogisticRegression,)
@sk_import preprocessing: (LabelEncoder,)
@sk_import decomposition: (PCA, FastICA)
@sk_import discriminant_analysis: (LinearDiscriminantAnalysis,)
@sk_import manifold: (Isomap, LocallyLinearEmbedding)
@sk_import neural_network: MLPClassifier
@sk_import neighbors: KNeighborsClassifier
@sk_import svm: SVC
@sk_import metrics: f1_score
@sk_import ensemble: BaggingClassifier
@sk_import ensemble: AdaBoostClassifier
@sk_import ensemble: GradientBoostingClassifier
Random.seed!(172)
```

```
 Info: Running `conda install -y -c anaconda conda` in root environment
 @ Conda /home/clown/.julia/packages/Conda/zReqD/src/Conda.jl:181

Channels:
```

```
 - anaconda
 - conda-forge
Platform: linux-64
Collecting package metadata (repodata.json): …working… done
Solving environment: …working… done

## Package Plan ##

  environment location: /home/clown/.julia/conda/3/x86_64

  added / updated specs:
    - conda


The following packages will be SUPERSEDED by a higher-priority channel:

  certifi            conda-forge/noarch::certifi-2024.12.1~ -->
anaconda/linux-64::certifi-2024.12.14-py312h06a4308_0



Downloading and Extracting Packages: …working… done
Preparing transaction: done
Verifying transaction: done
Executing transaction: done

  Info: Running `conda install -y -c conda-forge 'libstdcxx-ng>=3.4,<13.0'` in
root environment
  @ Conda /home/clown/.julia/packages/Conda/zReqD/src/Conda.jl:181

Channels:
 - conda-forge
 - anaconda
Platform: linux-64
Collecting package metadata (repodata.json): …working… done
Solving environment: …working… done

## Package Plan ##

  environment location: /home/clown/.julia/conda/3/x86_64

  added / updated specs:
    - libstdcxx-ng[version='>=3.4,<13.0']


The following packages will be SUPERSEDED by a higher-priority channel:

  certifi            anaconda/linux-64::certifi-2024.12.14~ --> conda-
forge/noarch::certifi-2024.12.14-pyhd8ed1ab_0
```

```
Downloading and Extracting Packages: …working… done
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

```
TaskLocalRNG()
```

3 Análisis Inicial del Dataset

Para asegurar una correcta carga y comprensión de nuestros datos, hemos implementado tres funciones principales de análisis. Estas funciones nos permiten verificar la integridad de los datos y obtener una visión general de su estructura y distribución.

3.0.1 Funciones de Análisis

1. `loadDataset()`
   - Función encargada de cargar el archivo CSV en memoria
2. `csvDescription(csv)`
   - Genera un resumen del dataset
   - Proporciona información sobre:
     - Número de variables
     - Número total de instancias
     - Número de individuos únicos
     - Distribución de las clases
3. `getNullValues(data)`
   - Analiza la presencia de valores nulos en el dataset
   - Calcula y muestra el porcentaje de valores faltantes

3.0.2 Resultados del Análisis Inicial

Los resultados de la ejecución de estas funciones nos muestran que el DataSet se corresponde con lo indicado en el enunciado. La distribución de las clases muestra un balance relativamente equilibrado entre las diferentes actividades:

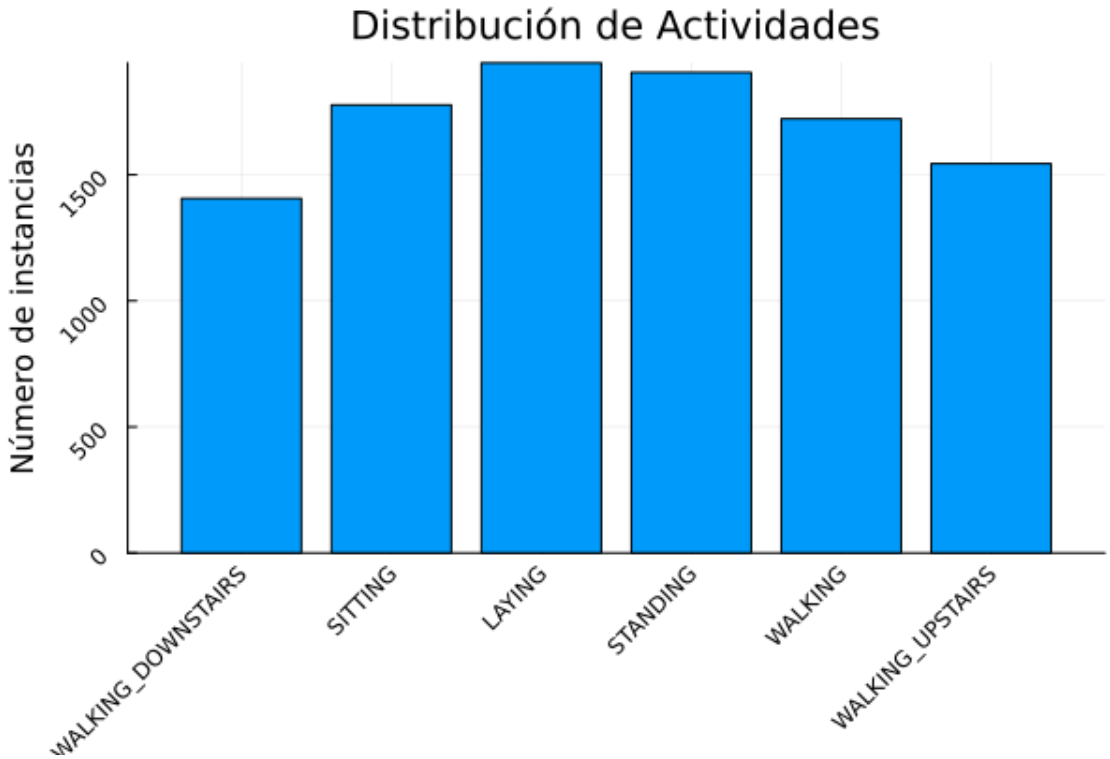
- WALKING_DOWNSTAIRS: 1406 instancias (13.65%)
- SITTING: 1777 instancias (17.25%)
- LAYING: 1944 instancias (18.88%)
- STANDING: 1906 instancias (18.51%)
- WALKING: 1722 instancias (16.72%)
- WALKING_UPSTAIRS: 1544 instancias (14.99%)

Esta distribución relativamente equilibrada de las clases es beneficiosa para el entrenamiento de nuestros modelos, ya que reduce el riesgo de sesgos en el aprendizaje debido a clases desproporcionadas.

[2]:
```
using DataFrames
using CSV
using StatsBase: countmap
using Plots

function loadDataset(PATH::String="../proyecto_investigacion/
 ↪Datos_Práctica_Evaluación_1.csv"; nrows::Union{Int, Nothing}=nothing)
    if !isfile(PATH)
        throw(ArgumentError("El archivo no existe en la ruta especificada"))
    end

    try
        # Permitimos cambiar las rows para agilizar las pruebas
        if isnothing(nrows)
            return CSV.read(PATH, DataFrame)
        else
            return CSV.read(PATH, DataFrame, limit=nrows)
        end
    catch e
        error("Error al leer el archivo CSV: $e")
    end
end

function csvDescription(df::DataFrame)
    println("===  Descrición del Dataset ===")
    println("Variables (sin subject & Activity): ", size(df, 2) - 2)  # Numero␣
 ↪de columnas
    println("Instancias Totales: ", size(df, 1))                      # Numero␣
 ↪de filas
    println("Individuos: ", length(unique(df[:, 1])))

    # Distribución de clases
    class_dist = countmap(df[:, end])
    activities = collect(keys(class_dist))
    counts = collect(values(class_dist))

    println("\nDistribción de Clases:")
    for (class, count) in class_dist
        percentage = round(count/size(df,1)*100, digits=2)
        println("  $class: $count instancias ($percentage%)")
    end

    # Imprimimos el grafico de las distribuciones
    p = bar(activities,
            counts,
            title="Distribución de Actividades",
            xlabel="Actividad",
```

```
            ylabel="Número de instancias",
            rotation=45,
            legend=false)

    return p
end

# Cargar y mostrar los datos
dataset = loadDataset()  # El mínimo para que tengamos 1 conjunto de test es␣
 ↪5000
csvDescription(dataset)
```

```
===  Descrición del Dataset ===
Variables (sin subject & Activity): 561
Instancias Totales: 10299
Individuos: 30

Distribción de Clases:
  WALKING_DOWNSTAIRS: 1406 instancias (13.65%)
  SITTING: 1777 instancias (17.25%)
  LAYING: 1944 instancias (18.88%)
  STANDING: 1906 instancias (18.51%)
  WALKING: 1722 instancias (16.72%)
  WALKING_UPSTAIRS: 1544 instancias (14.99%)
```



[3]:
```
function getNullValues(data::DataFrame)
    println("=== Análisis de Valores Nulos ===")
    println("Total columnas: ", ncol(data))
    println("Total filas: ", nrow(data))

    # DataFrame con el análisis de nulos
    null_analysis = DataFrame(
        Columna = String[],
        Nulos = Int[],
        Porcentaje = Float64[]
    )
    total_nulls = 0

    # Analizar cada columna
    for col in names(data)
        n_nulls = sum(ismissing.(data[:, col]))
        total_nulls += n_nulls

        if n_nulls > 0
            # Cargamos la columna, numero de nulos y el porcentaje
            push!(null_analysis, (
                col,
                n_nulls,
                (n_nulls / nrow(data)) * 100
            ))
        end
    end

    println("\nColumnas con valores nulos:")
    # Calcular y mostrar estadísticas globales
    if !isempty(null_analysis)
        total_elements = nrow(data) * ncol(data)
        total_pct_nulls = (total_nulls / total_elements) * 100
        println("\n=== Resumen Global ===")
        println("Total de valores nulos: $total_nulls")
        println("Porcentaje global de nulos: $(round(total_pct_nulls,␣
↪digits=5))%")
        println("Columnas afectadas: $(nrow(null_analysis)) de $(ncol(data))")

        return null_analysis
    else
        return "\n No se encontraron valores nulos en ninguna columna"
    end

end
```

```
# Ejecutamos sin ";" para que se vean las columnas que tienen nulos
getNullValues(dataset)
```

```
=== Análisis de Valores Nulos ===
Total columnas: 563
Total filas: 10299

Columnas con valores nulos:

=== Resumen Global ===
Total de valores nulos: 270
Porcentaje global de nulos: 0.00466%
Columnas afectadas: 131 de 563
```

	Columna	Nulos	Porcentaje
	String	Int64	Float64
1	tBodyAcc-std()-X	2	0.0194194
2	tBodyAcc-std()-Z	2	0.0194194
3	tBodyAcc-mad()-X	2	0.0194194
4	tBodyAcc-mad()-Y	3	0.029129
5	tBodyAcc-mad()-Z	3	0.029129
6	tBodyAcc-max()-X	2	0.0194194
7	tBodyAcc-max()-Z	1	0.00970968
8	tBodyAcc-sma()	1	0.00970968
9	tBodyAcc-iqr()-X	3	0.029129
10	tBodyAcc-iqr()-Y	1	0.00970968
11	tBodyAccJerk-std()-X	1	0.00970968
12	tBodyAccJerk-mad()-X	4	0.0388387
13	tBodyAccJerk-mad()-Y	4	0.0388387
14	tBodyAccJerk-mad()-Z	2	0.0194194
15	tBodyAccJerk-max()-X	1	0.00970968
16	tBodyAccJerk-sma()	3	0.029129
17	tBodyAccJerk-energy()-X	1	0.00970968
18	tBodyAccJerk-energy()-Y	1	0.00970968
19	tBodyAccJerk-iqr()-Y	3	0.029129
20	tBodyAccJerk-iqr()-Z	2	0.0194194
21	tBodyAccJerk-entropy()-X	2	0.0194194
22	tBodyAccJerk-entropy()-Y	1	0.00970968
23	tBodyAccJerk-entropy()-Z	1	0.00970968
24	tBodyGyro-mad()-Y	2	0.0194194
...	...	...	...

4 Preprocesado de los Datos

El preprocesado de nuestros datos se realiza a través de cuatro etapas fundamentales, implementadas de manera que se evite cualquier tipo de *data leakage* y se mantenga la integridad de los datos.

4.1 Etapas del Preprocesado

4.1.1 1. División Train/Test (HoldOut)

La primera etapa implementa una división del 10% de los individuos para test que se separa totalmente hasta el final, siguiendo un enfoque *individual-wise*. Esto significa que: - Todas las muestras de un mismo sujeto permanecen en el mismo conjunto - La división se realiza de manera aleatoria pero reproducible para la primera ejecución mediante una semilla

4.1.2 2. Preparación de Datos

Esta etapa realiza dos transformaciones esenciales utilizando únicamente el conjunto de entrenamiento: - **Codificación de actividades**: Transforma las etiquetas categóricas en valores numéricos - **Tratamiento de valores nulos**: Utiliza interpolación basada en las medias del conjunto de entrenamiento para evitar *data leakage*. - Las transformaciones se aplican posteriormente al conjunto de test para mantener la consistencia

4.1.3 3. Creación de Folds para Cross-Validation

Implementa una validación cruzada de 5 folds: - Mantiene la integridad de los individuos (*individual-wise*) - Distribuye los individuos de manera equilibrada entre los folds

4.1.4 4. Normalización MinMaxScaler

Aplica la normalización a las variables numéricas: - Excluye el identificador del sujeto ('subject') y la variable objetivo ('Activity') - Calcula los parámetros de normalización usando solo el conjunto de entrenamiento - Aplica la transformación a los conjuntos

4.2 Pipeline de Preprocesado

Para facilitar la experimentación, se ha implementado un pipeline que integra todas las etapas anteriores en una única función.

[4]:
```
function HoldOut(data::DataFrame)

    # Obtener individuos únicos
    individuals = unique(data.subject)
    n_individuals = length(individuals)
    test_size = floor(Int, 0.1 * n_individuals)  # 10% Separado completo para␣
 ↪test

    # Seleccionar individuos de test aleatoriamente
    test_individuals = sort(shuffle(individuals)[1:test_size])

    # Realizar la división
    test = data[in.(data.subject, Ref(Set(test_individuals))), :]
    train = data[.!in.(data.subject, Ref(Set(test_individuals))), :]

    # Mostrar información sobre la división
    println("\n=== Hold-Out Split Information ===")
```

```
    println("Individuos Totales: ", n_individuals)
    println("Individuos Train: ", n_individuals - test_size)
    println("Individuos Test : ", test_size)
    println("Individuos en test set: ", test_individuals)
    println("\nDistribución:")
    println("Train set: ", size(train, 1), " instancias (", round(size(train,␣
↪1)/size(data, 1)*100, digits=2), "%)")
    println("Test set: ", size(test, 1), " instancias (", round(size(test, 1)/
↪size(data, 1)*100, digits=2), "%)")

    # Verificar que no hay solapamiento de individuos para que sea␣
↪"individual-wise"
    @assert isempty(intersect(Set(train.subject), Set(test.subject))) "Error:␣
↪Found overlapping individuals between train and test sets"

    return train, test
end
```

```
HoldOut (generic function with 1 method)
```

[5]:
```
function prepareData!(train::DataFrame, test::DataFrame)

    println("\n=== Valores Nulos Iniciales ===")
    train_initial_nulls = sum(col -> count(ismissing, col), eachcol(train))
    test_initial_nulls = sum(col -> count(ismissing, col), eachcol(test))
    println("Train nulos iniciales: ", train_initial_nulls)
    println("Test nulos iniciales: ", test_initial_nulls)

    # Codificar variable objetivo
    activities = unique(train.Activity)
    Activity_dict = Dict(act => i for (i, act) in enumerate(activities))

    # Transformar la columna Activity
    train.Activity = map(x -> Activity_dict[x], train.Activity)
    test.Activity = map(x -> Activity_dict[x], test.Activity)

    # Rellenar valores nulos con la media de train
    null_cols_fixed = 0
    for col in names(train)
        train_nulls = count(ismissing, train[:, col])
        test_nulls = count(ismissing, test[:, col])

        if train_nulls > 0 || test_nulls > 0
            null_cols_fixed += 1
            train_mean = mean(skipmissing(train[:, col]))
            train[!, col] = coalesce.(train[:, col], train_mean)
            test[!, col] = coalesce.(test[:, col], train_mean)
```

```
        end
    end

    # Hacemos un check para comprobar si la eliminacion funciona
    println("\n=== Check Final ===")
    println("Número de columnas arregladas: ", null_cols_fixed)

    final_train_nulls = sum(col -> count(ismissing, col), eachcol(train))
    final_test_nulls = sum(col -> count(ismissing, col), eachcol(test))

    println("Valores nulos en Train: ", final_train_nulls)
    println("Valores nulos en Test: ", final_test_nulls)

    if final_train_nulls > 0 || final_test_nulls > 0
        @warn "There are still null values present after cleaning"
        for col in names(train)
            train_nulls = count(ismissing, train[:, col])
            test_nulls = count(ismissing, test[:, col])
            if train_nulls > 0
                println("Train - Columna $col tiene $train_nulls nulls")
            end
            if test_nulls > 0
                println("Test - Columna $col tiene $test_nulls nulls")
            end
        end
    else
        println("Dataset limpio, no contiene nulos!")
    end

    return train, test, Activity_dict
end

# Hacemos una ejecución aqui para ver que funciona correctamente
train_data, test_data = HoldOut(dataset)
train_prepared, test_prepared, activity_dict = prepareData!(train_data,␣
 ↪test_data);
```

```
=== Hold-Out Split Information ===
Individuos Totales: 30
Individuos Train: 27
Individuos Test : 3
Individuos en test set: [1, 9, 16]

Distribución:
Train set: 9298 instancias (90.28%)
Test set: 1001 instancias (9.72%)
```

```
=== Valores Nulos Iniciales ===
Train nulos iniciales: 237
Test nulos iniciales: 33

=== Check Final ===
Número de columnas arregladas: 131
Valores nulos en Train: 0
Valores nulos en Test: 0
Dataset limpio, no contiene nulos!
```

[6]:
```julia
function createFolds(data::DataFrame, n_folds=5)

    # Obtener individuos únicos del conjunto de train
    individuals = unique(data.subject)
    n_individuals = length(individuals)

    # Distribuir los individuos de manera equilibrada
    shuffled_individuals = shuffle(individuals)
    base_size = floor(Int, n_individuals/n_folds)
    extras = n_individuals % n_folds

    folds = Vector{Tuple{DataFrame,DataFrame}}()
    current_idx = 1

    for i in 1:n_folds
        # Calcular tamaño de este fold
        fold_size = base_size + (i <= extras ? 1 : 0)
        val_individuals = shuffled_individuals[current_idx:
 ↪current_idx+fold_size-1]
        current_idx += fold_size

        # Crear fold de validación
        val_fold = data[in.(data.subject, Ref(Set(val_individuals))), :]
        # Crear fold de entrenamiento con el resto
        train_fold = data[.!in.(data.subject, Ref(Set(val_individuals))), :]

        push!(folds, (train_fold, val_fold))
        println("Fold $i:")
        println("  Train: $(length(unique(train_fold.subject))) individuos,␣
 ↪$(size(train_fold,1)) instancias")
        println("  Val: $(length(val_individuals)) individuos␣
 ↪($(val_individuals)), $(size(val_fold,1)) instancias")
    end

    return folds
end
```

```
createFolds (generic function with 2 methods)
```

[7]:
```
function applyMinMaxScaler!(train_fold::DataFrame, val_fold::DataFrame)
    # Cojemos las los datos menos el identificar y la columna objetivo
    numeric_cols = setdiff(names(train_fold), ["subject", "Activity"])

    # Aplicamos MinMaxScaler a todas las columnas
    for col in numeric_cols
        train_min = minimum(train_fold[:, col])
        train_max = maximum(train_fold[:, col])

        train_fold[!, col] = (train_fold[:, col] .- train_min) ./ (train_max -␣
 ↪train_min)
        val_fold[!, col] = (val_fold[:, col] .- train_min) ./ (train_max -␣
 ↪train_min)
    end
end
```

```
applyMinMaxScaler! (generic function with 1 method)
```

[8]:
```
function processDataPipeline(dataset::DataFrame)
    # División inicial train/test (10%) "individual-wise"
    train_data, test_data = HoldOut(dataset);

    # Codificación + Nulos
    train_data_prepared, test_data_prepared, _ = prepareData!(train_data,␣
 ↪test_data);

    # Crear los 5 folds
    println("\n=== 5 Cross-Validation ===")
    folds = createFolds(train_data_prepared);

    # Para cada fold, aplicar MinMaxScaler
    for (i, (train_fold, val_fold)) in enumerate(folds)
        applyMinMaxScaler!(train_fold, val_fold)
    end

    return folds, test_data_prepared
end

train_folds, test_folds = processDataPipeline(dataset);
```

```
=== Hold-Out Split Information ===
Individuos Totales: 30
Individuos Train: 27
Individuos Test : 3
Individuos en test set: [1, 8, 25]

Distribución:
```

```
Train set: 9262 instancias (89.93%)
Test set: 1037 instancias (10.07%)

=== Valores Nulos Iniciales ===
Train nulos iniciales: 250
Test nulos iniciales: 20

=== Check Final ===
Número de columnas arregladas: 131
Valores nulos en Train: 0
Valores nulos en Test: 0
Dataset limpio, no contiene nulos!

=== 5 Cross-Validation ===
Fold 1:
  Train: 21 individuos, 7257 instancias
  Val: 6 individuos ([3, 7, 22, 29, 16, 6]), 2005 instancias
Fold 2:
  Train: 21 individuos, 7206 instancias
  Val: 6 individuos ([24, 10, 28, 15, 9, 30]), 2056 instancias
Fold 3:
  Train: 22 individuos, 7477 instancias
  Val: 5 individuos ([5, 19, 21, 14, 26]), 1785 instancias
Fold 4:
  Train: 22 individuos, 7543 instancias
  Val: 5 individuos ([23, 13, 2, 20, 18]), 1719 instancias
Fold 5:
  Train: 22 individuos, 7565 instancias
  Val: 5 individuos ([17, 27, 12, 11, 4]), 1697 instancias
```

5 Técnicas de Reducción de Dimensionalidad

Antes de entrenar nuestros modelos de clasificación, aplicaremos diferentes técnicas para reducir la dimensionalidad de nuestros datos. Esto nos permitirá: 1. Reducir la complejidad computacional 2. Mitigar el problema de la maldición de la dimensionalidad 3. Eliminar características redundantes o poco relevantes 4. Mejorar la interpretabilidad de nuestros modelos

Implementaremos cuatro estrategias diferentes:

5.1 1. Sin Reducción (Baseline)

Mantenemos todas las características originales como punto de referencia para comparar el rendimiento de las diferentes técnicas de reducción.

5.2 2. Filtrado ANOVA

- Evaluar la relación entre cada característica y la variable objetivo
- Identificar qué características muestran diferencias significativas entre las clases

- Seleccionar características basándonos en su significancia estadística (p-valor < 0.05)

5.3 3. Filtrado por Información Mutua

- Mide la dependencia estadística entre cada característica y la variable objetivo
- No asume relaciones lineales entre las variables
- Selecciona características basándose en un umbral mínimo de información mutua

5.4 4. Recursive Feature Elimination (RFE)

- Elimina iterativamente las características menos importantes
- Utiliza los coeficientes del modelo para rankear las características
- Reduce el conjunto de características en pasos del 50%
- Mantiene las interacciones entre características durante el proceso de selección

[9]:
```
function no_reduction(data::DataFrame; info::Bool=false)
    if info
        println("No se aplica reducción de dimensionalidad")
        println("Dimensiones originales: $(size(data))")
    end

    return data, Dict()
end
```

```
no_reduction (generic function with 1 method)
```

[10]:
```
function anova_filter(data::DataFrame, target_col::Symbol;
                      alpha::Float64=0.05,
                      info::Bool=false)
    try
        # Separar features y target, excluyendo 'subject'
        X = select(data, Not([:subject, target_col]))
        y = data[:, target_col]

        # Inicializar diccionario para p-valores
        p_values = Dict{String, Float64}()

        # Calcular ANOVA para cada característica
        for col in names(X)
            # Agrupar por clase
            grouped_data = groupby(data, target_col)
            groups = [df[!, col] for df in grouped_data]

            # Calcular estadísticas ANOVA
            f_stat, p_val = oneway_anova(groups)
            p_values[col] = p_val
        end

        # Seleccionar características significativas
```

```
        selected_features = [col for (col, p_val) in p_values if p_val < alpha]

        if info
            println("\nTotal de características: $(length(names(X)))")
            println("Características seleccionadas:␣
 ↪$(length(selected_features))")
        end

        reduced_data = select(data, vcat([:subject, target_col], Symbol.
 ↪(selected_features)))
        return reduced_data, p_values
    catch e
        error("Error en ANOVA filter: $e")
    end
end

function oneway_anova(groups::Vector)
    # Calcular medias y media global
    n_groups = length(groups)
    means = [mean(g) for g in groups]
    grand_mean = mean(vcat(groups...))

    # Calcular de la variabilidad entre los grupos
    ss_between = sum(length(g) * (mean(g) - grand_mean)^2 for g in groups)
    df_between = n_groups - 1

    # Calcular de la variabilidad dentro de los grupos
    ss_within = sum(sum((x - mean(g))^2 for x in g) for g in groups)
    df_within = sum(length(g) - 1 for g in groups)

    # Calcular MS y estadístico F
    ms_between = ss_between / df_between
    ms_within = ss_within / df_within
    f_statistic = ms_between / ms_within

    # Calcular p-valor
    f_dist = FDist(df_between, df_within)
    p_value = 1 - cdf(f_dist, f_statistic)

    return f_statistic, p_value
end
```

```
oneway_anova (generic function with 1 method)
```

[11]:
```
function mutual_information_filter(data::DataFrame, target_col::Symbol;
                                   threshold::Float64=0.05,
                                   info::Bool=false)
```

```
    try
        # Separar features y target
        X = Matrix(select(data, Not([:subject, target_col])))
        y = data[:, target_col]

        # Calcular scores de información mutua
        mi_scores = mutual_info_classif(X, y)

        # Crear diccionario de resultados
        feature_names = names(select(data, Not([:subject, target_col])))
        mi_dict = Dict(zip(feature_names, mi_scores))

        # Seleccionar características por encima del umbral
        selected_features = feature_names[mi_scores .> threshold]

        if info
            println("Total de características: $(length(feature_names))")
            println("Características seleccionadas:␣
 ↪$(length(selected_features))")
            println("Umbral de información mutua: $threshold")
            println("\nRango de scores MI:")
            println("Mínimo: $(minimum(mi_scores))")
            println("Máximo: $(maximum(mi_scores))")
            println("Media: $(mean(mi_scores))")
        end

        # Crear DataFrame reducido
        reduced_data = select(data, vcat([:subject, target_col], Symbol.
 ↪(selected_features)))
        return reduced_data, mi_dict
    catch e
        error("Error en Mutual Information filter: $e")
    end
end
```

```
mutual_information_filter (generic function with 1 method)
```

[12]:
```
function rfe_logistic_regression(data::DataFrame, target_col::Symbol;
                                 n_features::Int=50,
                                 step::Float64=0.5,
                                 info::Bool=false)
    try
        # Separar features y target, excluyendo 'subject'
        X = select(data, Not([:subject, target_col]))
        feature_names = names(X)
        X_matrix = Matrix(X)
        y = data[:, target_col]
```

```
    # Inicializar variables
    n_features_current = size(X_matrix, 2)
    features_to_keep = trues(n_features_current)
    rankings = Dict(name => 0 for name in feature_names)
    current_rank = 1

    if info
        println("\nIniciando RFE:")
        println("Características iniciales: ", n_features_current)
        println("Características objetivo: ", n_features)
        println("Paso de eliminación: ", step)
    end

    # Iterar hasta alcanzar el número deseado de características
    while n_features_current > n_features
        # Entrenar modelo con características actuales
        clf = LogisticRegression(multi_class="ovr", max_iter=1000)
        clf.fit(X_matrix[:, features_to_keep], y)

        # Obtener importancia de características
        if hasproperty(clf, :coef_)
            importance = vec(sum(abs.(clf.coef_), dims=1))
        else
            error("El modelo no proporciona coeficientes")
        end

        # Calcular número de características a eliminar
        n_to_drop = min(
            floor(Int, n_features_current * step),
            n_features_current - n_features
        )

        # Encontrar características menos importantes
        sorted_idx = sortperm(importance)
        features_to_drop = sorted_idx[1:n_to_drop]

        # Actualizar rankings
        current_features = findall(features_to_keep)
        for idx in current_features[features_to_drop]
            rankings[feature_names[idx]] = current_rank
        end
        current_rank += 1

        # Actualizar máscara de características
        mask_update = trues(n_features_current)
        mask_update[current_features[features_to_drop]] .= false
```

```
            features_to_keep .&= mask_update

            # Actualizar contador
            n_features_current -= n_to_drop

            if info
                println("Iteración completada: ", n_features_current, "␣
↪características restantes")
            end
        end

        # Asignar ranking final a características restantes
        for (i, keep) in enumerate(features_to_keep)
            if keep
                rankings[feature_names[i]] = current_rank
            end
        end

        if info
            println("\nRFE completado:")
            println("Características seleccionadas: ", sum(features_to_keep))
            println("Rankings asignados: ", length(unique(values(rankings))))
        end

        selected_features = feature_names[features_to_keep]
        reduced_data = select(data, vcat([:subject, target_col], Symbol.
↪(selected_features)))
        return reduced_data, rankings
    catch e
        error("Error en RFE: $e")
    end
end
```

```
rfe_logistic_regression (generic function with 1 method)
```

6 Evaluación de Técnicas de Reducción de Dimensionalidad

Tras implementar las diferentes técnicas de filtrado, procedemos a evaluar y comparar los distintos métodos de reducción de dimensionalidad.

6.0.1 Funciones Principales

1. `apply_dimensionality_reduction`:
   - Implementa las diferentes técnicas de reducción (PCA, LDA, ICA, Isomap, LLE)
   - Maneja tanto el conjunto de entrenamiento como el de validación
   - Preserva las etiquetas y los identificadores de sujetos
2. `evaluate_reduction_techniques`:
   - Evalúa sistemáticamente cada técnica usando validación cruzada

- Utiliza regresión logística como clasificador base
- Calcula métricas de rendimiento (accuracy) para cada método
- Proporciona estadísticas (media y desviación estándar)

3. `plot_reduction_results`:
   - Visualiza los resultados de cada técnica

[13]:
```
# Colocamos esta funcion aqui para que estea el "pipeline" todo junto
function apply_dimensionality_filter(data::DataFrame, method::Symbol;
                                     info::Bool=false, target_col::Symbol=:
 ↪Activity,
                                     kwargs...)
    if method == :none
        return no_reduction(data; kwargs...)
    elseif method == :anova
        return anova_filter(data, target_col; kwargs...)
    elseif method == :mutual_info
        return mutual_information_filter(data, target_col; kwargs...)
    elseif method == :rfe
        return rfe_logistic_regression(data, target_col; kwargs...)
    else
        error("Método de reducción no reconocido: $method")
    end
end
```

```
apply_dimensionality_filter (generic function with 1 method)
```

[14]:
```
function apply_dimensionality_reduction(
    fold::Tuple{DataFrame, DataFrame},
    method::Symbol;
    target_col::Symbol = :Activity,
    n_components::Int = 2
)
    train_fold, val_fold = fold

    # Separar features y target
    X_train = Matrix{Float64}(select(train_fold, Not([:subject, target_col])))
    y_train = train_fold[:, target_col]
    X_val = Matrix{Float64}(select(val_fold, Not([:subject, target_col])))
    y_val = val_fold[:, target_col]

    # Aplicar reducción
    if method == :none
        X_train_reduced = X_train[:, 1:2]
        X_val_reduced = X_val[:, 1:2]
    elseif method == :pca
        reducer = PCA(n_components = n_components)
        X_train_reduced = reducer.fit_transform(X_train)
        X_val_reduced = reducer.transform(X_val)
```

```
    elseif method == :lda
        reducer = LinearDiscriminantAnalysis(n_components = n_components)
        X_train_reduced = reducer.fit_transform(X_train, y_train)
        X_val_reduced = reducer.transform(X_val)
    elseif method == :ica
        reducer = FastICA(n_components = n_components)
        X_train_reduced = reducer.fit_transform(X_train)
        X_val_reduced = reducer.transform(X_val)
    elseif method == :isomap
        reducer = Isomap(n_components = n_components)
        X_train_reduced = reducer.fit_transform(X_train)
        X_val_reduced = reducer.transform(X_val)
    elseif method == :lle
        reducer = LocallyLinearEmbedding(n_components = n_components)
        X_train_reduced = reducer.fit_transform(X_train)
        X_val_reduced = reducer.transform(X_val)
    else
        error("Método de reducción no reconocido: $method")
    end

    # Crear DataFrames reducidos
    train_reduced = DataFrame(X_train_reduced, :auto)
    train_reduced.Activity = y_train
    train_reduced.subject = train_fold.subject

    val_reduced = DataFrame(X_val_reduced, :auto)
    val_reduced.Activity = y_val
    val_reduced.subject = val_fold.subject

    return train_reduced, val_reduced
end
```

```
apply_dimensionality_reduction (generic function with 1 method)
```

[15]:
```
function evaluate_reduction_techniques(
    folds::Vector{Tuple{DataFrame, DataFrame}},
    test_data::DataFrame,
)
    methods = [:none, :pca, :lda, :ica, :isomap, :lle]
    results = DataFrame(Fold = Int[], Method = String[], Accuracy = Float64[])

    for (i, fold) in enumerate(folds)
        # println("\nProcesando fold $i...")

        for method in methods
            try
                # Reducir dimensionalidad
```

```
                train_reduced, val_reduced =␣
↪apply_dimensionality_reduction(fold, method)

                # Preparar datos
                X_train = Matrix{Float64}(select(train_reduced, Not([:subject, :
↪Activity])))
                y_train = Array(train_reduced.Activity)
                X_val = Matrix{Float64}(select(val_reduced, Not([:subject, :
↪Activity])))
                y_val = Array(val_reduced.Activity)

                # Crear y entrenar el clasificador
                clf = LogisticRegression(max_iter = 1000)
                clf.fit(X_train, y_train)

                # Hacer predicciones
                y_pred = clf.predict(X_val)
                accuracy = mean(y_pred .== y_val)

                push!(results, (i, String(method), accuracy))
                # println("  Método $method - Accuracy: $(round(accuracy,␣
↪digits = 4))")

            catch e
                println("Error en fold $i, método $method:")
                println(e)
                bt = catch_backtrace()
                println("Backtrace:")
                display(stacktrace(bt))
                push!(results, (i, String(method), NaN))
            end
        end
    end

    if nrow(results) > 0
        avg_results = combine(
            groupby(results, :Method),
            :Accuracy => mean => :Mean_Accuracy,
            :Accuracy => std => :Std_Accuracy
        )
        # Redondear los valores a 3 decimales
        avg_results[:, :Mean_Accuracy] .= round.(avg_results.Mean_Accuracy,␣
↪digits=3)
        avg_results[:, :Std_Accuracy] .= round.(avg_results.Std_Accuracy,␣
↪digits=3)
        # Ordenar los resultados
```

```
        sort!(avg_results, :Mean_Accuracy, rev=true)

        println("\nResultados promedio por método:")
        println(avg_results)
        return results, avg_results
    else
        println("\nNo hay resultados para procesar")
        return results, DataFrame(
            Method = String[],
            Mean_Accuracy = Float64[],
            Std_Accuracy = Float64[]
        )
    end

end
```

```
evaluate_reduction_techniques (generic function with 1 method)
```

[16]:
```
function plot_reduction_results(fold::Tuple{DataFrame,DataFrame}, method::
 ↪Symbol, n_rows::Int=100)
    # Aplicar la reducción dimensional
    train, _ = apply_dimensionality_reduction(fold, method)

    p = scatter(train[:,1], train[:,2],
                group=train.Activity,
                title="Reducción usando $(method)",
                xlabel="Dimensión 1",
                ylabel="Dimensión 2",
                legend=:topright,
                markersize=4)

    return p
end

# Simplificar el analisis de todas las tecnicas
function plot_all_reductions(fold::Tuple{DataFrame,DataFrame}, n_rows::Int=100)
    methods = [:none, :pca, :lda, :ica, :isomap, :lle]
    plots = []

    for method in methods
        push!(plots, plot_reduction_results(fold, method))
    end
    final_plot = plot(plots...,
                      layout=(2,3),
                      size=(1500,800))
    return final_plot
end
```

```
plot_all_reductions (generic function with 2 methods)
```

6.1 Rendimiento de las Técnicas

Basándonos en los resultados obtenidos mediante validación cruzada, podemos observar que:

1. **PCA** muestra el mejor rendimiento (69.6% ± 2.9%):
   - Mantiene la mayor parte de la varianza en los datos
2. **ICA** y **Isomap** tienen un rendimiento similar:
   - ICA: 69.4% ± 2.8%
   - Isomap: 69.1% ± 2.5%
   - Ambos muestran patrones de separación diferentes pero efectivos
3. **LDA** muestra un rendimiento menor (66.4% ± 5%):
   - Mayor variabilidad entre folds (desviación estándar más alta de todos)
   - Sin embargo, la visualización muestra la separación más clara entre clases
4. **LLE** y **None** muestran el peor rendimiento:
   - LLE: 25.6% ± 5.5%
   - None: 22.2% ± 1%

6.2 Análisis Visual de las Proyecciones

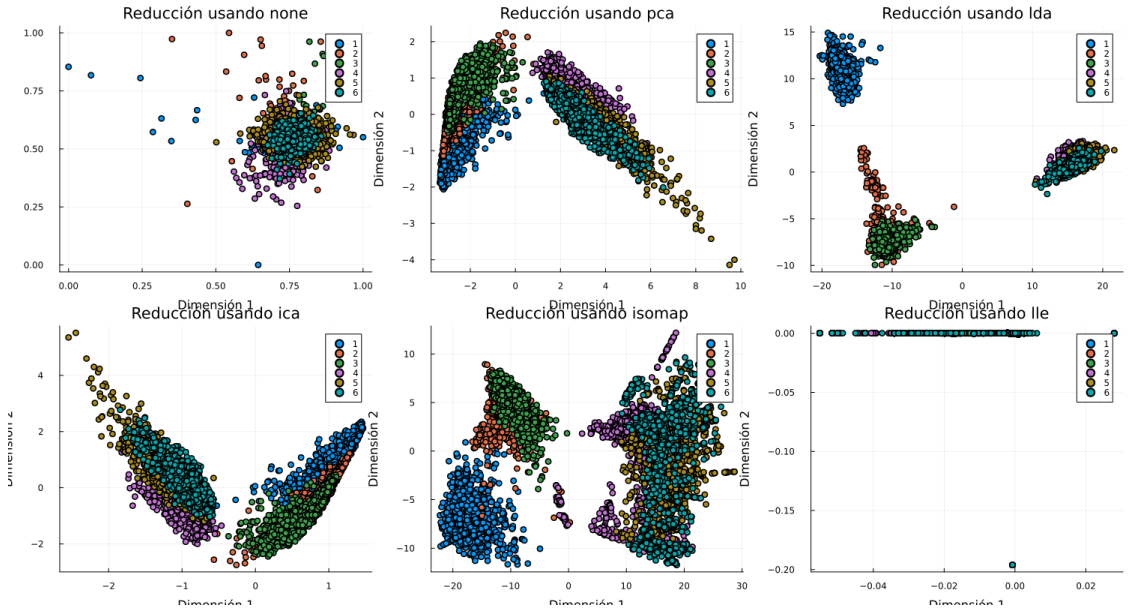
1. **PCA**:
   - Las actividades similares aparecen cercanas (ej: diferentes tipos de walking)
   - Mantiene una separación clara entre actividades estáticas y dinámicas
2. **LDA**:
   - Muestra clusters bien definidos
   - Especialmente efectivo para distinguir entre actividades estáticas (SITTING, STANDING, LAYING)
3. **ICA**:
   - Revela una estructura similar a PCA pero con diferente orientación
4. **Isomap**:
   - Muestra clusters más dispersos pero identificables
5. **LLE**:
   - Muestra un colapso significativo de la estructura
   - No logra preservar las relaciones importantes entre los datos

[17]:
```
evaluate_reduction_techniques(train_folds, test_folds);
```

```
Resultados promedio por método:
6×3 DataFrame
 Row   Method  Mean_Accuracy  Std_Accuracy
       String  Float64        Float64

   1   pca             0.696         0.029
   2   ica             0.694         0.028
   3   isomap          0.69          0.026
   4   lda             0.664         0.05
   5   lle             0.261         0.049
```

```
   6   none              0.222          0.01
6×3 DataFrame
 Row   Method  Mean_Accuracy  Std_Accuracy
       String  Float64        Float64

   1   pca               0.696          0.029
   2   ica               0.694          0.028
   3   isomap            0.69           0.026
   4   lda               0.664          0.05
   5   lle               0.261          0.049
   6   none              0.222          0.01
```

[18]:
```
fold = train_folds[1]
plot_all_reductions(fold)
```



7 Implementación de Clasificadores Basicos

7.1 Clasificadores Implementados

1. **Perceptrón Multicapa (MLP)**
   - Arquitecturas evaluadas:
     - Una capa oculta de 50 neuronas
     - Una capa oculta de 100 neuronas
     - Dos capas ocultas de 100 y 50 neuronas
2. **K-Nearest Neighbors (KNN)**
   - Valores de k evaluados:
     - k=1: Para capturar patrones muy locales
     - k=10: Balance entre local y global

- k=20: Para capturar patrones más globales

3. **Support Vector Machines (SVM)**
   - Valores del parámetro C evaluados:
     - C=0.1: Mayor regularización
     - C=0.5: Regularización moderada
     - C=1.0: Menor regularización

[19]:
```
function evaluate_classifiers(
    folds::Vector{Tuple{DataFrame, DataFrame}},
    reduction_method::Symbol = :none,
    n_components::Int = 2
)
    classifiers = Dict(
        # MLPs con diferentes arquitecturas
        "MLP_50" => MLPClassifier(hidden_layer_sizes = (50,), max_iter = 1000,␣
↪random_state = 42),
        "MLP_100" => MLPClassifier(hidden_layer_sizes = (100,), max_iter =␣
↪1000, random_state = 42),
        "MLP_100_50" => MLPClassifier(hidden_layer_sizes = (100, 50), max_iter␣
↪= 1000, random_state = 42),

        # KNNs con diferentes números de vecinos
        "KNN_1" => KNeighborsClassifier(n_neighbors = 1),
        "KNN_10" => KNeighborsClassifier(n_neighbors = 10),
        "KNN_20" => KNeighborsClassifier(n_neighbors = 20),

        # SVMs con diferentes valores de C
        "SVM_0.1" => SVC(C = 0.1, random_state = 42),
        "SVM_0.5" => SVC(C = 0.5, random_state = 42),
        "SVM_1.0" => SVC(C = 1.0, random_state = 42)
    )

    results = DataFrame(
        Fold = Int[],
        Classifier = String[],
        Accuracy = Float64[],
        F1_Score = Float64[]
    )
    trained_models = Dict{String, Vector{Any}}()

    for name in keys(classifiers)
        trained_models[name] = []
    end

    for (i, fold) in enumerate(folds)
        train_fold, val_fold = fold
        # Aplicar reducción de dimensionalidad
```

```
        train_reduced, val_reduced = apply_dimensionality_reduction(
            fold, reduction_method, n_components = n_components
        )

        # Preparar datos
        X_train = Matrix(select(train_reduced, Not([:subject, :Activity])))
        y_train = train_reduced.Activity
        X_val = Matrix(select(val_reduced, Not([:subject, :Activity])))
        y_val = val_reduced.Activity

        # Evaluar cada clasificador
        for (name, clf) in classifiers
            try
                clf_copy = deepcopy(clf)  # Crear una copia del clasificador
                fit!(clf_copy, X_train, y_train)
                y_pred = clf_copy.predict(X_val)

                # Como metrica usamos la Acc y F1
                accuracy = mean(y_pred .== y_val)
                f1 = f1_score(y_val, y_pred, average="weighted")
                push!(results, (i, name, accuracy, f1))
                push!(trained_models[name], clf_copy)
            catch e
                push!(results, (i, name, NaN, NaN))
                push!(trained_models[name], nothing)
            end
        end
    end

    if nrow(results) > 0
        stats = combine(
            groupby(results, :Classifier),
            :Accuracy => mean => :Mean_Accuracy,
            :Accuracy => std => :Std_Accuracy,
            :F1_Score => mean => :Mean_F1,
            :F1_Score => std => :Std_F1
        )
        # Ordenamos los modelos por precisión media
        sort!(stats, :Mean_Accuracy, rev=true)
        return results, stats, trained_models
    else
        return results, DataFrame(
            Classifier = String[],
            Mean_Accuracy = Float64[],
            Std_Accuracy = Float64[],
            Mean_F1 = Float64[],
            Std_F1 = Float64[]
```

```
        ), trained_models
    end
end
```

```
evaluate_classifiers (generic function with 3 methods)
```

[20]:
```
# Funcion para simplicar el analisis de las combinaciones
function evaluate_all_combinations(
    folds::Vector{Tuple{DataFrame, DataFrame}},
    reduction_methods = [:none, :pca, :lda, :ica, :isomap, :lle]
)
    all_results = DataFrame(
        Reduction = String[],
        Classifier = String[],
        Mean_Accuracy = Float64[],
        Std_Accuracy = Float64[],
        Mean_F1 = Float64[],
        Std_F1 = Float64[]
    )
    all_models = Dict{Symbol, Dict{String, Any}}()

    for method in reduction_methods
        # Evaluar clasificadores con el método de reducción actual
        results, stats, trained_models = evaluate_classifiers(folds, method)

        # Agregar método de reducción a las estadísticas
        transform!(stats, [] => ByRow(() -> String(method)) => :Reduction)
        append!(all_results, stats[:, [:Reduction, :Classifier, :Mean_Accuracy,␣
 ↪:Std_Accuracy, :Mean_F1, :Std_F1]])

        # Guardar resultados, estadísticas y modelos
        all_models[method] = Dict(
            "Results" => results,
            "Stats" => stats,
            "Models" => trained_models
        )
    end

    # Ordenar resultados por precisión media
    sort!(all_results, :Mean_Accuracy, rev = true)

    println("\n=== TOP 10 MEJORES COMBINACIONES ===")
    println("====================================")
    for (i, row) in enumerate(eachrow(first(all_results, 10)))
        println(
            @sprintf(
```

```
                "%2d. %-6s con %-10s → Acc: %.2f%% (±%.2f%%) F1: %.2f%% (±%.
↪2f%%)",
                i,
                row.Reduction,
                row.Classifier,
                row.Mean_Accuracy * 100,
                row.Std_Accuracy * 100,
                row.Mean_F1 * 100,
                row.Std_F1 * 100
            )
        )
    end

    return all_results, all_models
end
```

```
evaluate_all_combinations (generic function with 2 methods)
```

[21]:
```
function get_best_models_overall(all_results::DataFrame, all_models::Dict)
    global_best = Dict()

    # Para cada tipo de modelo
    for model_type in ["MLP", "KNN", "SVM"]
        # Filtrar todos los resultados de este tipo de modelo
        type_results = filter(row -> startswith(row.Classifier, model_type),␣
↪all_results)

        if nrow(type_results) > 0
            # Encontrar la mejor fila basada en precisión media
            best_row = type_results[argmax(type_results.Mean_Accuracy), :]

            # Obtener el método de reducción y nombre del clasificador
            reduction = Symbol(best_row.Reduction)
            classifier_name = best_row.Classifier

            # Guardar la información del mejor modelo
            global_best[model_type] = Dict(
                "reduction" => reduction,
                "name" => classifier_name,
                "metrics" => best_row,
                "model" => all_models[reduction]["Models"][classifier_name]
            )
        end
    end

    return global_best
end
```

```
get_best_models_overall (generic function with 1 method)
```

8 Análisis de Resultados de los Modelos Base

8.0.1 Mejores Combinaciones

El análisis revela que **Isomap** destaca como la técnica de reducción más efectiva, con las primeras 6 posiciones:

1. **MLP (100,50) + Isomap**: 71.42% ±1.28% de accuracy
2. **KNN (k=20) + Isomap**: 71.38% ±2.16% de accuracy
3. **MLP (50) + Isomap**: 71.13% ±1.66% de accuracy

Esta dominancia de Isomap sugiere que los datos tienen una estructura no lineal importante que esta técnica consigue capturar mejor que las alternativas lineales como PCA o LDA.

8.1 Análisis por Tipo de Modelo

8.1.1 Perceptrón Multicapa (MLP)

- **Mejor configuración**: Arquitectura [100,50] con Isomap
- **Rendimiento**: 71.42% de accuracy (±1.28%)
- **F1-Score**: 70.78% (±2.08%)

8.1.2 K-Nearest Neighbors (KNN)

- **Mejor configuración**: k=20 con Isomap
- **Rendimiento**: 71.38% de accuracy (±2.16%)
- **F1-Score**: 71.27% (±2.29%)

8.1.3 Support Vector Machine (SVM)

- **Mejor configuración**: C=1.0 con Isomap
- **Rendimiento**: 70.93% de accuracy (±1.68%)
- **F1-Score**: 70.28% (±2.09%)

8.2 Observaciones

1. La dominancia de Isomap sugiere que la estructura no lineal de los datos es un factor crítico para la clasificación.
2. Todos los mejores modelos muestran desviaciones estándar relativamente bajas (±1.28% a ±2.16%).
3. La proximidad entre los valores de accuracy y F1-Score sugiere que los modelos manejan bien el balance entre clases.

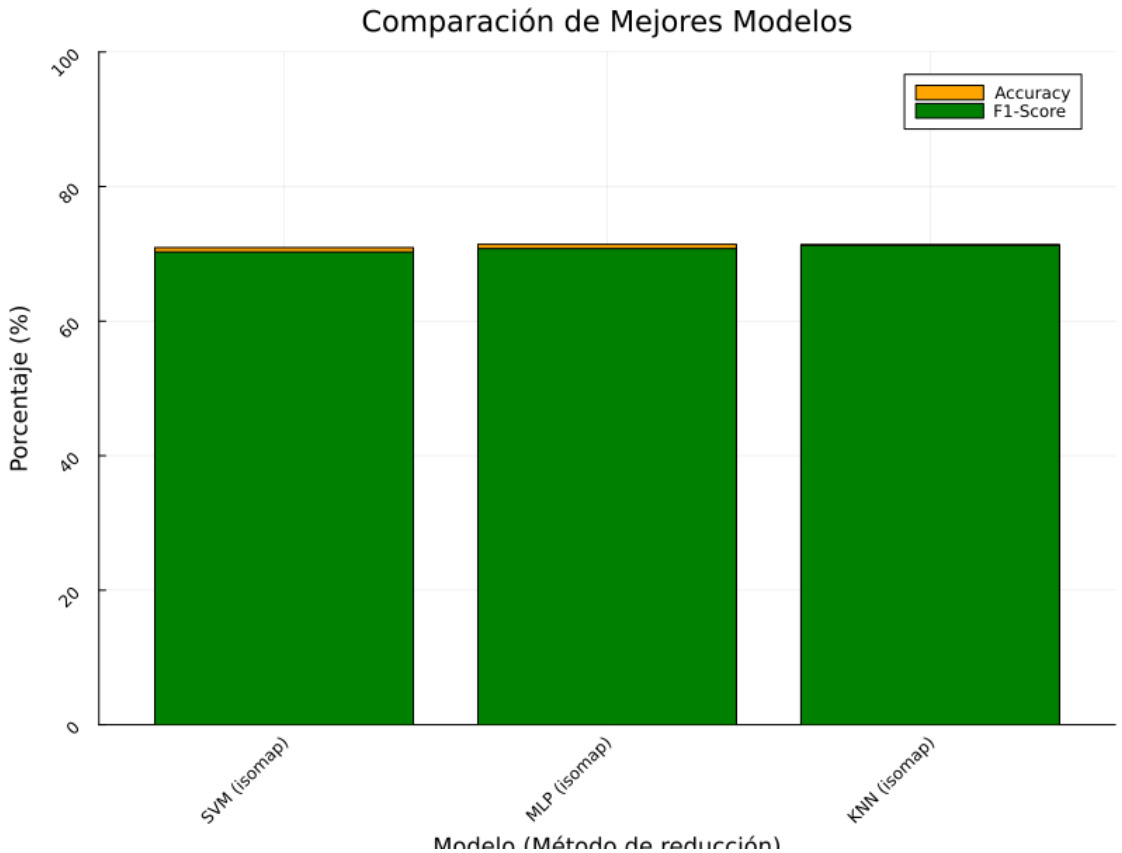
[22]:
```
all_results, all_models = evaluate_all_combinations(train_folds)
best_models_overall = get_best_models_overall(all_results, all_models)

# Mostrar los resultados
println("\n=== MEJORES MODELOS GLOBALES POR TIPO ===")
```

```
for (model_type, info) in best_models_overall
    metrics = info["metrics"]
    println("$model_type: $(info["reduction"]) con $(info["name"]) → " *
            "Acc: $(round(metrics.Mean_Accuracy*100, digits=2))% " *
            "(±$(round(metrics.Std_Accuracy*100, digits=2))%) " *
            "F1: $(round(metrics.Mean_F1*100, digits=2))% " *
            "(±$(round(metrics.Std_F1*100, digits=2))%)")
end
```

```
=== TOP 10 MEJORES COMBINACIONES ===
====================================
 1. isomap con MLP_100_50 → Acc: 71.42% (±1.28%) F1: 70.78% (±2.08%)
 2. isomap con KNN_20     → Acc: 71.38% (±2.16%) F1: 71.27% (±2.29%)
 3. isomap con MLP_50     → Acc: 71.13% (±1.66%) F1: 70.69% (±1.91%)
 4. isomap con MLP_100    → Acc: 70.96% (±1.74%) F1: 70.22% (±2.35%)
 5. isomap con SVM_1.0    → Acc: 70.93% (±1.68%) F1: 70.28% (±2.09%)
 6. isomap con SVM_0.5    → Acc: 70.75% (±1.51%) F1: 70.10% (±1.79%)
 7. ica     con MLP_100_50 → Acc: 70.19% (±2.70%) F1: 69.15% (±3.25%)
 8. pca     con MLP_100_50 → Acc: 70.19% (±2.81%) F1: 69.13% (±3.51%)
 9. isomap con SVM_0.1    → Acc: 70.09% (±1.36%) F1: 69.21% (±1.86%)
10. ica     con MLP_100    → Acc: 70.06% (±2.44%) F1: 68.97% (±3.15%)

=== MEJORES MODELOS GLOBALES POR TIPO ===
SVM: isomap con SVM_1.0 → Acc: 70.93% (±1.68%) F1: 70.28% (±2.09%)
MLP: isomap con MLP_100_50 → Acc: 71.42% (±1.28%) F1: 70.78% (±2.08%)
KNN: isomap con KNN_20 → Acc: 71.38% (±2.16%) F1: 71.27% (±2.29%)
```

[23]:
```
function plot_model_comparison(best_models_overall)

    # Extraer nombres y métricas
    model_names = collect(keys(best_models_overall))
    accuracies = [best_models_overall[m]["metrics"].Mean_Accuracy * 100 for m␣
 ↪in model_names]
    f1_scores = [best_models_overall[m]["metrics"].Mean_F1 * 100 for m in␣
 ↪model_names]

    labels = ["$name ($(best_models_overall[name]["reduction"]))" for name in␣
 ↪model_names]

    # Graficar barras (no conseguí ponerlas separadas la acc y f1, solo así)
    bar(
        labels,
        [accuracies f1_scores],
        label = ["Accuracy" "F1-Score"],
        legend = :topright,
        bar_position = :dodge,
```

```
        c = [:orange :green],
        title = "Comparación de Mejores Modelos",
        xlabel = "Modelo (Método de reducción)",
        ylabel = "Porcentaje (%)",
        ylims = (0, 100),
        rotation = 45,
        size = (800, 600)
    )
end

p = plot_model_comparison(best_models_overall)
display(p)
```




```
  Warning: Keyword argument bar_position not supported with Plots.GRBackend().
Choose from: annotationcolor, annotationfontfamily, annotationfontsize,
annotationhalign, annotationrotation, annotations, annotationvalign, arrow,
aspect_ratio, axis, background_color, background_color_inside,
background_color_outside, background_color_subplot, bar_width, bins,
bottom_margin, camera, clims, color_palette, colorbar, colorbar_entry,
colorbar_scale, colorbar_title, colorbar_titlefont, colorbar_titlefontcolor,
colorbar_titlefontrotation, colorbar_titlefontsize, connections, contour_labels,
```

discrete_values, fill, fill_z, fillalpha, fillcolor, fillrange, fillstyle, flip,
fontfamily, fontfamily_subplot, foreground_color, foreground_color_axis,
foreground_color_border, foreground_color_grid, foreground_color_subplot,
foreground_color_text, formatter, framestyle, grid, gridalpha, gridlinewidth,
gridstyle, group, guide, guidefont, guidefontcolor, guidefontfamily,
guidefonthalign, guidefontrotation, guidefontsize, guidefontvalign,
html_output_format, inset_subplots, label, layout, left_margin,
legend_background_color, legend_column, legend_font, legend_font_color,
legend_font_family, legend_font_halign, legend_font_pointsize,
legend_font_rotation, legend_font_valign, legend_foreground_color,
legend_position, legend_title, legend_title_font_color,
legend_title_font_family, legend_title_font_pointsize,
legend_title_font_rotation, legend_title_font_valigm, levels, lims, line,
line_z, linealpha, linecolor, linestyle, linewidth, link, margin, marker_z,
markeralpha, markercolor, markershape, markersize, markerstrokealpha,
markerstrokecolor, markerstrokewidth, minorgrid, minorgridalpha,
minorgridlinewidth, minorgridstyle, minorticks, mirror, normalize, orientation,
overwrite_figure, permute, plot_title, plot_titlefontcolor,
plot_titlefontfamily, plot_titlefontrotation, plot_titlefontsize,
plot_titlelocation, plot_titlevspan, polar, primary, projection, quiver, ribbon,
right_margin, rotation, scale, series_annotations, seriesalpha, seriescolor,
seriestype, show, show_empty_bins, showaxis, size, smooth, subplot,
subplot_index, thickness_scaling, tick_direction, tickfontcolor, tickfontfamily,
tickfonthalign, tickfontrotation, tickfontsize, tickfontvalign, ticks, title,
titlefontcolor, titlefontfamily, titlefonthalign, titlefontrotation,
titlefontsize, titlefontvalign, top_margin, unitformat, weights, widen,
window_title, x, xdiscrete_values, xerror, xflip, xforeground_color_axis,
xforeground_color_border, xforeground_color_grid, xforeground_color_text,
xformatter, xgrid, xgridalpha, xgridlinewidth, xgridstyle, xguide,
xguidefontcolor, xguidefontfamily, xguidefonthalign, xguidefontrotation,
xguidefontsize, xguidefontvalign, xlims, xlink, xminorgrid, xminorgridalpha,
xminorgridlinewidth, xminorgridstyle, xminorticks, xmirror, xrotation, xscale,
xshowaxis, xtick_direction, xtickfontcolor, xtickfontfamily, xtickfonthalign,
xtickfontrotation, xtickfontsize, xtickfontvalign, xticks, xunitformat, xwiden,
y, ydiscrete_values, yerror, yflip, yforeground_color_axis,
yforeground_color_border, yforeground_color_grid, yforeground_color_text,
yformatter, ygrid, ygridalpha, ygridlinewidth, ygridstyle, yguide,
yguidefontcolor, yguidefontfamily, yguidefonthalign, yguidefontrotation,
yguidefontsize, yguidefontvalign, ylims, ylink, yminorgrid, yminorgridalpha,
yminorgridlinewidth, yminorgridstyle, yminorticks, ymirror, yrotation, yscale,
yshowaxis, ytick_direction, ytickfontcolor, ytickfontfamily, ytickfonthalign,
ytickfontrotation, ytickfontsize, ytickfontvalign, yticks, yunitformat, ywiden,
z, z_order, zdiscrete_values, zerror, zflip, zforeground_color_axis,
zforeground_color_border, zforeground_color_grid, zforeground_color_text,
zformatter, zgrid, zgridalpha, zgridlinewidth, zgridstyle, zguide,
zguidefontcolor, zguidefontfamily, zguidefonthalign, zguidefontrotation,
zguidefontsize, zguidefontvalign, zlims, zlink, zminorgrid, zminorgridalpha,
zminorgridlinewidth, zminorgridstyle, zminorticks, zmirror, zrotation, zscale,

```
zshowaxis, ztick_direction, ztickfontcolor, ztickfontfamily, ztickfonthalign,
ztickfontrotation, ztickfontsize, ztickfontvalign, zticks, zunitformat, zwiden
  @ Plots /home/clown/.julia/packages/Plots/FFuQi/src/args.jl:1557
```

9 Modelos Ensemble para Clasificación de Actividades

9.0.1 1. Bagging con KNN

- Utiliza KNN como clasificador base (k=5)
- Se prueban dos configuraciones:
  - 10 estimadores
  - 50 estimadores

9.0.2 2. AdaBoost con SVM

- Clasificador base: SVM lineal con probabilidades
- 5 estimadores

9.0.3 3. Gradient Boosting (GBM)

- 50 estimadores
- Learning rate de 0.2

9.1 Pipeline de Entrenamiento y Evaluación

El proceso se divide en varias etapas clave:

1. **Preprocesamiento**:
   - Aplicación de filtrado ANOVA
2. **Entrenamiento**:
   - 5-fold cross-validation
3. **Evaluación**:
   - Métricas principales: Accuracy y F1-Score

[24]:
```
function train_bagging_knn(X, y; n_estimators=10)
    X_matrix = Matrix(X)

    base_knn = KNeighborsClassifier(n_neighbors=5)

    bagging = BaggingClassifier(
        estimator=base_knn,
        n_estimators=n_estimators,
        random_state=42
    )

    fit!(bagging, X_matrix, y)
    return bagging
end
```

```
function train_adaboost_svm(X, y)
    X_matrix = Matrix(X)

    base_svm = SVC(kernel="linear", probability=true)

    ada = AdaBoostClassifier(
        estimator=base_svm,
        n_estimators=5,
        random_state=42
    )

    fit!(ada, X_matrix, y)
    return ada
end

function train_gbm(X, y)
    X_matrix = Matrix(X)

    gbm = GradientBoostingClassifier(
        n_estimators=50,
        learning_rate=0.2,
        random_state=42
    )

    fit!(gbm, X_matrix, y)
    return gbm
end
```

```
train_gbm (generic function with 1 method)
```

[ ]:
```
function train_ensemble_models(fold; verbose=true)

    # Separar el fold
    train_fold, val_fold = fold
    if verbose
        println("\nPreparando datos...")
    end

    # Aplicar ANOVA al conjunto de entrenamiento
    n_features = size(select(train_fold, Not([:subject, :Activity])), 2)
    alpha_bonferroni = 0.05 / n_features # Corrección de Bonferroni
    train_filtered, p_values = anova_filter(train_fold, :Activity, info=verbose)

    # Aplicar la misma selección de características al conjunto de validación
    selected_features = names(select(train_filtered, Not([:subject, :
 ↪Activity])))
```

```
    val_filtered = select(val_fold, vcat([:subject, :Activity], Symbol.
↪(selected_features)))

    # Preparar los datos filtrados
    X_train = Matrix{Float64}(select(train_filtered, Not([:subject, :
↪Activity])))
    y_train = train_filtered[!, :Activity]
    X_test = Matrix{Float64}(select(val_filtered, Not([:subject, :Activity])))
    y_test = val_filtered[!, :Activity]

    models = Dict()
    results = Dict()

    # Entrenar modelos
    if verbose
        println("\nEntrenando modelos...")
    end

    # Bagging KNN (10 estimadores)
    if verbose
        println("Entrenando Bagging KNN (10 estimadores)...")
    end
    models["bagging_knn_10"] = train_bagging_knn(X_train, y_train,␣
↪n_estimators=10)

    # Bagging KNN (50 estimadores)
    if verbose
        println("Entrenando Bagging KNN (50 estimadores)...")
    end
    models["bagging_knn_50"] = train_bagging_knn(X_train, y_train,␣
↪n_estimators=50)

    # AdaBoost SVM
    if verbose
        println("Entrenando AdaBoost SVM...")
    end
    models["adaboost_svm"] = train_adaboost_svm(X_train, y_train)

    # GBM
    if verbose
        println("Entrenando GBM...")
    end
    models["gbm"] = train_gbm(X_train, y_train)

    # Evaluación de modelos
    if verbose
        println("\nEvaluando modelos...")
```

```
    end
    # Calcular estadisticas de los modelos
    for (name, model) in models
        y_pred = model.predict(X_test)
        accuracy = mean(y_pred .== y_test)
        f1 = f1_score(y_test, y_pred, average="weighted")
        results[name] = Dict("accuracy" => accuracy, "f1" => f1)

        if verbose
            println(" - $name: Accuracy = $(round(accuracy, digits=4)), F1 =␣
↪$(round(f1, digits=4))")
        end
    end

    return models, results
end
```

```
train_ensemble_models (generic function with 1 method)
```

[ ]:
```
function ensemble_pipeline(folds; verbose=true)
    all_results = []
    all_models = Dict()
    # Para guardar los modelos por nombre
    trained_models = Dict{String, Vector{Any}}()

    # Inicializar los vectores de modelos entrenados
    first_fold_models, _ = train_ensemble_models(folds[1], verbose=false)
    for model_name in keys(first_fold_models)
        trained_models[model_name] = []
    end

    for (i, fold) in enumerate(folds)
        if verbose
            println("\nProcesando fold $i de $(length(folds))...")
            println("----------------------------------------")
            train_fold, val_fold = fold
            println("Tamaño conjunto entrenamiento: $(size(train_fold, 1))␣
↪muestras")
            println("Tamaño conjunto test: $(size(val_fold, 1)) muestras")
        end

        # Entrenar modelos y obtener resultados
        fold_models, fold_results = train_ensemble_models(fold, verbose=verbose)

        # Guardar resultados y modelos
        push!(all_results, fold_results)
        all_models[i] = fold_models
```

```
        # Guardar los modelos entrenados por nombre
        for (model_name, model) in fold_models
            push!(trained_models[model_name], model)
        end

        if verbose
            println("\nResultados fold $i:")
            for (model, metrics) in fold_results
                println(" - $model: acc=$(round(metrics["accuracy"],␣
↪digits=4)), f1=$(round(metrics["f1"], digits=4))")
            end
        end
    end
    # Calcular estadísticas finales y guardar información de los mejores folds
    stats = Dict()
    if verbose
        println("\nCalculando estadísticas finales...")
    end
    for model_name in keys(all_results[1])
        accuracies = [result[model_name]["accuracy"] for result in all_results]
        f1_scores = [result[model_name]["f1"] for result in all_results]
        best_fold_idx = argmax(accuracies)
        stats[model_name] = Dict(
            "mean_accuracy" => mean(accuracies),
            "std_accuracy" => std(accuracies),
            "min_accuracy" => minimum(accuracies),
            "max_accuracy" => maximum(accuracies),
            "mean_f1" => mean(f1_scores),
            "std_f1" => std(f1_scores),
            "best_fold" => best_fold_idx,
            "best_model" => trained_models[model_name][best_fold_idx]
        )
    end
    return stats, trained_models
end
```

```
ensemble_pipeline (generic function with 1 method)
```

[27]:
```
stats, models = ensemble_pipeline(train_folds, verbose=true);
```

```
Procesando fold 1 de 5…
----------------------------------------
Tamaño conjunto entrenamiento: 7257 muestras
Tamaño conjunto test: 2005 muestras

Preparando datos…
```

```
Total de características: 561
Características seleccionadas: 556

Entrenando modelos…
Entrenando Bagging KNN (10 estimadores)…
Entrenando Bagging KNN (50 estimadores)…
Entrenando AdaBoost SVM…
Entrenando GBM…

Evaluando modelos…
 - bagging_knn_10: Accuracy = 0.9037, F1 = 0.9036
 - bagging_knn_50: Accuracy = 0.9062, F1 = 0.9061
 - gbm: Accuracy = 0.9372, F1 = 0.9372
 - adaboost_svm: Accuracy = 0.4175, F1 = 0.2878

Resultados fold 1:
 - bagging_knn_10: acc=0.9037, f1=0.9036
 - bagging_knn_50: acc=0.9062, f1=0.9061
 - gbm: acc=0.9372, f1=0.9372
 - adaboost_svm: acc=0.4175, f1=0.2878

Procesando fold 2 de 5…
----------------------------------------
Tamaño conjunto entrenamiento: 7206 muestras
Tamaño conjunto test: 2056 muestras

Preparando datos…

Total de características: 561
Características seleccionadas: 556

Entrenando modelos…
Entrenando Bagging KNN (10 estimadores)…
Entrenando Bagging KNN (50 estimadores)…
Entrenando AdaBoost SVM…
Entrenando GBM…

Evaluando modelos…
 - bagging_knn_10: Accuracy = 0.8974, F1 = 0.8993
 - bagging_knn_50: Accuracy = 0.8998, F1 = 0.9015
 - gbm: Accuracy = 0.9392, F1 = 0.9391
 - adaboost_svm: Accuracy = 0.5603, F1 = 0.4698

Resultados fold 2:
 - bagging_knn_10: acc=0.8974, f1=0.8993
 - bagging_knn_50: acc=0.8998, f1=0.9015
 - gbm: acc=0.9392, f1=0.9391
```

```
 - adaboost_svm: acc=0.5603, f1=0.4698

Procesando fold 3 de 5…
----------------------------------------
Tamaño conjunto entrenamiento: 7477 muestras
Tamaño conjunto test: 1785 muestras

Preparando datos…

Total de características: 561
Características seleccionadas: 555

Entrenando modelos…
Entrenando Bagging KNN (10 estimadores)…
Entrenando Bagging KNN (50 estimadores)…
Entrenando AdaBoost SVM…
Entrenando GBM…

Evaluando modelos…
 - bagging_knn_10: Accuracy = 0.8812, F1 = 0.8805
 - bagging_knn_50: Accuracy = 0.8857, F1 = 0.8849
 - gbm: Accuracy = 0.921, F1 = 0.9202
 - adaboost_svm: Accuracy = 0.4896, F1 = 0.3768

Resultados fold 3:
 - bagging_knn_10: acc=0.8812, f1=0.8805
 - bagging_knn_50: acc=0.8857, f1=0.8849
 - gbm: acc=0.921, f1=0.9202
 - adaboost_svm: acc=0.4896, f1=0.3768

Procesando fold 4 de 5…
----------------------------------------
Tamaño conjunto entrenamiento: 7543 muestras
Tamaño conjunto test: 1719 muestras

Preparando datos…

Total de características: 561
Características seleccionadas: 556

Entrenando modelos…
Entrenando Bagging KNN (10 estimadores)…
Entrenando Bagging KNN (50 estimadores)…
Entrenando AdaBoost SVM…
Entrenando GBM…

Evaluando modelos…
 - bagging_knn_10: Accuracy = 0.8883, F1 = 0.8848
```

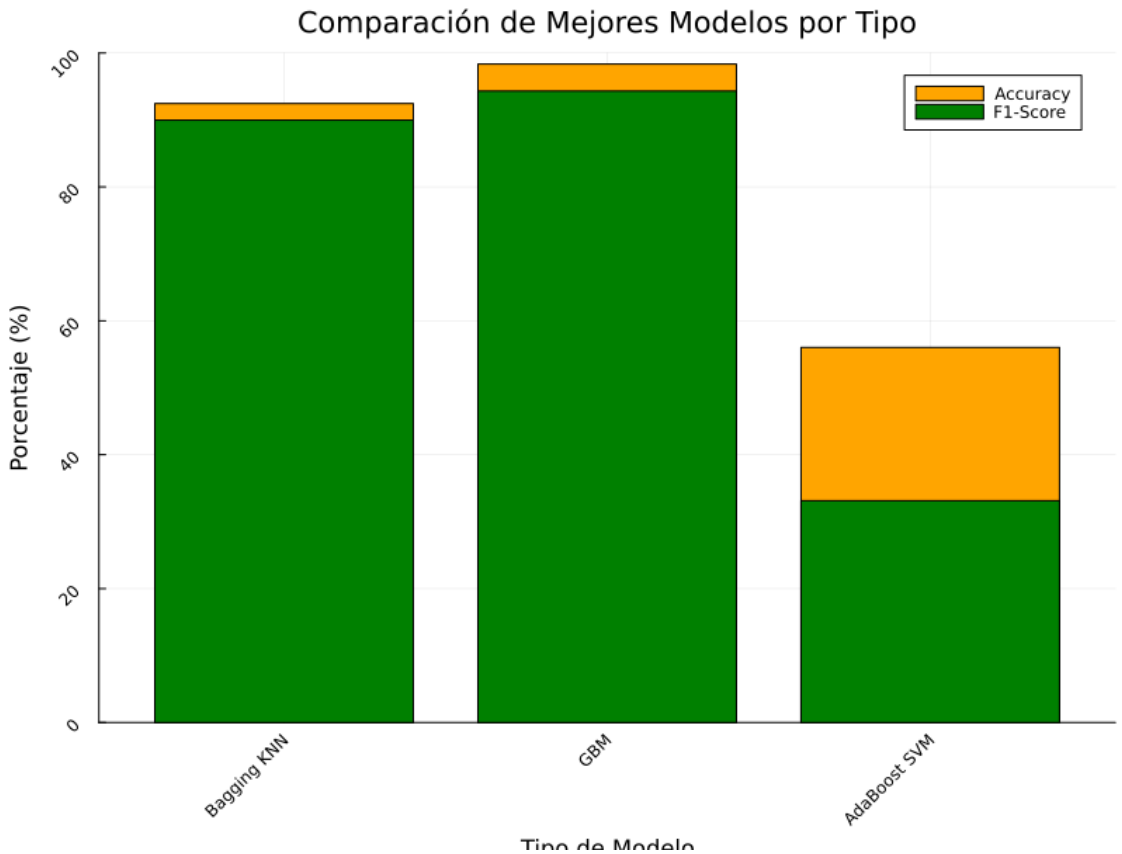
```
 - bagging_knn_50: Accuracy = 0.8854, F1 = 0.8817
 - gbm: Accuracy = 0.9366, F1 = 0.9357
 - adaboost_svm: Accuracy = 0.409, F1 = 0.2715

Resultados fold 4:
 - bagging_knn_10: acc=0.8883, f1=0.8848
 - bagging_knn_50: acc=0.8854, f1=0.8817
 - gbm: acc=0.9366, f1=0.9357
 - adaboost_svm: acc=0.409, f1=0.2715

Procesando fold 5 de 5…
----------------------------------------
Tamaño conjunto entrenamiento: 7565 muestras
Tamaño conjunto test: 1697 muestras

Preparando datos…

Total de características: 561
Características seleccionadas: 556

Entrenando modelos…
Entrenando Bagging KNN (10 estimadores)…
Entrenando Bagging KNN (50 estimadores)…
Entrenando AdaBoost SVM…
Entrenando GBM…

Evaluando modelos…
 - bagging_knn_10: Accuracy = 0.9216, F1 = 0.9214
 - bagging_knn_50: Accuracy = 0.9246, F1 = 0.9244
 - gbm: Accuracy = 0.9835, F1 = 0.9835
 - adaboost_svm: Accuracy = 0.3824, F1 = 0.2506

Resultados fold 5:
 - bagging_knn_10: acc=0.9216, f1=0.9214
 - bagging_knn_50: acc=0.9246, f1=0.9244
 - gbm: acc=0.9835, f1=0.9835
 - adaboost_svm: acc=0.3824, f1=0.2506

Calculando estadísticas finales…
```

[28]:
```julia
# Función para obtener el tipo de modelo base de forma más sencilla
function get_model_type(model_name::String)
    if startswith(model_name, "bagging_knn")
        return "Bagging KNN"
    elseif startswith(model_name, "adaboost")
        return "AdaBoost SVM"
    elseif startswith(model_name, "gbm")
```

```
        return "GBM"
    else
        return "Unknown"
    end
end

results_df = DataFrame(
    Model = String[],
    Mean_Accuracy = Float64[],
    Max_Accuracy = Float64[],
    Best_Fold = Int[],
    Mean_F1 = Float64[],
    Std_F1 = Float64[]
)

for (model_name, metrics) in stats
    push!(results_df, [
        model_name,
        metrics["mean_accuracy"],
        metrics["max_accuracy"],
        metrics["best_fold"],
        metrics["mean_f1"],
        metrics["std_f1"]
    ])
end

# Añadir columna con el tipo de modelo
results_df.ModelType = get_model_type.(results_df.Model)

# Obtener el mejor modelo de cada tipo
best_models = combine(groupby(results_df, :ModelType)) do group
    sort(group, :Max_Accuracy, rev=true)[1, :]
end

# Imprimimos el mejor modelo por tipo segun la mejor Accuracy
for row in eachrow(best_models)
    best_model = stats[row.Model]["best_model"]
    println("Mejor modelo para $(row.ModelType): accuracy=$(row.Max_Accuracy),␣
 ↪f1=$(row.Max_Accuracy) (fold $(row.Best_Fold))")
end
```

```
Mejor modelo para Bagging KNN: accuracy=0.924572775486152, f1=0.924572775486152
(fold 5)
Mejor modelo para GBM: accuracy=0.9835002946375958, f1=0.9835002946375958 (fold
5)
Mejor modelo para AdaBoost SVM: accuracy=0.5603112840466926,
f1=0.5603112840466926 (fold 2)
```

[29]:
```julia
function plot_model_comparison(results_df)

    # Agrupar por tipo de modelo y obtener el mejor de cada tipo
    best_models = combine(groupby(results_df, :ModelType)) do group
        sort(group, :Max_Accuracy, rev=true)[1, :]
    end

    # Extraer datos para el gráfico
    model_types = best_models.ModelType
    accuracies = best_models.Max_Accuracy .* 100  # Convertir a porcentaje
    f1_scores = best_models.Mean_F1 .* 100        # Convertir a porcentaje

    bar(
        model_types,
        [accuracies f1_scores],
        label = ["Accuracy" "F1-Score"],
        legend = :topright,
        bar_position = :dodge,
        c = [:orange :green],
        title = "Comparación de Mejores Modelos por Tipo",
        xlabel = "Tipo de Modelo",
        ylabel = "Porcentaje (%)",
        ylims = (0, 100),
        rotation = 45,
        size = (800, 600)
    )
end
p = plot_model_comparison(results_df)
display(p)
```



9.2 Resultados y Análisis

1. **Gradient Boosting (GBM)**:
   - Mejor rendimiento global con 98.35% de accuracy máxima (fold 5)
   - F1-Score consistentemente alto (93-98%)
   - Menor variabilidad entre folds (92.1%-98.35%)
2. **Bagging KNN**:
   - 50 estimadores supera ligeramente a 10 estimadores
   - Accuracy entre 88.54% y 92.46%
   - Estabilidad notable en F1-Score (~90%)
3. **AdaBoost SVM**:
   - Rendimiento inferior (38.24%-56.03% accuracy)
   - Alta variabilidad entre folds
   - F1-Scores bajos, especialmente en últimos folds

La superioridad del GBM se mantiene consistente a través de los folds, mientras que AdaBoost SVM muestra limitaciones significativas en este conjunto de datos específico, principalmente porque:

1. AdaBoost está usando un SVM con kernel lineal que no puede capturar las relaciones no lineales
2. GBM usa árboles de decisión que pueden adaptarse a estas relaciones no lineales y manejar mejor la alta dimensionalidad

10 Modelos Ensemble Avanzados con Conjunto Completo

10.0.1 1. Random Forest

- 500 árboles de decisión
- Profundidad máxima de 10

10.0.2 2. Hard Voting

- Combina los mejores modelos de cada tipo:
  - Mejor Bagging KNN
  - Mejor AdaBoost SVM
  - Mejor MLP
- Decisión por mayoría simple

10.0.3 3. Soft Voting

- Mismos clasificadores base que Hard Voting
- Ponderación basada en accuracy CV
- Decisión por probabilidades ponderadas

10.0.4 4. Stacking

- Clasificadores base: KNN, SVM, MLP
- Metaclasificador: Mejor MLP
- Validación cruzada con 5 folds

10.1 Pipeline de Entrenamiento y Evaluación

El proceso integra varias etapas:

1. **Preparación de Datos**:
   - Se utiliza la preparación inicial de datos pero sin aplicar Cross-Validation
2. **Selección de Mejores Modelos**:
   - Selección basada en accuracy y F1-score
3. **Entrenamiento y Evaluación**:
   - Entrenamiento con conjunto completo
   - Evaluación con test set reservado

[30]:
```
using Statistics: mean, std
```

[31]:
```
# Importar los modelos necesarios
@sk_import ensemble: (RandomForestClassifier, VotingClassifier,␣
 ↪StackingClassifier)
@sk_import ensemble: AdaBoostClassifier
@sk_import metrics: f1_score
```

```
WARNING: redefinition of constant Main.AdaBoostClassifier. This may fail, cause
incorrect answers, or produce other errors.
WARNING: redefinition of constant Main.f1_score. This may fail, cause incorrect
answers, or produce other errors.
```

```
PyObject <function f1_score at 0x76409dd12e80>
```

[32]:
```
function create_base_ensemble_models(X_train, y_train, stats, results_df,␣
↪best_models_overall)
    models = Dict()

    # Obtener los mejores modelos de cada tipo
    best_models_dict = Dict()
    cv_scores = Float64[]

    model_types = ["Bagging KNN", "AdaBoost SVM"]

    println("\nSeleccionando mejores modelos:")
    println("============================")

    for model_type in model_types
        # Filtrar modelos de este tipo
        type_models = filter(r -> r.ModelType == model_type, results_df)

        if !isempty(type_models)
            # Ordenar por precisión y tomar el mejor
            best_model_row = sort(type_models, :Max_Accuracy, rev=true)[1, :]
            model_name = best_model_row.Model

            # Guardar el mejor modelo y su score
            best_models_dict[model_type] = stats[model_name]["best_model"]
            push!(cv_scores, best_model_row.Max_Accuracy)

            println("Mejor $model_type: $model_name (Accuracy:␣
↪$(round(best_model_row.Max_Accuracy * 100, digits=2))%)")
        else
            @warn "No se encontraron modelos de tipo $model_type"
        end
    end

    # Añadir el mejor MLP desde best_models_overall
    if haskey(best_models_overall, "MLP")
        mlp_info = best_models_overall["MLP"]
        # Seleccionamos el modelo en la primera posición (max accuracy)
        best_models_dict["MLP"] = mlp_info["model"][1]
        push!(cv_scores, mlp_info["metrics"].Mean_Accuracy)
        println("Mejor MLP: $(mlp_info["name"]) (Accuracy:␣
↪$(round(mlp_info["metrics"].Mean_Accuracy * 100, digits=2))%)")
    end

    # Random Forest
    println("\nCreando Random Forest...")
```

```
    rf = RandomForestClassifier(
        n_estimators=500,
        max_depth=10,
        random_state=172
    )
    ScikitLearn.fit!(rf, X_train, y_train)
    models["random_forest"] = rf

    # Hard Voting
    println("Creando Hard Voting Classifier...")
    estimators = [
        ("bagging", best_models_dict["Bagging KNN"]),
        ("adaboost", best_models_dict["AdaBoost SVM"]),
        ("mlp", best_models_dict["MLP"])
    ]

    hard_voting = VotingClassifier(
        estimators=estimators,
        voting="hard"
    )
    ScikitLearn.fit!(hard_voting, X_train, y_train)
    models["hard_voting"] = hard_voting

    # Soft Voting
    println("Creando Soft Voting Classifier...")
    soft_voting = VotingClassifier(
        estimators=estimators,
        voting="soft",
        weights=cv_scores
    )
    ScikitLearn.fit!(soft_voting, X_train, y_train)
    models["soft_voting"] = soft_voting

    # Stacking
    println("Creando Stacking Classifier...")
    stacking = StackingClassifier(
        estimators=estimators,
        final_estimator=best_models_dict["MLP"],
        cv=5
    )
    ScikitLearn.fit!(stacking, X_train, y_train)
    models["stacking"] = stacking

    #= XGBoost
    println("\nCreando XGBoost...")
    try
        num_class = length(unique(y_train))
```

```
        y_train_xgb = Vector(y_train) .- 1
        bst = xgboost((X_train, y_train_xgb);
            num_round = 10,
            objective = "multi:softmax",
            num_class = num_class,
        )
        models["xgboost"] = bst
        println("XGBoost creado exitosamente")
    catch e
        println("Error creando XGBoost: ", e)
    end
    =#
    return models
end
```

```
create_base_ensemble_models (generic function with 1 method)
```

[40]:
```
# using XGBoost: predict as predict_xgb

function evaluate_ensemble_models(models, test_data)
    # Preparar datos de test
    X_test = Matrix{Float64}(select(test_data, Not([:subject, :Activity])))
    y_test = test_data.Activity
    results = Dict()

    println("\nEvaluación en conjunto de test (10% reservado):")
    println("================================================")

    for (name, model) in models
        println("Empezando evaluación de $name...")
        try
            # y_pred = if name == "xgboost"
            #    predict_xgb(model, X_test)
            # else
            y_pred = model.predict(X_test)
            # end

            # Obtener las metricas del modelo
            accuracy = mean(y_pred .== y_test)
            f1 = f1_score(y_test, y_pred, average="weighted")

            results[name] = Dict(
                "accuracy" => accuracy,
                "f1_score" => f1
            )

            println("$name:")
```

```
            println(" - Accuracy: $(round(accuracy * 100, digits=2))%")
            println(" - F1-Score: $(round(f1 * 100, digits=2))%")
        catch e
            println("Error evaluando $name: $e")
            println("Error completo: ")
            println(e)
        end
    end

    return results
end

function train_ensembles(train, test, stats, results_df)
    println("\nPreparando datos...")
    println("==================")

    # Comprobar el que los sets sean correctos
    println("Total de instancias de entrenamiento: $(size(train, 1))")
    println("Total de instancias de test: $(size(test, 1))")

    # Preparar datos de entrenamiento
    X_train = Matrix{Float64}(select(train, Not([:subject, :Activity])))
    y_train = train.Activity

    # Crear y entrenar modelos
    println("\nCreando y entrenando modelos ensemble...")
    println("=======================================")
    models = create_base_ensemble_models(X_train, y_train, stats, results_df,␣
 ↪best_models_overall)

    return models
end
```

```
train_ensembles (generic function with 1 method)
```

[34]:
```
function allDataPipeline(dataset::DataFrame)
    # División inicial train/test (10%) "individual-wise"
    train_data, test_data = HoldOut(dataset)

    # Codificación + Nulos
    train_data_prepared, test_data_prepared, _ = prepareData!(train_data,␣
 ↪test_data)

    # Aplicar MinMaxScaler
    applyMinMaxScaler!(train_data_prepared, test_data_prepared)

    return train_data_prepared, test_data_prepared
```

```
 end

train, test = allDataPipeline(dataset);

# Verificación
println("Train size: $(size(train))")
println("Test size: $(size(test))")
println("Total size: $(nrow(test) + nrow(train)) filas, $(ncol(train))␣
 ↪columnas")
```

```
=== Hold-Out Split Information ===
Individuos Totales: 30
Individuos Train: 27
Individuos Test : 3
Individuos en test set: [15, 19, 28]

Distribución:
Train set: 9229 instancias (89.61%)
Test set: 1070 instancias (10.39%)

=== Valores Nulos Iniciales ===
Train nulos iniciales: 247
Test nulos iniciales: 23

=== Check Final ===
Número de columnas arregladas: 131
Valores nulos en Train: 0
Valores nulos en Test: 0
Dataset limpio, no contiene nulos!
Train size: (9229, 563)
Test size: (1070, 563)
Total size: 10299 filas, 563 columnas
```

[35]:
```
models = train_ensembles(train, test, stats, results_df)
```

```
Preparando datos…
==================
Total de instancias de entrenamiento: 9229
Total de instancias de test: 1070

Creando y entrenando modelos ensemble…
======================================

Seleccionando mejores modelos:
============================
Mejor Bagging KNN: bagging_knn_50 (Accuracy: 92.46%)
```

```
Mejor AdaBoost SVM: adaboost_svm (Accuracy: 56.03%)
Mejor MLP: MLP_100_50 (Accuracy: 71.42%)

Creando Random Forest…
Creando Hard Voting Classifier…
Creando Soft Voting Classifier…
Creando Stacking Classifier…
```

```
Dict{Any, Any} with 4 entries:
  "hard_voting"   => PyObject VotingClassifier(estimators=[('bagging',…
  "random_forest" => PyObject RandomForestClassifier(max_depth=10, n_estimators…
  "soft_voting"   => PyObject VotingClassifier(estimators=[('bagging',…
  "stacking"      => PyObject StackingClassifier(cv=5,…
```

[41]:
```
results = evaluate_ensemble_models(models, test)
```

```
Evaluación en conjunto de test (10% reservado):
================================================
Empezando evaluación de hard_voting…
hard_voting:
 - Accuracy: 93.27%
 - F1-Score: 93.26%
Empezando evaluación de random_forest…
random_forest:
 - Accuracy: 92.15%
 - F1-Score: 91.94%
Empezando evaluación de soft_voting…
soft_voting:
 - Accuracy: 95.05%
 - F1-Score: 95.01%
Empezando evaluación de stacking…
stacking:
 - Accuracy: 96.45%
 - F1-Score: 96.41%
```

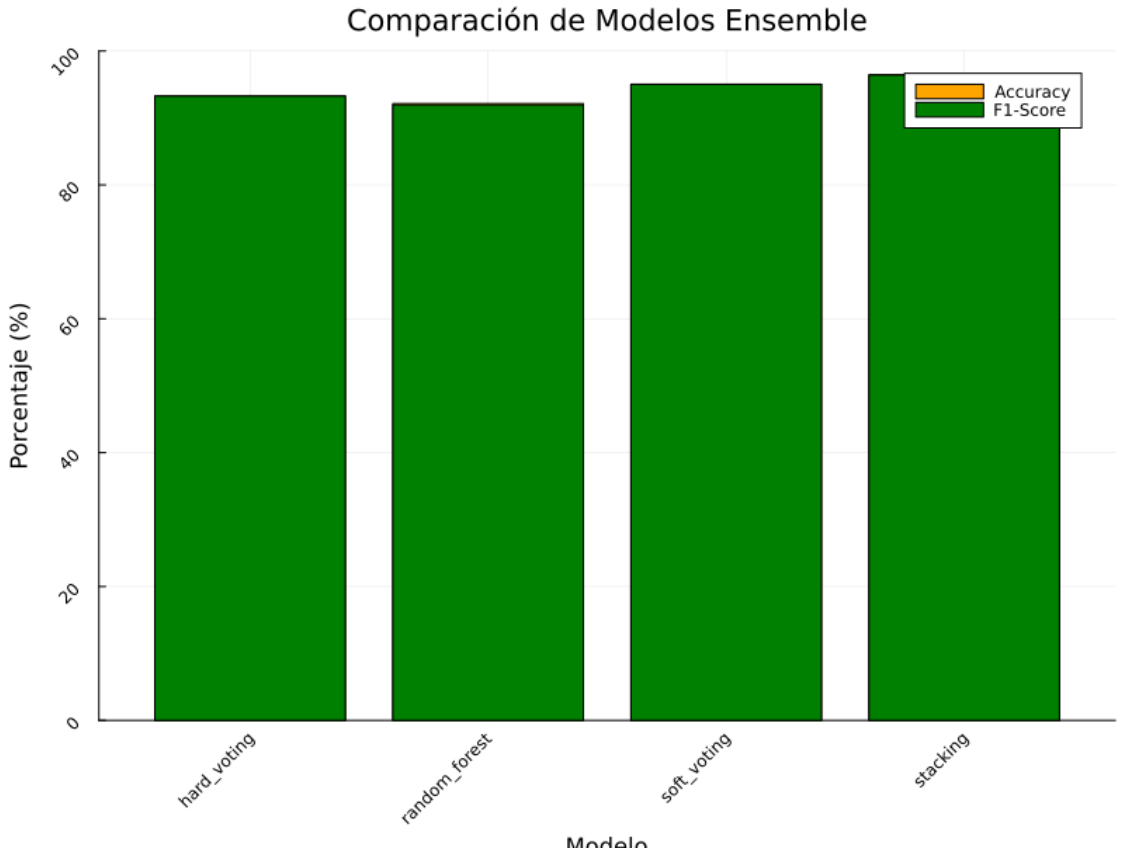
```
Dict{Any, Any} with 4 entries:
  "hard_voting"   => Dict("f1_score"=>0.932567, "accuracy"=>0.93271)
  "random_forest" => Dict("f1_score"=>0.919448, "accuracy"=>0.921495)
  "soft_voting"   => Dict("f1_score"=>0.950145, "accuracy"=>0.950467)
  "stacking"      => Dict("f1_score"=>0.964083, "accuracy"=>0.964486)
```

[45]:
```
function display_ensemble_results(results)
    println("\n=== MEJORES MODELOS ENSEMBLE ===")
    for (model_name, metrics) in results
        println("$model_name → " *
                "Acc: $(round(metrics["accuracy"]*100, digits=2))% " *
                "F1: $(round(metrics["f1_score"]*100, digits=2))%")
    end
```

```
end

function plot_ensemble_comparison(results)
    # Extraer nombres y métricas
    model_names = collect(keys(results))
    accuracies = [results[m]["accuracy"] * 100 for m in model_names]
    f1_scores = [results[m]["f1_score"] * 100 for m in model_names]

    # Graficar barras
    bar(
        model_names,
        [accuracies f1_scores],
        label = ["Accuracy" "F1-Score"],
        legend = :topright,
        bar_position = :dodge,
        c = [:orange :green],
        title = "Comparación de Modelos Ensemble",
        xlabel = "Modelo",
        ylabel = "Porcentaje (%)",
        ylims = (0, 100),
        rotation = 45,
        size = (800, 600)
    )
end

display_ensemble_results(results)
```

```
=== MEJORES MODELOS ENSEMBLE ===
hard_voting → Acc: 93.27% F1: 93.26%
random_forest → Acc: 92.15% F1: 91.94%
soft_voting → Acc: 95.05% F1: 95.01%
stacking → Acc: 96.45% F1: 96.41%
```

[46]:
```
p = plot_ensemble_comparison(results)
display(p)
```



10.2 Resultados y Análisis

1. **Stacking (96.45%)**:

- Mejor rendimiento global con 96.45% de accuracy
- Excelente generalización a través de todas las clases

2. **Soft Voting (95.05%)**:

- La ponderación mejora significativamente sobre el hard voting

3. **Hard Voting (93.27%)**:

- Demuestra el poder de la votación mayoritaria simple

4. **Random Forest (92.15%)**:

- Baseline robusto como modelo individual
- Excelente balance complejidad/rendimiento

La similitud entre accuracy y F1-Score indica un excelente balance en la clasificación de todas las clases, sin sesgo hacia ninguna actividad específica. Sin embargo, esto viene con diferentes niveles de complejidad:

- Stacking: Mayor complejidad computacional, pero ofrece el mejor rendimiento

- Soft Voting: Complejidad moderada, necesita estimaciones de probabilidad pero mantiene excelente rendimiento
- Hard Voting: Solo requiere predicciones de clase, manteniendo buen rendimiento
- Random Forest: Menor complejidad al ser modelo único, ofrece buen balance entre simplicidad y rendimiento

Esta relación entre complejidad y rendimiento permite elegir el modelo más apropiado según los requisitos que consideremos más importantes, ya sea priorizando precisión o eficiencia computacional.

11 Conclusiones

Se implementaron diversos modelos ensemble para la clasificación de actividades humanas. Sin embargo, debido a problemas técnicos, no se pudieron completar todas las secciones planeadas.

11.1 Limitaciones

1. **Problemas técnicos**:
   - **CatBoost**: Detenía el entorno de ejecución de Julia sin mostrar errores.
   - **XGBoost y LightGBM**: Congelaban el sistema operativo *Ubuntu 24.04.1 LTS* y el entorno *VSCode*, impidiendo su evaluación.
2. **Impacto en las secciones**:
   - En la **Sección 12**, solo se obtuvieron las variables importantes de **andom Forest**.
   - En la **Sección 13**, no se completó el contraste de hipótesis por la falta de resultados de XGBoost.

11.2 Importancia de Variables (Random Forest)

Las 10 variables más importantes seleccionadas por **Random Forest** fueron:

Variable	Importancia
angle(X,gravityMean)	0.0328
tGravityAcc-energy()-X	0.0309
tGravityAcc-min()-X	0.0296
tGravityAcc-max()-X	0.0292
tGravityAcc-mean()-X	0.0252
angle(Y,gravityMean)	0.0248
tGravityAcc-mean()-Y	0.0221
tGravityAcc-max()-Y	0.0202
tGravityAcc-min()-Y	0.0199
tGravityAcc-energy()-Y	0.0144

Estas variables destacan aspectos relacionados con la gravedad como las más relevantes para la clasificación. Esto podría reflejar las diferencias más significativas entre las posiciones estáticas, como estar sentado o tumbado, y las de movimiento, como estar de pie o andando. ## Resumen Final Aunque no se completaron las secciones 12 y 13, se lograron resultados sólidos con los modelos implementados.

[49]:
```
function print_feature_importances(rf_model, feature_names::Vector{String})

    # Obtener importancias del Random Forest
    rf_importances = rf_model.feature_importances_

    rf_df = DataFrame(
        feature = feature_names,
        importance = rf_importances
    )

    # Ordenar por importancia descendente
    sort!(rf_df, :importance, rev=true)

    println("\n=== Random Forest Feature Importances ===")
    println("\nTop 10 variables más importantes:")
    println("-" ^ 40)

    for i in 1:10
        println("$(rf_df.feature[i]): $(round(rf_df.importance[i], digits=4))")
    end

    return rf_df
end

feature_names = names(select(train, Not([:subject, :Activity])))
# Se puede eliminar el ";" si se quiere ver la importancia de las 561 variables
importance_df = print_feature_importances(models["random_forest"],␣
 ↪feature_names);
```

```
=== Random Forest Feature Importances ===

Top 10 variables más importantes:
----------------------------------------
angle(X,gravityMean): 0.0328
tGravityAcc-energy()-X: 0.0309
tGravityAcc-min()-X: 0.0296
tGravityAcc-max()-X: 0.0292
tGravityAcc-mean()-X: 0.0252
angle(Y,gravityMean): 0.0248
tGravityAcc-mean()-Y: 0.0221
tGravityAcc-max()-Y: 0.0202
tGravityAcc-min()-Y: 0.0199
tGravityAcc-energy()-Y: 0.0144
```