Anexos

Apéndice A

Código Python

En este anexo se muestra parte del código implementado para la creación y optimización de los modelos, empleando el lenguaje Python.

A.1. Creación de las matrices

Listing A.1: Creación de una matriz con todas las frases de Financial Phrasebank, donde se le asocia su sentimiento y el grado de acuerdo.

```
import numpy as np
from pathlib import Path

dir_frases = Path.cwd()/'FinBERT'/'FinancialPhraseBank-v1.0'


frases50 = []
clases50 = []
frases66 = []
clases66 = []
frases75 = []
clases75 = []
frases100 = []
clases100 = []


with open(dir_frases/'Sentences_50AgreeUTF8.txt', 'r') as file:
    for linea in file:
        linea = linea.strip()
        frase, clase = linea.rsplit('@', 1)
        frases50.append(frase)
        clases50.append(clase)

with open(dir_frases/'Sentences_66AgreeUTF8.txt', 'r') as file:
    for linea in file:
        linea = linea.strip()
        frase, clase = linea.rsplit('@', 1)
        frases66.append(frase)
        clases66.append(clase)
```

```
with open(dir_frases/'Sentences_75AgreeUTF8.txt', 'r') as file:
    for linea in file:
        linea = linea.strip()
        frase, clase = linea.rsplit('@', 1)
        frases75.append(frase)
        clases75.append(clase)

with open(dir_frases/'Sentences_AllAgreeUTF8.txt', 'r') as file:
    for linea in file:
        linea = linea.strip()
        frase, clase = linea.rsplit('@', 1)
        frases100.append(frase)
        clases100.append(clase)


# Crear un diccionario para almacenar las frases y las clases
datos = {}
for frase, clase in zip(frases50, clases50):
    datos[frase] = clase



frases50 = [frase for frase in frases50 if frase not in frases66]
clases50 = []
for frase in frases50:
    clases50.append(datos[frase])
datos50 = np.array([frases50, clases50, [50]*len(frases50)]).T


frases66 = [frase for frase in frases66 if frase not in frases75]
clases66 = []
for frase in frases66:
    clases66.append(datos[frase])
datos66 = np.array([frases66, clases66, [66]*len(frases66)]).T


frases75 = [frase for frase in frases75 if frase not in frases100]
clases75 = []
for frase in frases75:
    clases75.append(datos[frase])
datos75 = np.array([frases75, clases75, [75]*len(frases75)]).T


datos100 = np.array(list(set(zip(frases100, clases100))))
exactitud = np.full((datos100.shape[0], 1), 100)
datos100 = np.concatenate((datos100, exactitud), axis=1)


# Crear la matriz y guardarla
frases = np.vstack((datos50, datos66, datos75, datos100))
np.savetxt('frase_sentimiento_fiabilidad.csv', frases, delimiter='@', fmt='%s')
```

Listing A.2: Creación de la matriz de coordenadas de los modelos FinBERT y BERT.

```
import numpy as np
import torch
from transformers import BertTokenizer, BertModel
import pickle

# Importar matriz de frases Financial PhraseBank
frase_sentimiento_fiabilidad = np.genfromtxt('frase_sentimiento_fiabilidad.csv',
    delimiter='@', dtype=str)
# Crear matriz con frases con el 100% de acuerdo
frases100agree = frase_sentimiento_fiabilidad[frase_sentimiento_fiabilidad[:, 2]
                                              == '100']

# Importar matriz de tuits
tweets_sentimiento = np.genfromtxt('tweet_sentiment.csv', delimiter=',',
                                   dtype=str, skip_header=True)

# Importar el modelo BERT base
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased',
                                  output_hidden_states = True)

# Importar el modelo FinBERT del que se parte
finbert_model = pickle.load(open("FinBERT_entrenado.pkl", 'rb'))


def coordenadas_F(frase):
    token_ids = tokenizer(frase)['input_ids']
    with torch.no_grad():
        embeddings = finbert_model(torch.tensor([token_ids]),
                                   output_hidden_states = True)
    return embeddings.hidden_states[-1][0,0,:]

mc_FinBERT = torch.empty((frase_sentimiento_fiabilidad.shape[0], 768))
for i in range(frase_sentimiento_fiabilidad.shape[0]):
    rf = coordenadas_F(frase_sentimiento_fiabilidad[i][0])
    mc_FinBERT[i] = rf.squeeze()
np.savetxt('mc_FinBERT.csv', mc_FinBERT, delimiter=',' )


def coordenadas_B(frase):
    token_ids = tokenizerB(frase)['input_ids']
    with torch.no_grad():
        embeddings = modelB(torch.tensor([token_ids]))
    return embeddings.last_hidden_state[0,0,:]

mc_BERT = torch.empty((frase_sentimiento_fiabilidad.shape[0], 768))
for i in range(frase_sentimiento_fiabilidad.shape[0]):
    rf = coordenadas_B(frase_sentimiento_fiabilidad[i][0])
    mc_BERT[i] = rf.squeeze()
np.savetxt('mc_BERT.csv', mc_BERT, delimiter=',')
```

A.2. Matrices PCA

Listing A.3: Creación de las matrices de PCA para los modelos FinBERT y BERT.

```
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

# Importar las matrices creadas
frase_sentimiento_fiabilidad = np.genfromtxt('frase_sentimiento_fiabilidad.csv',
    delimiter='@', dtype=str)
mc_FinBERT = np.genfromtxt('mc_FinBERT.csv', delimiter=',', dtype=float)
mc_BERT = np.genfromtxt('mc_BERT.csv', delimiter=',', dtype=float)

# Creación PCA con todas las componentes principales
pca = PCA(n_components=768)
matrizPCA = pca.fit_transform(mc_FinBERT) #4838x768
# Varianza explicada acumulada
energia = np.cumsum(pca.explained_variance_)/np.sum(pca.explained_variance_)*100

# Número de componentes principales para una energía dada
np67 = np.sum(energia<67)
np80 = np.sum(energia<80)
np85 = np.sum(energia<85)
np90 = np.sum(energia<90)
np95 = np.sum(energia<95)

# Graficar la energía en función de las componentes principales
plt.plot(energia)
plt.xlabel('Número de Componentes Principales')
plt.ylabel('Energía acumulada %')
valores_energia = [67, 80, 85, 90, 95]
indices_valores = [np67, np80, np85, np90, np95]
for i, valor in enumerate(valores_energia):
    plt.scatter(indices_valores[i], valor, c='red', label=f'{valor}%' )
plt.show()

# Crear las matrices PCA en función de la varianza explicada de sus componentes
x67 = matrizPCA[:,range(0,np67)]
x80 = matrizPCA[:,range(0,np80)]
x85 = matrizPCA[:,range(0,np85)]
x90 = matrizPCA[:,range(0,np90)]
x95 = matrizPCA[:,range(0,np95)]

# Guardar las matrices
np.savetxt('x67_FinBERT.csv', x67, delimiter=',')
np.savetxt('x80_FinBERT.csv', x80, delimiter=',')
np.savetxt('x85_FinBERT.csv', x85, delimiter=',')
np.savetxt('x90_FinBERT.csv', x90, delimiter=',')
np.savetxt('x95_FinBERT.csv', x95, delimiter=',')
```

A.3. Creación de los clasificadores

Listing A.4: Función que calcula las métricas y las muestra por pantalla.

```
from sklearn.metrics import precision_score, recall_score, f1_score,
    confusion_matrix
from tabulate import tabulate

def Metricas(clasesreales, clasespredichas):
    # Etiquetas de los niveles
    labels = [2, 1, 0]
    nombre_labels = ['positive', 'neutral', 'negative']
    nombre_labels2 = ['Positivo', 'Neutral', 'Negativo', 'Media',
                      'Media ponderada']

    # Calcular la precisión, recall, F1-score y matriz de confusión
    precision = precision_score(clasesreales, clasespredichas, labels=labels,
                                average=None)
    recall = recall_score(clasesreales, clasespredichas, labels=labels,
                          average=None)
    f1 = f1_score(clasesreales, clasespredichas, labels=labels, average=None)
    num_elementos = np.array([np.count_nonzero(y_test == 2), np.count_nonzero(
        y_test == 1), np.count_nonzero(y_test == 0), y_test.shape[0], y_test.
        shape[0]])
    cm_valores = confusion_matrix(clasesreales, clasespredichas, labels=labels)

    # Se agregan la media y la media ponderada de cada métrica
    media = (precision[0] + precision[1] + precision[2])/3
    media_ponderada = (precision[0]*num_elementos[0] + precision[1]*
        num_elementos[1] + precision[2]*num_elementos[2])/y_test.shape[0]
    precision = np.round(np.append(precision, [media, media_ponderada]),2)

    media = (recall[0] + recall[1] + recall[2])/3
    media_ponderada = (recall[0]*num_elementos[0] + recall[1]*num_elementos[1] +
         recall[2]*num_elementos[2])/y_test.shape[0]
    recall = np.round(np.append(recall, [media, media_ponderada]),2)

    media = (f1[0] + f1[1] + f1[2])/3
    media_ponderada = (f1[0]*num_elementos[0] + f1[1]*num_elementos[1] + f1[2]*
        num_elementos[2])/y_test.shape[0]
    f1 = np.round(np.append(f1, [media, media_ponderada]),2)


    # Crear una lista para mostrar la precision, recall y f1_score
    resultados = []
    for p, r, f, n in zip(precision, recall, f1, num_elementos):
        resultados.append([p, r, f, n])

    # Imprimir la matriz de confusión
    print(tabulate(cm_valores, headers=['Real\Predicho', 'positive', 'neutral',
        'negative'], showindex=nombre_labels, tablefmt='fancy_grid'))
```

```
    # Calcular la precisión y la pérdida de entropía cruzada y mostrarlos
    accuracy = (cm_valores[0][0]+cm_valores[1][1]+cm_valores[2][2]) / (len(
        clasespredichas))
    print("\n - Accuracy: ", accuracy)

    # Imprimir los resultados utilizando tabulate
    headers = ['Precision', 'Recall', 'F1 Score', 'Elementos']
    print(tabulate(resultados, headers=headers, showindex=nombre_labels2,
                   tablefmt='fancy_grid'))
```

Listing A.5: Cálculo de la perdida de entropía cruzada ponderada.

```
import torch
import torch.nn as nn
import numpy as np

# Calcula la raíz cuadrada de la tasa inversa de la frecuencia: los pesos
pesos = np.sqrt(len(y_test) / np.bincount(y_test))

# Define la función de pérdida con pesos ponderados, en un tensor
loss = nn.CrossEntropyLoss(weight=torch.tensor(pesos, dtype=torch.float32))

# Pérdida de entropía cruzada ponderada
cel = loss(torch.tensor(xgb.predict_proba(X_test), dtype=torch.float32), torch.
    tensor(y_test, dtype=torch.int64))
```

Listing A.6: Importar paquetes para la creación de clasificadores.

```
import time
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Transformar sentimientos en: positive=2; neutral=1; negative=0
Ly = LabelEncoder()
sentimientos_num = Ly.fit_transform(sentimientos)
```

Listing A.7: Clasificador XGBoost con GridSearchCV.

```
import xgboost
from sklearn.model_selection import GridSearchCV

# Preparar la base de datos con conjunto de prueba y entrenamiento
X_train, X_test, y_train, y_test = train_test_split(matriz, sentimientos_num,
                                                    test_size=0.2, random_state=123)

ini = time.time()

# Crear el clasificador
xgb = xgboost.XGBClassifier(sampling_method='gradient_based')

# Tuneado de hiperparámetros
parametros = { 'objective': ['multi:softmax'], 'num_class': [3],
               'learning_rate': [0.1, 0.01],
               'gamma': [0.3, 0.5],
               'n_estimators': [100] }
clas_xgb = GridSearchCV(xgb, parametros, cv=3, scoring='accuracy')

clas_xgb.fit(X_train, y_train)

# Mostrar el mejor modelo con los mejores hiperparámetros
xgb_gs = clas_xgb.best_estimator_
print(xgb_gs)

# Obtener las predicciones, las métricas y el tiempo empleado
pred_xgb = xgb_gs.predict(X_test)
Metricas(y_test, pred_xgb)
tiempo = time.time() - ini

# Guardadar el modelo
pickle.dump(xgb_gs, open('xgb_gs.pkl', 'wb'))
```

Listing A.8: Clasificador XGBoost con RandomizedSearchCV.

```
import xgboost
from sklearn.model_selection import RandomizedSearchCV

# Preparar la base de datos con conjunto de prueba y entrenamiento
X_train, X_test, y_train, y_test = train_test_split(matriz, sentimientos_num,
                                                    test_size=0.2, random_state=123)

ini = time.time()

# Crear el clasificador
xgb = xgboost.XGBClassifier(sampling_method='gradient_based')

# Tuneado de hiperparámetros
parametros = { 'objective': ['multi:softmax'], 'num_class': [3],
               'learning_rate': [0.1, 0.01, 0.001],
               'gamma': [0.1, 0.3, 0.5, 0.7],
               'n_estimators': [50, 100],
               'max_depth': [3, 5, 6, 7] }
clas_xgb = RandomizedSearchCV(xgb, parametros, cv=3, scoring='accuracy',
                              n_iter=10)

clas_xgb.fit(X_train, y_train)

# Mostrar el mejor modelo con los mejores hiperparámetros
xgb_rs = clas_xgb.best_estimator_
print(xgb_rs)

# Obtener las predicciones, las métricas y el tiempo empleado
pred_xgb = xgb_rs.predict(X_test)
Metricas(y_test, pred_xgb)
tiempo = time.time() - ini

# Guardar el modelo
pickle.dump(xgb_rs, open('xgb_rs.pkl', 'wb'))
```

Listing A.9: Clasificador con algoritmo k-NN, analizando el *k* óptimo.

```
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

# Preparar la base de datos con conjunto de prueba y entrenamiento
X_train, X_test, y_train, y_test = train_test_split(matriz, sentimientos_num,
                                                    test_size=0.2, random_state=123)

# Analizamos que valor de k es el optimo y lo graficamos
k_range = range(1, 20)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_train, y_train)
    scores.append(knn.score(X_test, y_test))
plt.figure()
plt.xlabel( 'k' )
plt.ylabel( 'accuracy' )
plt.scatter(k_range, scores)
plt.xticks([0,5,10,15,20])

nvecinos_opt = scores.index(max(scores))+1
plt.scatter(nvecinos_opt, max(scores), color='red', marker='o')
```

Listing A.10: Clasificador con algoritmo k-NN.

```
n_neighbors = nvecinos_opt
ini = time.time()

# Crear el clasificador
knn = KNeighborsClassifier(n_neighbors)
knn.fit(X_train, y_train)

# Obtener las predicciones, las métricas y el tiempo empleado
pred_knn = knn.predict(X_test)
Metricas(y_test, pred_knn)
tiempo = time.time() - ini

# Guardar el modelo
pickle.dump(knn, open('kNN.pkl', 'wb'))
```

Listing A.11: Clasificador con red neuronal.

```
import tensorflow as tf

# Preparar la base de datos con conjunto de prueba y entrenamiento
X_train, X_test, y_train, y_test = train_test_split(matriz, sentimientos_num,
                                                    test_size=0.2, random_state=123)

ini = time.time()

# Crear el modelo y entrenar
red_neuronal = tf.keras.Sequential([
    tf.keras.layers.Dense(100, input_shape=[matriz.shape[1],]),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(3, activation=tf.nn.softmax),])

red_neuronal.compile(
    optimizer = 'adam',
    loss = tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)

historial = red_neuronal.fit(X_train, y_train, epochs=10)

# Obtener las predicciones, las métricas y el tiempo empleado
pred_nn_ptos = red_neuronal.predict(X_test)
pred_nn = np.zeros(pred_nn_ptos.shape[0])
for i in range(pred_nn_ptos.shape[0]):
    pred_nn[i] = np.argmax(pred_nn_ptos[i])
Metricas(y_test, pred_nn)
tiempo = time.time() - ini

# Guardar el modelo
pickle.dump(red_neuronal, open('neuralnet.pkl', 'wb'))
```

A.4. Implementación del algoritmo RR-PSO

Listing A.12: Algoritmo RR-PSO, construcción del conjunto inicial.

```
def conjunto_inicial(N, par_min, par_max):
    return par_min * np.ones((N,len(par_min))) + (par_max-par_min) *
           np.random.rand(N,len(par_min))
```

Listing A.13: Algoritmo RR-PSO, construcción de la función objetivo.

```
def eval_conjunto(conjunto, X_train, X_test, y_train, y_test):
    acc = []
    for i in range(len(conjunto)):
        print(f"P{i+1} \t", end="")
        xgb = xgboost.XGBClassifier(objective=['multi:softmax'], num_class=[3],
                n_estimators=int(conjunto[i][0]), max_depth=int(conjunto[i][1]),
                learning_rate=conjunto[i][2], gamma=conjunto[i][3],
                min_child_weight=conjunto[i][4], subsample=conjunto[i][5],
                colsample_bytree=conjunto[i][6])
        xgb.fit(X_train, y_train)
        y_pred = xgb.predict(X_test)
        accuracy = np.round(float(np.sum(y_pred==y_test))/y_test.shape[0]*100,2)
        acc.append(accuracy)
    return (np.array(acc))
```

Listing A.14: Algoritmo RR-PSO, construcción de la función posición y velocidad.

```
def velocidad_RR_GPSO(x,v,w,Dt,phi1,phi2,g,l):
    v_particulas = []
    for i in range(len(phi1)):
        v_particulas.append((phi1[i]*(g-x[i])*Dt+phi2[i]*(l[i]-x[i])*Dt+v[i])
            /(1+(1-w[i])*Dt+Dt**2*(phi1[i]+phi2[i])))
    return(np.array(v_particulas))

def posicion_RR_GPSO(x, v_p, Dt):
    return(v_p*Dt+x)
```

Listing A.15: Algoritmo RR-PSO, construcción del radio espectral.

```
w,phi,Dt = sp.symbols("w,phi,Dt",real=True)

A = (2+(1-w)*Dt)/(1+(1-w)*Dt+phi*Dt**2)
B = -1/(1+(1-w)*Dt+phi*Dt**2)

A_sigma2 = sp.Matrix([[A**2,2*A*B,B**2],[A,B,0],[1,0,0]])
aval_sigma2 = A_sigma2.eigenvals()

EQ = []
for eq in aval_sigma2:
    EQ.append(eq)

for i,eq in enumerate(EQ):
    EQ[i]=eq.subs(Dt,1)

def radio_espectral(W,P):
    lambda_sigma1 = np.abs(EQ[0].subs(w,W).subs(phi,P))
    lambda_sigma2 = np.abs(EQ[1].subs(w,W).subs(phi,P))
    lambda_sigma3 = np.abs(EQ[2].subs(w,W).subs(phi,P))
    return max([lambda_sigma1,lambda_sigma2,lambda_sigma3])
```

Listing A.16: Algoritmo RR-PSO, construcción de la distribución de las partículas.

```
def param_particulas(radio_espectral, N, phimin, phimax, wmin, wmax):
    parametros = [[],[],[]]
    for i in range(N):
        cond = False
        while not cond:
            w_p = np.random.rand()*10
            phi_p = np.random.rand()*10
            cond = (radio_espectral(w_p,phi_p) >= 0.8) and (radio_espectral(w_p,
                phi_p) <= 1.0)
            if cond:
                parametros[0].append(w_p)
                ag=np.random.rand()*phi_p
                parametros[1].append(ag)
                parametros[2].append(phi_p-ag)
    return(np.array(parametros))
```

Listing A.17: Algoritmo RR-PSO, programa principal.

```
def RR_PSO(matriz, sentimientos_num, N, Nit, par_min, par_max, phimin, phimax,
    wmin, wmax, tmin, tmax):
    # Aportar un valor inicial a las partículas
    conjunto = conjunto_inicial(N, par_min, par_max)

    # Establecer inercia y aceleración global y local de cada partícula
    par_par = param_particulas(radio_espectral, N, phimin, phimax, wmin, wmax)

    # Matriz y etiquetas del conjunto de datos para el entrenamiento y la prueba
    X_train,X_test, y_train,y_test = train_test_split(matriz, sentimientos_num,
                                                      test_size=0.2, random_state=1)

    # Evaluar el conjunto inicial para buscar cuál es el máximo global
    print(" Número de iteración: 0")
    acc = eval_conjunto(conjunto, X_train, X_test, y_train, y_test)
    glob = conjunto[np.argmax(acc)]
    acc_max = acc.max()

    # Crear la matriz con los mínimos locales. Ahora, el conjunto inicial
    locales = np.copy(conjunto)
    locales_acc = np.copy(acc)

    # Crear elementos para guardar la información de las iteraciones
    posiciones = [conjunto]
    velocidades = [np.zeros(conjunto.shape)]
    GLOB = [glob]
    ACC = [acc_max]
    print("\n",glob)
    print(acc_max)

    # Iniciar las iteraciones
    for i in range(Nit):
        print("\n Número de iteración:", i+1)

        Dt = tmin + (tmax-tmin)*np.random.rand()
        omega = par_par[0]*np.random.rand(len(par_par[0]))
        phi1 = par_par[1]*np.random.rand(len(par_par[1]))
        phi2 = par_par[2]*np.random.rand(len(par_par[2]))
        velocidades.append(velocidad_RR_GPSO(posiciones[i],velocidades[i],omega,
            Dt,phi1,phi2,glob,locales))
        posiciones.append(posicion_RR_GPSO(posiciones[i],velocidades[i+1],Dt))

        # Revisar si las posiciones se han salido de los rangos
        vmin_away = posiciones[i+1] <= par_min
        vmin_keep = posiciones[i+1] > par_min
        vmax_away = posiciones[i+1] >= par_max
        vmax_keep = posiciones[i+1] < par_max

        posiciones[i+1] = (vmin_away* par_min ) + (vmin_keep* posiciones[i+1] )
        posiciones[i+1] = (vmax_away* par_max ) + (vmax_keep* posiciones[i+1] )
```

```
        # Comprobar si hay un cambio en el máximo global
        acc = eval_conjunto(posiciones[i+1], X_train, X_test, y_train, y_test)
        if acc.max() > acc_max:
            acc_max = acc.max()
            glob = posiciones[i+1][np.argmax(acc)]

        # Comprobar si hay cambio en cada máximo local
        for j in range(len(posiciones[i+1])):
            if acc[j] > locales_acc[j]:
                locales_acc[j] = acc[j]
                locales[j] = posiciones[i+1][j]

        GLOB.append(glob)
        ACC.append(acc_max)
        print("\n",glob)
        print(acc_max)

    return(posiciones,velocidades,GLOB,ACC,glob,acc_max)
```

Listing A.18: Algoritmo RR-PSO, una ejecición del programa.

```
# RR_PSO(matriz, sentimientos_num, N, Nit, par_min, par_max, phimin, phimax,
    wmin, wmax, tmin, tmax)
par_min = np.array([100, 2, 0.01, 0, 0, 0.4, 0.6])
par_max = np.array([200, 10, 0.3, 0.5, 1, 1, 1])
posiciones,velocidades,GLOB,ACC,glob,acc_max = RR_PSO(x85_FinBERT,
    sentimientos_num, 20,20, par_min,par_max, 0,10,0,10, 0.8,1.2)
```