

Proyecto: Reconocimiento de documentos y extracción de información.

Entrenamiento de SVM

14 de Diciembre 2020

Introducción

El presente documento es una guía sobre como realizar el entrenamiento de la Máquina de Vectores de Soporte (SVM) como clasificador de documentos. Se cubrirán temas como:

- Nomenclatura de documentos.
- Preparación de sets de datos.
- Extracción de características.
- Entrenamiento de SVM.

Nomenclatura de documentos

Se normalizó el nombre de los archivos con la intención de facilitar su organización, lectura y clasificación. Para poder identificar fácilmente los archivos, se implementó la siguiente estructura: "Documento + Posición + Tipo de documento + Numeración por tipo". Por ejemplo:

- IFEFRC_001 ->IFE + Frente + Tipo C + Numeración
- INEREGH_001 ->INE + Reverso + Tipo GH + Numeración
- Luz_001 ->Recibo de luz + Numeración

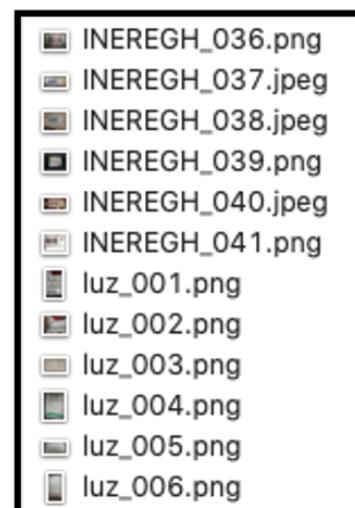
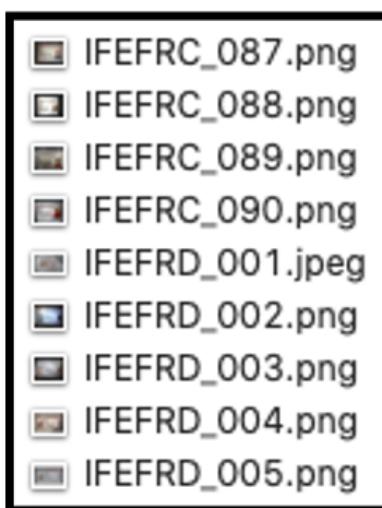


Figura 1: Ejemplos.

De esta manera, uno puede conocer de que tipo de documento se trata y, al cargar los archivos en Python, es posible usar el nombre como etiqueta de clase al separar el nombre con el guión bajo. Los nombres recomendados para los documentos son:

- IFEFRC
- IFEFRD
- INEFREF
- INEFRGH
- IFEREC

- IFERED
 - INEREFF
 - INEREGH
 - Luz
 - Telmex (pudiesen existir de otras compañías)
 - Pasaporte

Preparación de sets de datos

Usualmente para un entrenamiento de clasificación se utiliza una proporción del 70 % de dataset para crear el set de entrenamiento y un 30 % para el set de pruebas. Otro punto importante es que se tomen los datos de manera aleatoria, asegurando que no se está sesgando el entrenamiento. Una manera sencilla de aleatorizar la información y de conocer que ejemplos quedaron en cada conjunto es enlistarlos en un archivo de excel, crear valores aleatorios en la siguiente columna y ordenar los ambas columnas. Para entrenar la SVM de manera correcta, es necesario aleatorizar los archivos por cada clase y después concatenar las distintas clases. Por ejemplo, se necesita tener los samples de IFEFRC en la posición 1 a la 30, después todos los de INEREEF. No importa el orden de las clases pero si que todos sus ejemplos se encuentren de manera consecutiva.

	A	B	C
1		16	12
2		agua_014.png	0.25422066
3		agua_010.png	0.35092302
4		agua_006.png	0.37658727
5		agua_012.png	0.38011978
6		agua_007.png	0.40831058
7		agua_013.png	0.46568132
8		agua_005.png	0.51869486
9		agua_016.png	0.52038388
10		agua_008.png	0.59961131
11		agua_002.png	0.66124351
12		agua_001.png	0.7840655
13	**	agua_011.png	0.7973199
14		agua_015.png	0.83925349
15		agua_003.png	0.84597326
16		agua_004.png	0.87355188
17		agua_009.png	0.90600672
18			

Figura 2: Caption

Una vez que se tiene el listado de los archivos pertenecientes a cada set, yo recomiendo guardarlos como listas en un archivo de Python para poder importarlos rápidamente en los entrenamientos.

```
1 trainingFiles = [  
2   'IFEFRC_086.jpeg',  
3   'IFEFRC_071.png',  
4   'IFEFRC_070.png',  
5   'IFEFRC_031.png',  
6   'IFEFRC_025.png',  
7   'IFEFRC_069.png',  
8   'IFEFRC_044.png',  
9   'IFEFRC_013.png',  
10  'IFEFRC_034.png',  
11  'IFEFRC_021.png',  
12  'IFEFRC_057.png',  
13  'IFEFRC_088.png',  
14  'IFEFRC_030.png',  
15  'IFEFRC_066.png',  
16  'IFEFRC_004.png',  
  
612 testFiles = [  
613   'IFEFRC_047.png',  
614   'IFEFRC_067.jpeg',  
615   'IFEFRC_050.png',  
616   'IFEFRC_054.png',  
617   'IFEFRC_027.png',  
618   'IFEFRC_052.png',  
619   'IFEFRC_077.png',  
620   'IFEFRC_079.png',  
621   'IFEFRC_075.png',  
622   'IFEFRC_045.png',  
623   'IFEFRC_040.png',  
624   'IFEFRC_037.png',  
625   'IFEFRC_061.png',  
626   'IFEFRC_051.png',  
627   'IFEFRC_020.png',
```

Figura 3: Archivos para entrenamiento y prueba.

Extracción de características

El archivo “featureExtractor.py” contiene el algoritmo donde se procesa cada archivo y se extrae su vector de características. Aunque el archivo se encuentra comentado, se explicarán algunos segmentos importantes para entender su funcionamiento.

```
1 # featureExtractor.py
2 from tensorflow.keras.applications.vgg16 import VGG16
3 from tensorflow.keras.applications.vgg16 import preprocess_input
4 from tensorflow.keras.preprocessing import image
5 from sklearn.preprocessing import StandardScaler
6 from sklearn.decomposition import PCA
7 from tqdm import tqdm
8 import numpy as np
9 import pickle
10 import os
11
12
13 # Archivo con listas de ejemplos para entrenamiento y pruebas
14 from TrainingFiles import *
15
16 # Carpeta de imágenes para entrenamiento y pruebas
17 filesDir = "/Volumes/SD128/Proyectos/Xira/XiraDataset/"
18
19 # Modelo pre-entrenado
20 model = VGG16(weights='imagenet', include_top=False)
```

Las líneas 2 y 3 importan las librerías del modelo pre-entrenado en Keras. Se pueden consultar otros modelos en <https://keras.io/api/applications/>. La línea 4 se encarga de cargar y redimensionar cada ejemplo. 5 y 6 son los modelos para normalización de datos y reducción de dimensionalidad. 7 carga una barra de proceso para conocer el avance. En la línea 14 encontramos el archivo que se realizó en la sección pasada, donde se encuentran las listas de archivos de entrenamiento y pruebas. La ubicación en la línea 17 es a la carpeta donde, después de renombrar los archivos con la nomenclatura adecuada, se encuentran todos los archivos juntos. Las líneas 14 y 17 depende del nombre y ubicación que haya creado el usuario para los archivos a entrenar. Finalmente, la línea 20 define el modelo pre-entrenado, que se va a usar para extraer las características. En este caso, la opción **include_top=False** remueve la etapa de clasificación del modelo.

```
22 # Archivos para entrenamiento
23 trainingLabels = []
24 trainingClasses = []
25 embeddings = []
26 trainingEmbeddings = np.array([])
27 trainingDelimitations = []
28
29 fileNumber = 0
30 firstClassFile = 0
31 lastClassFile = 0
32 lastFile = len(trainingFiles)-1
33
34 # Barra de progreso
35 pbar = tqdm(total=len(trainingFiles))
```

Para procesar las muestras de entrenamiento se generan las siguientes variables:

- **trainingLabels**: guarda, por cada archivo, la etiqueta de la clase a la que pertenece. Si existen 300 ejemplos de entrenamiento, esta variable tendrá 300 valores.
- **trainingClasses**: guarda 1 sola vez el nombre de cada clase. Funciona para extraer el índice de cada clase y guardarlo en trainingLabels. Solo depende del número de clases que queramos entrenar.
- **embeddings**: lista donde temporalmente se guardan los vectores de características que se van obteniendo. Es más fácil de procesar y agregar nuevos valores.
- **trainingEmbeddings** (arreglo Numpy): para el entrenamiento es necesario tener la información en formato de arreglo de Numpy. Al finalizar la extracción se convierte la lista embeddings a este formato.
- **trainingDelimitations**: Esta variable guarda las posiciones donde se encuentras los archivos de cada clase. La SVM genera un modelo por cada clase en el formato **One vs All**, por lo cual es importante ordenar la información e ir intercambiando ejemplos.

Todas estas variables tienen su homólogo para pruebas (testLabels, testClasses, ...) y se guardarán al finalizar para poder realizar el entrenamiento correctamente. Además, en la líneas 29-32 se tienen variables de apoyo como un contador (29), temporales para guardar los límites de cada clase (30-31) y el indicador de donde termina el conteo (32). La línea 35 genera la barra de progreso con el número total de archivos a procesar.

```

37 for file in trainingFiles:
38     # Actualiza barra de progreso
39     pbar.update(1)
40     # Carga una imagen
41     img_path = os.path.join(filesDir, file)
42
43     # Si es el primer ejemplo...
44     if fileNumber == 0:
45         # Agrega la primera clase a la lista
46         trainingClasses.append(file.split("_")[0])
47         fileNumber += 1
48         # Agrega la etiqueta numérica de la clase a la que pertenece
49         trainingLabels.append(trainingClasses.index(file.split("_")[0]))

```

Por cada uno de los ejemplos en la lista de entrenamiento, primero se actualiza la barra de progreso y se carga la imagen correspondiente (41). Si es el primer ejemplo de la lista, se obtiene su clase del nombre del archivo (IFEFRC_001.png = IFEFRC) y se anexa al listado de clases. Se incrementa el contador de archivos y se agrega el valor de la clase al listado de etiquetas (49), en este caso por ser el primero será **0**. La consulta realmente es buscar la clase del nombre del archivo en el listado de clases y obtener su índice.

```

51     # Carga y redimensiona la imagen para procesarla con el modelo pre-entrenado
52     img = image.load_img(img_path, target_size=(224, 224))
53     x = image.img_to_array(img)
54     x = np.expand_dims(x, axis=0)
55     x = preprocess_input(x)
56     # Obtiene las características en una matriz de 7x7x512 (VGG16)
57     features = model.predict(x)
58     # Convierte la lista a un arreglo de Numpy
59     featuresNumpy = np.array(features)
60     # Concatena los valores de la matriz de 7x7x512 y genera un vector de 25088 características
61     singleVector = featuresNumpy.flatten()
62     # Agrega el vector al listado de ejemplos de entrenamiento
63     embeddings.append(singleVector.tolist())

```

Después de agregar la etiqueta de la clase, comienza la etapa de extracción de características. Primeramente, se vuelve a cargar la imagen pero redimensionada a 224 por 224 pixeles y se realiza un preprocesamiento (52-55). En la línea 57 se obtienen las características de la imagen, en este caso una matriz de 7x7x512 por tratarse del modelo VGG16. Otros modelos tienen otro tipo de estructuras de salida. Tras convertir la salida en un arreglo Numpy, el algoritmo concatena todos los valores dentro de la matriz mencionada y genera un vector de 25,088 características. Este vector es anexado a la lista de embeddings.

```

65     # Para los siguientes archivos
66     else:
67         # Revisa si la clase ya existe en el listado de clases. Si no existe, la anexa
68         fileClass = file.split("_")[0]
69         if fileClass not in trainingClasses:
70             trainingClasses.append(fileClass)
71             # Como es una clase distinta, agrega el valor de donde termina la clase previa
72             lastClassFile = fileNumber-1
73             trainingDelimitations.append([firstClassFile, lastClassFile])
74             # Guarda la posición del primer ejemplo de la nueva clase
75             firstClassFile = fileNumber
76             # Agrega la etiqueta numérica de la clase a la que pertenece
77             trainingLabels.append(trainingClasses.index(file.split("_")[0]))

```

Si el ejemplo no es el primero, primeramente se analiza la clase a la que pertenece el archivo. Si ya existe en el listado, se continua el proceso de agregar la etiqueta correspondiente al listado (como en la línea 49) y lo sucesivo hasta la extracción de características. Si la clase no existe en el listado, es agregada (70) y se utilizan los indicadores de límites (firstClassFile y lastClassFile) para indicar los índices de los ejemplos que abarca la clase previa (73). Finalmente, se asigna la posición actual como el primer ejemplo de la siguiente clase. Después se continua con el procesamiento del ejemplo desde agregar la etiqueta hasta la extracción de características.

```

92     # Si es el último ejemplo...
93     if fileNumber == lastFile:
94         # Agrega los delimitadores de la última clase
95         trainingDelimitations.append([firstClassFile, fileNumber])
96         # O continua la cuenta
97     else:
98         fileNumber += 1
99
100    # Convierte la lista de características en un arreglo de Numpy
101    trainingEmbeddings = np.asarray(embeddings)
102
103    # Guarda las características en un archivo CSV
104    np.savetxt("TrainingFeatures_VGG16_25088.csv", trainingEmbeddings, delimiter=",")

```

Una vez que han sido procesados todos los ejemplos y nos encontramos en el último, se anexan los límites de última clase (95), se convierte la lista de embeddings a un arreglo Numpy (101) y finalmente se guardan todos los vectores de características en un archivo CSV.

```
191 # Guarda las características en un archivo CSV
192 np.savetxt("TestFeatures_VGG16_25088.csv", testEmbeddings, delimiter=",")
193
194
195 # Guarda todas las variables necesarias para un entrenamiento con las 25,088 características
196 np.savez("VGG16_25088features.npz", trainingFiles=trainingFiles, testFiles=testFiles,
197     trainingLabels=trainingLabels, testLabels=testLabels, trainingClasses=trainingClasses,
198     testClasses=testClasses, trainingDelimitations=trainingDelimitations, testDelimitations=testDelimitations,
199     trainingEmbeddings=trainingEmbeddings, testEmbeddings=testEmbeddings)
```

Para los ejemplos de prueba ocurre el mismo procedimiento. Esto se encuentra de la línea 108 a la 192, concluyendo con guardar todas los vectores de características de los ejemplos de pruebas. Una vez concluida la extracción de características con el modelo VGG16, se guardan todas las variables en un archivo NPZ por si se desean procesar de alguna manera o con algún clasificador en particular (196).

```
202 # Estandarización de valores
203 scaler = StandardScaler()
204
205 # Se ajusta con los valores de entrenamiento
206 scaler.fit(trainingEmbeddings)
207
208 # Se aplica el ajuste a ambos conjuntos de características
209 trainEmbeddings2 = scaler.transform(trainingEmbeddings)
210 testEmbeddings2 = scaler.transform(testEmbeddings)
211
212 # Igualmente, el algoritmo de PCA se ajusta con los nuevos valores de entrenamiento
213 pca = PCA(.95)
214 pca.fit(trainEmbeddings2)
215
216 # Se aplica el PCA a ambos conjuntos de características
217 trainPCA = pca.transform(trainEmbeddings2)
218 testPCA = pca.transform(testEmbeddings2)
219
220 # Guarda todas las variables necesarias para un entrenamiento con las 475 características obtenidas del PCA
221 np.savez("VGG16_PCA_475features.npz", trainingFiles=trainingFiles, testFiles=testFiles,
222     trainingLabels=trainingLabels, testLabels=testLabels, trainingClasses=trainingClasses,
223     testClasses=testClasses, trainingDelimitations=trainingDelimitations, testDelimitations=testDelimitations,
224     trainingEmbeddings=trainPCA, testEmbeddings=testPCA)
225
226 # Guarda SCALER y PCA (necesarios para procesar cualquier nueva muestra)
227 pickle.dump(scaler, open('scaler', 'wb'))
228 pickle.dump(pca, open('pca', 'wb'))
```

La última etapa del archivo consiste en normalizar los valores obtenidos y encontrar aquellas características que son más representativas. En la línea 203 se define el Scaler que permitirá ajustar los valores de los vectores entre 0 y 1. Por cada dimensión de los vectores de entrenamiento, se obtiene la media y la desviación estándar, información que se guarda en el Scaler (206). Cada valor es normalizado al restarle la media y dividirlo entre la desviación, estándar, tanto en el conjunto de entrenamiento (209), como en el conjunto de pruebas (210).

En el caso del Análisis de Componentes Principales (PCA), igualmente se defina y se ajusta con los valores del conjunto de entrenamiento (213-214), se guarda la información sobre que columnas son las más representativas y se aplica a ambos conjuntos (217-218). En este caso, el PCA generó vectores de 475 características.

Finalmente, todas las variables necesarias para el entrenamiento son guardadas en un archivo NPZ (221). También se guarda el Scaler (227) y el PCA (228) ya que se necesitan para procesar cualquier nueva muestra.

Entrenamiento de SVM

El archivo “trainingSVM.py” contiene el algoritmo para entrenar la SVM con las características obtenidas en la sección pasada. El archivo se encuentra ajustado para las 9 clases con las que se realizaron pruebas. Aunque el archivo se encuentra comentado, se explicarán algunos segmentos importantes para entender su funcionamiento.

```
1 # trainingSVM
2 from sklearn.svm import SVC
3 import numpy as np
4 import pickle
5
6 # Archivo con las variables para el entrenamiento
7 datasetFile = "VGG16_PCA_475features.npz"
8
9 # Cargar archivo y asignar variables
10 loadeddata = np.load(datasetFile)
11 trainingFiles = loadeddata["trainingFiles"]
12 testFiles = loadeddata["testFiles"]
13 trainingLabels = loadeddata["trainingLabels"]
14 testLabels = loadeddata["testLabels"]
15 trainingClasses = loadeddata["trainingClasses"]
16 testClasses = loadeddata["testClasses"]
17 trainingDelimitations = loadeddata["trainingDelimitations"]
18 testDelimitations = loadeddata["testDelimitations"]
19 trainingEmbeddings = loadeddata["trainingEmbeddings"]
20 testEmbeddings = loadeddata["testEmbeddings"]
```

En las líneas 2-4 se cargan las librerías necesarias para el entrenamiento. En la línea 7 se da la ruta hacia el archivo donde se guardaron las variables para el entrenamiento. Si el archivo se encuentra en otra ubicación será necesario poner el path completo. Posteriormente, se carga el archivo y se asignan los valores a las variables correspondientes (10-20).

```
21 # Se entrena tantos modelos como clases se tengan, la SVM utiliza un enfoque One vs All
22 for targetClass in range(len(trainingClasses)):
23
24     yTrain = []
25     # Se obtienen los límites de la clase a entrenar (que archivos comprenden esta clase)
26     lowBoundry = trainingDelimitations[targetClass, 0]
27     highBoundry = trainingDelimitations[targetClass, 1]
28
29     # Si el ejemplo pertenece a la clase a entrenar, su etiqueta será 1, caso contrario 0
30     for i in range(len(trainingEmbeddings)):
31         if i >= lowBoundry and i <= highBoundry:
32             yTrain.append(1)
33         else:
34             yTrain.append(0)
```

Para realizar una clasificación multiclas con una SVM, es necesario entender que el algoritmo genera modelos One vs All, que significa que una solo puede clasificar una clase por cada modelo. Para entrenar un modelo, asigna a los ejemplos de la clase a entrenar una etiqueta de 1, y para el resto de las clases las agrupa con una etiqueta 0. Al evaluar una nueva muestra con un modelo determinado, la respuesta que obtenemos es la probabilidad de que pertenezca a dicha clase. En nuestro caso en particular, se generarán 9 modelos correspondientes a los 9 tipos de documentos que buscamos clasificar.

Para entrenar cada uno de los modelos, el algoritmo busca los límites de ejemplos (26-27) e irá creando el nuevo vector de etiquetas de clasificación. A todos los archivos que se encuentren dentro del rango les asignará la etiqueta 1 (32), caso contrario un 0 (34).

```

36 # Dependencia del orden de los archivos a entrenar
37 if targetClass == 0:
38     # Inicializa el modelo con un tipo de kernel y la opción de obtener la probabilidad en decimales.
39     IFEFRCmodel = SVC(kernel='linear', probability = True)
40     # Se entrena el modelo con las muestras y etiquetas de una determinada clase.
41     IFEFRCmodel.fit(trainingEmbeddings, yTrain)
42 elif targetClass == 1:
43     IFEFRDmodel = SVC(kernel='linear', probability = True)
44     IFEFRDmodel.fit(trainingEmbeddings, yTrain)
45 elif targetClass == 2:
46     INEFREFmodel = SVC(kernel='linear', probability = True)
47     INEFREFmodel.fit(trainingEmbeddings, yTrain)
48 elif targetClass == 3:
49     INEFRGHmodel = SVC(kernel='linear', probability = True)
50     INEFRGHmodel.fit(trainingEmbeddings, yTrain)
51 elif targetClass == 4:
52     IFERECmodel = SVC(kernel='linear', probability = True)
53     IFERECmodel.fit(trainingEmbeddings, yTrain)
54 elif targetClass == 5:
55     IFEREDmodel = SVC(kernel='linear', probability = True)
56     IFEREDmodel.fit(trainingEmbeddings, yTrain)
57 elif targetClass == 6:
58     INEREEFmodel = SVC(kernel='linear', probability = True)
59     INEREEFmodel.fit(trainingEmbeddings, yTrain)
60 elif targetClass == 7:
61     INEREGHmodel = SVC(kernel='linear', probability = True)
62     INEREGHmodel.fit(trainingEmbeddings, yTrain)
63 elif targetClass == 8:
64     LUZmodel = SVC(kernel='linear', probability = True)
65     LUZmodel.fit(trainingEmbeddings, yTrain)

```

Es importante conocer el orden en que se enlistaron los ejemplos de entrenamiento para poder diferenciar que tipo de documento clasificará cada modelo. En este caso, al conocer el orden de las 9 clases, puedo inicializar modelos diferentes con nombres particulares. Igualmente se pueden crear modelos enumerados, pero es importante saber que información clasifica cada uno. Una vez que identifica que clase es la que va a entrenar, se inicializa un modelo con un kernel determinado y se activa la opción de conocer la probabilidad de pertenencia a dicha clase (39). Después se entrena el modelo con los vectores de entrenamiento y el vector de clases de salida (41).

```

67 # Inicialización de la matriz de confusión (depende de las clases que se entrena)
68 matResults = np.zeros([len(trainingClasses), len(trainingClasses)], dtype=int)
69
70 # Evaluación de las muestras de prueba
71 for i in range(len(testEmbeddings)):
72     testSample = testEmbeddings[i:i+1]
73     results = []
74     # Evalúa la muestra con los modelos creados y se guarda la probabilidad en la variable results
75     results.append(float(IFEFRCmodel.decision_function(testSample)))
76     results.append(float(IFERFDmodel.decision_function(testSample)))
77     results.append(float(INEFREFmodel.decision_function(testSample)))
78     results.append(float(INEFRGHmodel.decision_function(testSample)))
79     results.append(float(IFERECmodel.decision_function(testSample)))
80     results.append(float(IFEREDmodel.decision_function(testSample)))
81     results.append(float(INEREEFmodel.decision_function(testSample)))
82     results.append(float(INEREGHmodel.decision_function(testSample)))
83     results.append(float(LUZmodel.decision_function(testSample)))

```

Después de crear los modelos necesarios, se evalúan los ejemplos para prueba. Primero se inicializa la matriz de confusión, que es una matriz cuadrada de ceros que depende del número de clases (68). Cada ejemplo del conjunto de pruebas se evalúa con los modelos creados y se guarda la probabilidad de pertenencia con cada uno (75-83).

```

85 # Busca el mayor valor en los resultados
86 bestModel = max(results)
87 # Busca a qué modelo pertenece el mejor resultado
88 position = results.index(bestModel)
89 # Suma 1 punto acorde a la posición de la clase real contra la predicha por los modelos
90 matResults[testLabels[i],position] += 1
91
92 # Imprime la clase predicha para la muestra y la clase real a la que pertenecía
93 print("Pred: " + str(position) + " (" + testClasses[position] + ")", Real: " + str(testLabels[i]) + " (" +
94     + testClasses[testLabels[i]] + ")")
95
96 # Imprime la matriz de confusión
97 print(matResults)

```

Posteriormente, se analiza el vector de resultados para encontrar el mayor valor (86). Se extrae la posición del mejor valor (88) y se suma un 1 en la matriz de confusión a la posición dada por la clase real y el valor predicho por los modelos (90). Como visualización de los resultados, por cada ejemplo se imprime también el valor predicho, o el modelo con el que tuvo mayor pertenencia la muestra, y la clase a la que realmente pertenece (93). Finalmente se presenta la matriz de confusión con todas las evaluaciones (97).

```
100 pickle.dump(IFEFRCmodel, open('IFEFRCmodel', 'wb'))
101 pickle.dump(IFEFRDmodel, open('IFEFRDmodel', 'wb'))
102 pickle.dump(INEFREFmodel, open('INEFREFmodel', 'wb'))
103 pickle.dump(INEFRGHmodel, open('INEFRGHmodel', 'wb'))
104 pickle.dump(IFERECmodel, open('IFERECmodel', 'wb'))
105 pickle.dump(IFEREDmodel, open('IFEREDmodel', 'wb'))
106 pickle.dump(INEREEFmodel, open('INEREEFmodel', 'wb'))
107 pickle.dump(INEREGHmodel, open('INEREGHmodel', 'wb'))
108 pickle.dump(LUZmodel, open('LUZmodel', 'wb'))
```

Por último, se guardan los modelos creados en archivos pickle.