UNIVERSIDAD DE GRANADA
Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería Informática
Trabajo Fin de Grado
Francisco Manuel García Moreno

Análisis de datos a partir de sensores de dispositivos *wearables*

1. Sampling Rate

Cuando recopilamos datos en un *wearable*, podemos hacerlo desde distintos sensores que tengamos disponibles. Además, podemos estar interesados en realizarlo más rápido o más lento en algunos de ellos, según nos convenga. Esto es a lo que se le conoce como «sampling rate», que representa la frecuencia a la ponemos en marcha el sensor a funcionar. La frecuencia se expresa en hercios (Hz), pero es habitual usar segundos (segundos=1/frecuencia en Hz).

2. Archivo CSV

Generaremos un archivo CSV (valores separados por comas), donde cada valor de los sensores (los datos en *bruto —raw data—*), y otros datos de utilidad que consideremos, estarán separados por comas tal como se muestra en Figura 1. Necesitaremos un campo especial que servirá como identificador (índice) de cada una de las filas del *dataset* (conjunto de datos) que vamos a generar. Dicho campo es de tipo *datetime* y estará formado por la fecha y hora en formato anglosajón:

YYYY-MM-DD HH:MM:SS.MS (atentos al punto detrás de los segundos; MS son milisegundos)

timestamp	gyr.x	gyr.y	gyr.z	linacc.x	linacc.y	linacc.z
2019-05-06 19:16:47.180	32.900002	17.15	-31.43	-0.380781	-2.981965	-5.608761
2019-05-06 19:16:47.200	33.8730019	22.448999899999997	-33.103	-0.44153812499999995	-2.600556875	-4.565442
2019-05-06 19:16:47.220	34.8460018	27.7479998	-34.775999999999996	-0.50229525	-2.21914875	-3.522123
2019-05-06 19:16:47.240	35.8190017	33.0469997	-36.449	-0.5630523749999999	-1.8377406250000001	-2.4788040000000002
2019-05-06 19:16:47.260	36.7920016	38.3459996	-38.122	-0.6238094999999999	-1.4563325	-1.435485
2019-05-06 19:16:47.280	37.7650015	43.6449995	-39.795	-0.684566625	-1.074924375	-0.39216599999999957

Figura 1. Ejemplo de archivo CSV con los datos en bruto de diferentes sensores de dispositivos wearables

2.1. Lectura de archivos CSV en Python

Usaremos la biblioteca *Pandas* para trabajar con CSVs y *Numpy* para trabajar con *arrays* matriciales. *Pandas* crea un *data.frame* con los datos leídos del CSV. El campo *header=0* indica que la cabecera es la fila 0 del CSV, así le establece nombres a cada columna en función a los datos de dicha fila.

```
import pandas as pd
import numpy as np

#cargamos en un dataframe nuestro CSV y lo indexamos por la columna timestamp parseando las dates
df = pd.read_csv("my_archivo.csv", header=0,index_col="timestamp", parse_dates=True)
```

Podemos encontrar problemas con el índice «timestamp» si no se ha generado siguiendo el formato adecuado.

3. Resampling

Con objeto de alinear los diferentes datos recopilados, provenientes de diferentes sensores trabajando a distinta frecuencia, es necesario realizar un alineamiento o *resampling* de los datos, para tenerlos alineados en función del tiempo y, así, poder realizar un análisis posterior. Aunque pongamos a funcionar varios sensores al mismo *sampling rate*, puede ser que alguno de ellos empiece antes que otro, por tanto, siempre será necesario este *resampling*.

Para realizar el *resampling*, usaremos el siguiente código:

```
#--------------------------------------------------------------------------------------
# df: dataset cargado previamente con Pandas
# mResample="0.04S". (string) valor del resampling. La "S" indica segundos.
# 0.04 segundos equivale a 1/0.04 = 25 Hz la frcuencia del resampling
# verbose. (bool) True si se quiere mostrar mensajes informativosd
#--------------------------------------------------------------------------------------
def perform_resample(df, mResample="0.04S", verbose=False):
    df = df.resample(mResample).mean()
    nas = sum(df.isna().sum())
    if nas > 0:
        if verbose:
            print("NAs after resample (" + mResample + "): " + str(nas))
        df = df.interpolate()
    return df

d_resampled = perform_resample(df, '0.04S')
```

Téngase en cuenta que después de realizar el *resampling* hay que calcular la media de los valores y si se generan valores perdidos (*missing values o* también conocidos como *NA*), tendremos que interpolar.

4. Sliding and Overlapping Windows

Una vez que tenemos los datos a la misma frecuencia de muestreo, una técnica común que suele realizarse para procesar los datos es «trocear» en ventanas de cierto tamaño los datos, es decir, se irán tomando cierto número de filas del *dataset* extrayendo nuevas variables por cada ventana (como la media, mediana, desviación típica, valor mínimo, valor máximo, amplitud, etc.) para, finalmente, comprimir todas las ventanas en un único valor que se calculará realizando la media (Figura 2).

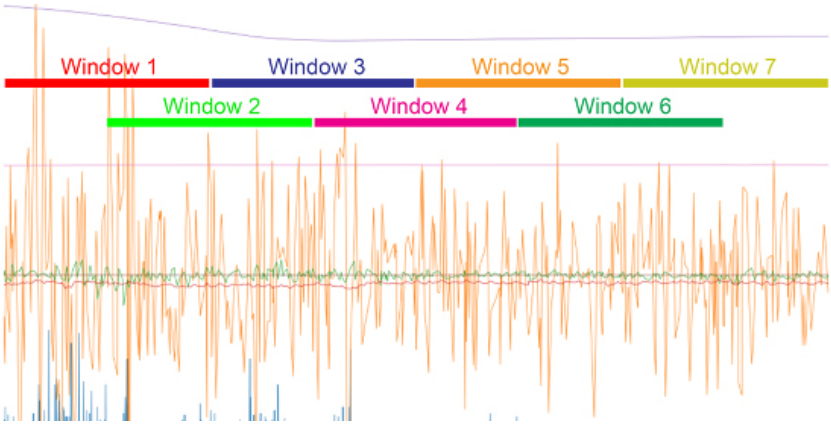

Figura 2. Sliding and 50% overlapping Window

El tamaño de la ventana es habitual establecerlo en segundos, y calcular cuántas filas (*samples:* las muestras) se corresponden. Para ello, haremos uso del índice de nuestro *dataset* usando el atributo «freq.nanos», y calcularemos los *samples* por ventana.

```
w = 0.5 #window size in seconds
hz = 1 / (df.index.freq.nanos / 10 ** 9)  # in Hz
w_samples_n = int(hz * w)
```

Usaremos la función de utilidad siguiente para calcular los índices de inicio y fin de la ventana teniendo en cuenta que queremos usar un 50% de *overlapping*:

```
#-------------------------------------------------------------------------------
# d=dataset
# w=windows samples number
# overlapping (default: 50%)
# Devuelve una lista que representa las diferentes ventanas encontradas.
# Cada ventana es una tupla (x, y), donde x es el índice de la fila inicial;
# y es el índice de la final final de la ventana
#-------------------------------------------------------------------------------
def sliding_windows_with_50_perc_overlapping(d, w=0.5):
    t = int(w / 2)
    r = np.arange(len(d))
    s = r[::t]
    z = list(zip(s, s + w))  # indices de las windows
    fz = list()
    ld = len(d)
    for z1, z2 in z:
        if z1 > ld or z2 > ld:  # cortamos los que sobrepasen
            break
        fz.append((z1, z2))
    return fz
```

Y para generar una lista de *dataframes* por cada una de las ventanas usaremos el siguiente código:

```
#-------------------------------------------------------------------------------
# d=dataset
# w=windows samples number
# overlapping (default: 50%)
# Devuelve: una lista con los dataframes por ventana preparados para trabajar
#-------------------------------------------------------------------------------

def sliding_windows_with_50_perc_overlapping_splitting_events(d, w=0.5):
    fz = list()
    df_list = []
    df_final = None

    fz = sliding_windows_with_50_perc_overlapping(d, w)
    for w_i in fz: # fz: devuelve
        df_list.append(d[w_i[0]:w_i[1]])
    return df_list
```

Tendremos una lista de *dataframes* con cada una de las ventanas preparada para extraer nuevas variables de cada columna:

```
df_w = sliding_windows_with_50_perc_overlapping_splitting_events(df, w_samples_n)
```

5. Normalización de los datos

Es habitual realizar una normalización o *reescalado* de los datos, por ejemplo con *MinMaxScaler*.

```
from sklearn.preprocessing import MinMaxScaler

def rescale_data(X):
    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(X)
    return X_scaled
```

6. Feature Extraction

Se trata de una técnica de preprocesamiento de datos habitual para realizar ingeniería de características (*feature engineering*), es decir, crear nuevas variables (*features*) a partir de los datos *en bruto* (*raw data*) de los sensores, como pueden ser: media, desviación típica, mínimo, máximo, diferencia máximo-mínimo, skewnes, kurtosis, etc.

Con la librería *Pandas* es muy fácil extraer *features* de origen estadístico:

```
dataset.mean()                  #media de cada columna
dataset.median()                #mediana de cada columna
dataset.std                     #desviación típica de cada columna
dataset.var()                   #varianza de cada columna
dataset.min()                   #valor mínimo de cada columna
dataset.max()                   #valor máximo de cada columna
dataset.max()-dataset.min()     #rango de cada columna
dataset.mean()                  #media de cada columna
dataset.mean()                  #media de cada columna
dataset.skew()                  #skewness de cada columna
dataset.kurt()                  #kurtosis de cada columna
```

Por supuesto, se pueden crear nuevas variables como deseemos, con fórmulas propia, a partir de ecuaciones…

Además, como nuestros datos, por su naturaleza, son señales, también podemos utilizar el paquete de *Python Scipy* para tratamiento de señales y, así, poder extraer por ejemplo las *features* siguientes:

- Power Spectral Density (PSD).
- Amplitude Spectral Density (ASD).
- Energy a partir de la Fast Fourier Transform (FFT).

Por ejemplo, el siguiente código extrae PSD y ASD:

```
from scipy import signal
import  numpy as np

freqs, psd_col   = signal.welch(dataset[column_name])      #PSD de una columna específica
asd_col          = np.sqrt(psd_col)                        #ASD de una columna específica
```

7. Feature Selection

Después de realizar el *feature extraction*, se incrimenta el número de variables de nuestro conjunto de datos. Sin embargo, no sabemos a priori cuáles son las más relevantes. Seguramente, muchas resulten ser redundantes y no aporten valor predictivo. Además, para lidiar con el problema clásico de la «maldición de la dimensionalidad» (*the curse of dimensionality*).

Para ello, usaremos la librería *ScikitLearn* para *Python*.

Es habitual conocer los tres tipos de métodos de *feature selection*: *filter*, *wrapper* y *embedded*.

En nuestro caso, según nuestra experiencia, usaremos *embedded* en combinación con el algoritmo *Recurrent Feature Elimination* (*RFE*) para encontrar el mejor subconjunto de *features* y *clasificador*.

Enlace a un tutorial: https://towardsdatascience.com/feature-selection-with-pandas-e3690ad8504b

Por ejemplo, podríamos probar diferentes algoritmos de *feature selection* con RFE:

```
def get_models(model=DecisionTreeClassifier(), model_type="knn", feat_sel_estimators=["rf", "lr",
"per", "dt", "gbm"]):
    models = dict()
    for estimator in feat_sel_estimators:
        if estimator == "rf":
            rfe = RFECV(estimator=RandomForestClassifier(n_estimators=100))
        elif estimator == "lr":
            rfe = RFECV(estimator=LogisticRegression(solver="lbfgs", max_iter=1000))
        elif estimator == "per": # perceptron
            rfe = RFECV(estimator=Perceptron(max_iter=1000, tol=1e-3))
        elif estimator == "dt":
            rfe = RFECV(estimator=DecisionTreeClassifier())
        else: #gbm
            rfe = RFECV(estimator=GradientBoostingClassifier())
        models[model_type + '-RFE_'+estimator] = Pipeline(steps=[('s', rfe), ('m', model)])
    return models
```

8. ***Ajuste de parámetros de los clasificadores (hyperparameter tuning)***

Con todo ello, usaremos el *GridSearchCV*, para encontrar la mejor configuración de los valores de los parámetros de cada clasificador que usemos.

Por ejemplo, podríamos preparar una función tal como la siguiente, preparada para hacer un *tuning* de los clasificadores clásicos *k-NN* y *Random Forests*:

```
from sklearn.model_selection import GridSearchCV

def tuning_model(X, model_type="knn", kfold=5):
    if model_type == "knn":
        model = KNeighborsClassifier()
        k_range = list(range(1, math.floor(math.sqrt(X.shape[0])), 2))
        weight_options = ["uniform", "distance"]
        distances = ["minkowski", "manhattan", "euclidean"]
        param_grid = dict(n_neighbors=k_range, weights=weight_options, metric=distances)
    elif model_type == "rf":
        model = RandomForestClassifier()
        n_estimators = [100, 300, 500, 800, 1200]
        max_depth = [5, 8, 15, 25, 30]
        min_samples_split = [2, 5, 10, 15, 100]
        min_samples_leaf = [1, 2, 5, 10]
        param_grid = dict(n_estimators=n_estimators, max_depth=max_depth,
                          min_samples_split=min_samples_split,
                          min_samples_leaf=min_samples_leaf)
    elif model_type == "lr": #logistic regression
        model = LogisticRegression(solver="lbfgs")
        param_grid = {"max_iter": [100, 1000, 2000], "C": np.logspace(-3, 3, 7), "penalty": ["l1",
"l2"]}  # l1 lasso l2 ridge
    elif model_type == "per": #perceptron
        model = Perceptron(tol=1e-3)
        param_grid = dict(max_iter=[100, 1000, 2000], kernel_pca__gamma=2. ** np.arange(-2, 2),
rceptron__n_iter=np.arange(1, 5))
    grid = GridSearchCV(model, param_grid, cv=kfold, scoring='accuracy', iid=False)

    return grid
```

9. Evaluar el modelo predictivo final

La evaluación de un modelo suele realizarse con la técnica de *stratified k-fold cross-validation* (*k-fold* validación cruza con estratificación). La *estraficación* consiste en considerar la misma proporción de datos de cada una de las clases (etiquetas) de nuestro conjunto de datos, haciendo la evaluación más equitativa.

```
# evaluate a given model using cross-validation
def evaluate_model(model, X, y, kfold=5, metric="accuracy"):
    np.random.seed(33)  # establecemos la semilla para los aleatorios y poder repetir el
experimento
    cv = StratifiedKFold(n_splits=kfold, random_state=33)
    scores = cross_val_score(model, X, y, scoring=metric, cv=cv, n_jobs=-1)
    return scores

def evaluate_several_models(models, X, y, kfold=5, metric="accuracy", verbose=True):
    results, names = list(), list()
    for name, model in models.items():
        scores = evaluate_model(model, X, y, kfold=kfold, metric=metric)
        results.append(scores)
        names.append(name)

        if verbose:
            print(">>>> Evaluating model:", name)
            print('>%s Accuracy: %.3f (SD: %.3f)' % (name, mean(scores), std(scores)))
    return results, names
```

Para terminar, mostramos un código de ejemplo completo donde se hace un *tuning*, *feature selection* con *RFE* y, finalmente, se evalúa el/los modelo/modelos que consideremos. En este caso, se evalúa sólo a *k-NN*. *X_train* son los datos e *y_train* son las etiquetas o clases de esos datos (por ejemplo, 0: estrés; 1: relajado).

```
kfold = 5
X_scaled = ml.rescale_data(X_train)
for m in ["knn"]: #lista de modelos a evaluar
    print("Creando modelo:", m, "con Diferentes algoritmos para feature selection con RFE")
    grid = ml.tuning_model(X_train, model_type=m, kfold=kfold)
    models = ml.get_models(model=grid, model_type=m, feat_sel_estimators=["rf", "lr"])
    scores, names = ml.evaluate_several_models(models, X_scaled, y_train, kfold=kfold,
metric="accuracy")
```