activity 3.1 - abnormal_fish_oversampling

November 26, 2025

Professor: Enrique Garcia Ceja **email:** enrique.gc@tec.mx

1 Demo: Compare abnormal fish behaviors (oversampling and weighted model).

Files: fishFeatures.csv

Refer to the *fish-behaviors.pptx* presentation for details about the dataset.

[107]:
```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from sklearn.metrics import accuracy_score, recall_score
```

[108]:
```
# Path to the dataset.
filepath = "../data/fishFeatures.csv"

# Read the data
dataset = pd.read_csv(filepath)
```

[109]:
```
dataset.head()
```

[109]:

	id	label	f.meanSpeed	f.sdSpeed	f.minSpeed	f.maxSpeed	f.meanAcc	f.sdAcc	f.minAcc	f.maxAcc
0	id1	normal	2.623236	2.228456	0.500000	8.225342	-0.053660	1.839475	-5.532760	3.500000
1	id2	normal	5.984859	3.820270	1.414214	15.101738	-0.038705	2.660073	-7.273932	7.058594
2	id3	normal	16.608716	14.502042	0.707107	46.424670	-1.000196	12.890386	-24.320298	30.714624
3	id5	normal	4.808608	4.137387	0.500000	17.204651	-0.281815	5.228209	-12.204651	15.623512
4	id6	normal	17.785747	9.926729	3.354102	44.240818	-0.537534	11.272472	-22.178067	21.768613

[110]:
```
# remove id column
dataset = dataset.drop('id', axis=1)
dataset.head()
```

[110]:

	label	f.meanSpeed	f.sdSpeed	f.minSpeed	f.maxSpeed	f.meanAcc	f.sdAcc	f.minAcc	f.maxAcc
0	normal	2.623236	2.228456	0.500000	8.225342	-0.053660	1.839475	-5.532760	3.500000
1	normal	5.984859	3.820270	1.414214	15.101738	-0.038705	2.660073	-7.273932	7.058594
2	normal	16.608716	14.502042	0.707107	46.424670	-1.000196	12.890386	-24.320298	30.714624
3	normal	4.808608	4.137387	0.500000	17.204651	-0.281815	5.228209	-12.204651	15.623512
4	normal	17.785747	9.926729	3.354102	44.240818	-0.537534	11.272472	-22.178067	21.768613

[111]:
```
# Count labels
dataset['label'].value_counts()
```

[111]:
```
label
normal      1093
abnormal      54
Name: count, dtype: int64
```

[112]:
```
# Shuffle the dataset
from sklearn.utils import shuffle

seed = 1234 #set seed for reproducibility

np.random.seed(seed)

dataset = shuffle(dataset)
```

[113]:
```
#Select features and class
features = dataset.drop('label', axis=1)

labels = dataset[['label']]

features = features.values.astype(float)

labels = labels.values
```

[114]:
```
features.shape
```

[114]:
```
(1147, 8)
```

[115]:
```
# Convert labels to integers.
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
labels_int = le.fit_transform(labels.ravel())
```

[116]:
```
print(labels[0])
print(labels_int[0])
```

```
['normal']
1
```

[117]:
```
# One hot encode labels using the to_categorical function of keras.
labels = tf.keras.utils.to_categorical(labels_int, num_classes = 2)
```

[118]:
```
labels[0:10,:]
```

[118]:
```
array([[0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

[119]:
```
# Split into train and test sets.
from sklearn.model_selection import train_test_split

train_features, test_features, train_labels, test_labels =␣
 ↪train_test_split(features, labels,

                                                          ␣
 ↪test_size = 0.50, random_state = 1234)
```

[120]:
```
# count unique values in train_labels
unique_labels, counts = np.unique(train_labels, axis=0, return_counts=True)
normal = counts[0]
abnormal = counts[1]

print("Normal: ", normal)
print("Abnormal: ", abnormal)
```

```
Normal:  547
Abnormal:  26
```

[121]:
```
# Normalize features between 0 and 1
```

```
# Normalization parameters are learned just from the training data to avoid␣
 ↪information injection.
from sklearn import preprocessing

normalizer = preprocessing.StandardScaler().fit(train_features)
train_normalized = normalizer.transform(train_features)
test_normalized = normalizer.transform(test_features)
```

1.0.1 Define the model

[122]:
```
# Define the model.
model = keras.Sequential([
    keras.layers.Dense(units = 16, input_shape=(8,), activation=tf.nn.relu),
    keras.layers.Dense(units = 8, activation=tf.nn.relu),
    keras.layers.Dense(units = 2, activation=tf.nn.softmax)
])
```

```
C:\Users\User\AppData\Roaming\Python\Python313\site-
packages\keras\src\layers\core\dense.py:93: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[123]:
```
print(model.summary())
```

```
Model: "sequential_5"
```

Layer (type)	Output Shape	Param #
dense_15 (Dense)	(None, 16)	144
dense_16 (Dense)	(None, 8)	136
dense_17 (Dense)	(None, 2)	18

```
Total params: 298 (1.16 KB)

Trainable params: 298 (1.16 KB)

Non-trainable params: 0 (0.00 B)

None
```

[124]:
```
# Calculate class weights.
# Scaling by total/2 helps keep the loss to a similar magnitude.
# The sum of the weights of all examples stays the same.
total = abnormal + normal
weight_for_0 = (1 / abnormal) * (total / 2.0)
weight_for_1 = (1 / normal) * (total / 2.0)

class_weight = {0: weight_for_0, 1: weight_for_1}

print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))
```

```
Weight for class 0: 11.02
Weight for class 1: 0.52
```

[125]:
```
# Define the optimizer. Stochastic Gradient Descent in this case.
optimizer = tf.keras.optimizers.SGD(learning_rate = 0.01)

model.compile(optimizer = optimizer,
              loss = "categorical_crossentropy",
              metrics = ['accuracy'])

# Train the model.
history = model.fit(train_normalized, train_labels,
                    epochs = 100,
                    validation_split = 0.0,
                    batch_size = 256,
                    class_weight=class_weight,
                    verbose = 1)
```

```
Epoch 1/100
3/3             1s 28ms/step -
accuracy: 0.0940 - loss: 0.6911
Epoch 2/100
3/3             0s 27ms/step -
accuracy: 0.0741 - loss: 0.5964
Epoch 3/100
3/3             0s 23ms/step -
accuracy: 0.0881 - loss: 0.5851
Epoch 4/100
3/3             0s 24ms/step -
accuracy: 0.1184 - loss: 0.5573
Epoch 5/100
3/3             0s 27ms/step -
accuracy: 0.1536 - loss: 0.5404
Epoch 6/100
3/3             0s 28ms/step -
accuracy: 0.1827 - loss: 0.5234
```

```
Epoch 7/100
3/3             0s 27ms/step -
accuracy: 0.2187 - loss: 0.5143
Epoch 8/100
3/3             0s 52ms/step -
accuracy: 0.2808 - loss: 0.5202
Epoch 9/100
3/3             0s 52ms/step -
accuracy: 0.3926 - loss: 0.5032
Epoch 10/100
3/3             0s 56ms/step -
accuracy: 0.4526 - loss: 0.5120
Epoch 11/100
3/3             0s 53ms/step -
accuracy: 0.4948 - loss: 0.4864
Epoch 12/100
3/3             0s 45ms/step -
accuracy: 0.5794 - loss: 0.4924
Epoch 13/100
3/3             0s 48ms/step -
accuracy: 0.6163 - loss: 0.4770
Epoch 14/100
3/3             0s 61ms/step -
accuracy: 0.6488 - loss: 0.4747
Epoch 15/100
3/3             0s 52ms/step -
accuracy: 0.6708 - loss: 0.4692
Epoch 16/100
3/3             0s 58ms/step -
accuracy: 0.7054 - loss: 0.4568
Epoch 17/100
3/3             0s 38ms/step -
accuracy: 0.7029 - loss: 0.4528
Epoch 18/100
3/3             0s 28ms/step -
accuracy: 0.7101 - loss: 0.4567
Epoch 19/100
3/3             0s 46ms/step -
accuracy: 0.7279 - loss: 0.4494
Epoch 20/100
3/3             0s 36ms/step -
accuracy: 0.7421 - loss: 0.4465
Epoch 21/100
3/3             0s 31ms/step -
accuracy: 0.7485 - loss: 0.4502
Epoch 22/100
3/3             0s 39ms/step -
accuracy: 0.7568 - loss: 0.4236
```

```
Epoch 23/100
3/3             0s 46ms/step -
accuracy: 0.7508 - loss: 0.4298
Epoch 24/100
3/3             0s 45ms/step -
accuracy: 0.7646 - loss: 0.4338
Epoch 25/100
3/3             0s 92ms/step -
accuracy: 0.7662 - loss: 0.4155
Epoch 26/100
3/3             0s 24ms/step -
accuracy: 0.7842 - loss: 0.4179
Epoch 27/100
3/3             0s 48ms/step -
accuracy: 0.7739 - loss: 0.4241
Epoch 28/100
3/3             0s 37ms/step -
accuracy: 0.7763 - loss: 0.4168
Epoch 29/100
3/3             0s 53ms/step -
accuracy: 0.7939 - loss: 0.4029
Epoch 30/100
3/3             0s 47ms/step -
accuracy: 0.7905 - loss: 0.4013
Epoch 31/100
3/3             0s 46ms/step -
accuracy: 0.7918 - loss: 0.4039
Epoch 32/100
3/3             0s 53ms/step -
accuracy: 0.7903 - loss: 0.4094
Epoch 33/100
3/3             0s 51ms/step -
accuracy: 0.7869 - loss: 0.4040
Epoch 34/100
3/3             0s 51ms/step -
accuracy: 0.8048 - loss: 0.3918
Epoch 35/100
3/3             0s 51ms/step -
accuracy: 0.8108 - loss: 0.3913
Epoch 36/100
3/3             0s 53ms/step -
accuracy: 0.8188 - loss: 0.3842
Epoch 37/100
3/3             0s 54ms/step -
accuracy: 0.8223 - loss: 0.3902
Epoch 38/100
3/3             0s 58ms/step -
accuracy: 0.8341 - loss: 0.3680
```

```
Epoch 39/100
3/3             0s 54ms/step -
accuracy: 0.8259 - loss: 0.3888
Epoch 40/100
3/3             0s 51ms/step -
accuracy: 0.8410 - loss: 0.3734
Epoch 41/100
3/3             0s 28ms/step -
accuracy: 0.8520 - loss: 0.3663
Epoch 42/100
3/3             0s 38ms/step -
accuracy: 0.8480 - loss: 0.3841
Epoch 43/100
3/3             0s 34ms/step -
accuracy: 0.8390 - loss: 0.3643
Epoch 44/100
3/3             0s 31ms/step -
accuracy: 0.8477 - loss: 0.3578
Epoch 45/100
3/3             0s 51ms/step -
accuracy: 0.8515 - loss: 0.3643
Epoch 46/100
3/3             0s 62ms/step -
accuracy: 0.8632 - loss: 0.3506
Epoch 47/100
3/3             0s 37ms/step -
accuracy: 0.8616 - loss: 0.3662
Epoch 48/100
3/3             0s 34ms/step -
accuracy: 0.8691 - loss: 0.3555
Epoch 49/100
3/3             0s 31ms/step -
accuracy: 0.8681 - loss: 0.3334
Epoch 50/100
3/3             0s 31ms/step -
accuracy: 0.8654 - loss: 0.3426
Epoch 51/100
3/3             0s 59ms/step -
accuracy: 0.8652 - loss: 0.3489
Epoch 52/100
3/3             0s 48ms/step -
accuracy: 0.8698 - loss: 0.3448
Epoch 53/100
3/3             0s 33ms/step -
accuracy: 0.8712 - loss: 0.3421
Epoch 54/100
3/3             0s 31ms/step -
accuracy: 0.8658 - loss: 0.3488
```

```
Epoch 55/100
3/3             0s 25ms/step -
accuracy: 0.8696 - loss: 0.3376
Epoch 56/100
3/3             0s 25ms/step -
accuracy: 0.8642 - loss: 0.3347
Epoch 57/100
3/3             0s 27ms/step -
accuracy: 0.8767 - loss: 0.3273
Epoch 58/100
3/3             0s 32ms/step -
accuracy: 0.8694 - loss: 0.3285
Epoch 59/100
3/3             0s 48ms/step -
accuracy: 0.8699 - loss: 0.3221
Epoch 60/100
3/3             0s 50ms/step -
accuracy: 0.8699 - loss: 0.3209
Epoch 61/100
3/3             0s 50ms/step -
accuracy: 0.8717 - loss: 0.3237
Epoch 62/100
3/3             0s 48ms/step -
accuracy: 0.8654 - loss: 0.3321
Epoch 63/100
3/3             0s 51ms/step -
accuracy: 0.8716 - loss: 0.3241
Epoch 64/100
3/3             0s 48ms/step -
accuracy: 0.8726 - loss: 0.3214
Epoch 65/100
3/3             0s 53ms/step -
accuracy: 0.8795 - loss: 0.3269
Epoch 66/100
3/3             0s 71ms/step -
accuracy: 0.8899 - loss: 0.3129
Epoch 67/100
3/3             0s 39ms/step -
accuracy: 0.8854 - loss: 0.3152
Epoch 68/100
3/3             0s 40ms/step -
accuracy: 0.8790 - loss: 0.3159
Epoch 69/100
3/3             0s 62ms/step -
accuracy: 0.8849 - loss: 0.3093
Epoch 70/100
3/3             0s 57ms/step -
accuracy: 0.8809 - loss: 0.3152
```

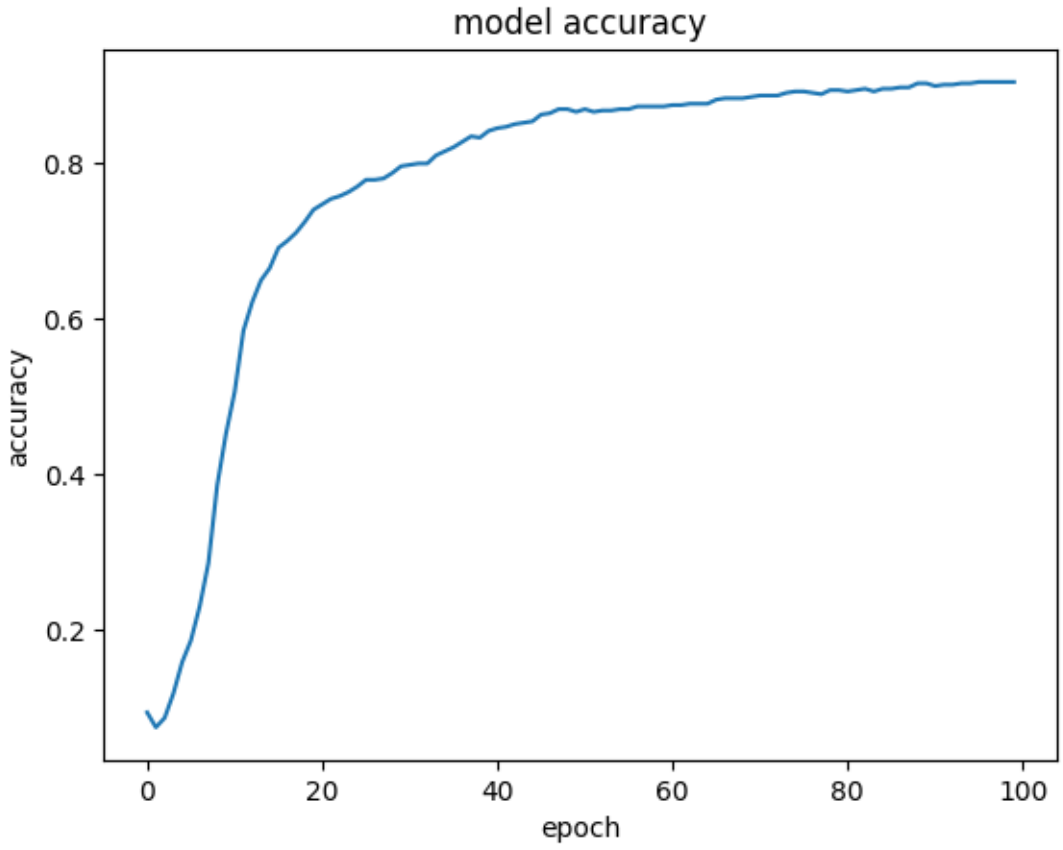
```
Epoch 71/100
3/3             0s 25ms/step -
accuracy: 0.8920 - loss: 0.3073
Epoch 72/100
3/3             0s 37ms/step -
accuracy: 0.8881 - loss: 0.3056
Epoch 73/100
3/3             0s 52ms/step -
accuracy: 0.8891 - loss: 0.3007
Epoch 74/100
3/3             0s 45ms/step -
accuracy: 0.8913 - loss: 0.2977
Epoch 75/100
3/3             0s 36ms/step -
accuracy: 0.8912 - loss: 0.3184
Epoch 76/100
3/3             0s 27ms/step -
accuracy: 0.8902 - loss: 0.2893
Epoch 77/100
3/3             0s 26ms/step -
accuracy: 0.8811 - loss: 0.3023
Epoch 78/100
3/3             0s 31ms/step -
accuracy: 0.8943 - loss: 0.3036
Epoch 79/100
3/3             0s 36ms/step -
accuracy: 0.8989 - loss: 0.2914
Epoch 80/100
3/3             0s 33ms/step -
accuracy: 0.8955 - loss: 0.2923
Epoch 81/100
3/3             0s 31ms/step -
accuracy: 0.8937 - loss: 0.3033
Epoch 82/100
3/3             0s 25ms/step -
accuracy: 0.8989 - loss: 0.2948
Epoch 83/100
3/3             0s 31ms/step -
accuracy: 0.9027 - loss: 0.2749
Epoch 84/100
3/3             0s 30ms/step -
accuracy: 0.8941 - loss: 0.2928
Epoch 85/100
3/3             0s 35ms/step -
accuracy: 0.8949 - loss: 0.3000
Epoch 86/100
3/3             0s 34ms/step -
accuracy: 0.9017 - loss: 0.2811
```

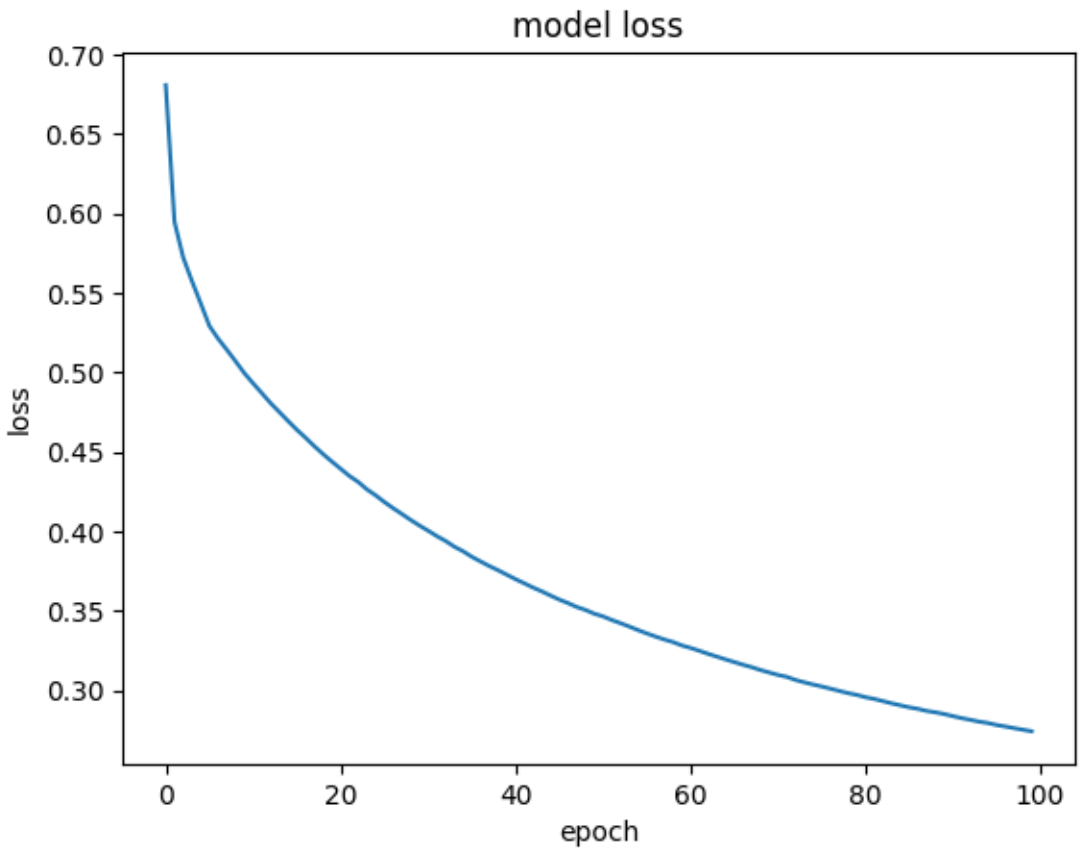
```
Epoch 87/100
3/3             0s 31ms/step -
accuracy: 0.8992 - loss: 0.2904
Epoch 88/100
3/3             0s 35ms/step -
accuracy: 0.9026 - loss: 0.2722
Epoch 89/100
3/3             0s 44ms/step -
accuracy: 0.9028 - loss: 0.2949
Epoch 90/100
3/3             0s 44ms/step -
accuracy: 0.8984 - loss: 0.2904
Epoch 91/100
3/3             0s 34ms/step -
accuracy: 0.8952 - loss: 0.2960
Epoch 92/100
3/3             0s 33ms/step -
accuracy: 0.9049 - loss: 0.2737
Epoch 93/100
3/3             0s 37ms/step -
accuracy: 0.9024 - loss: 0.2746
Epoch 94/100
3/3             0s 37ms/step -
accuracy: 0.9043 - loss: 0.2938
Epoch 95/100
3/3             0s 32ms/step -
accuracy: 0.8979 - loss: 0.2865
Epoch 96/100
3/3             0s 26ms/step -
accuracy: 0.9051 - loss: 0.2828
Epoch 97/100
3/3             0s 34ms/step -
accuracy: 0.8998 - loss: 0.2896
Epoch 98/100
3/3             0s 32ms/step -
accuracy: 0.9081 - loss: 0.2753
Epoch 99/100
3/3             0s 30ms/step -
accuracy: 0.9051 - loss: 0.2900
Epoch 100/100
3/3             0s 39ms/step -
accuracy: 0.9046 - loss: 0.2846
```

[126]:
```
# Plot accuracy and loss curves

import matplotlib.pyplot as plt
%matplotlib inline
```

```
# summarize history for accuracy
plt.plot(history.history['accuracy'])
#plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
#plt.legend(['train', 'validation'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
#plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
#plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```





[127]:
```
# Evaluate the model on the test set and print the loss and accuracy.
model.evaluate(test_normalized, test_labels) # [loss, accuracy]
```

```
18/18             1s 7ms/step -
accuracy: 0.9263 - loss: 0.3312
```

[127]:
```
[0.3695667088031769, 0.9163762927055359]
```

[128]:
```
# Make predictions on the test set.
predictions = model.predict(test_normalized)
```

```
18/18             0s 16ms/step
```

[129]:
```
# Print the first 5 predictions.
predictions[0:5]
```

[129]:
```
array([[0.19104162, 0.8089584 ],
       [0.44952026, 0.5504797 ],
       [0.8938469 , 0.10615302],
       [0.19465633, 0.8053437 ],
       [0.16325381, 0.83674616]], dtype=float32)
```

The predictions are the probabilities for each of the classes. Thus, we need to get the class with the highest probability.

[130]:
```
# Get the column index with max probability from predictions.
predictions_int = np.argmax(predictions, axis=1)

# Ground truth
true_values_int = np.argmax(test_labels, axis=1)
```

[131]:
```
# Convert back to strings
predictions_str = le.inverse_transform(predictions_int)

true_values_str = le.inverse_transform(true_values_int)
```

[132]:
```
pd.crosstab(true_values_str, predictions_str, rownames=['True labels'],␣
 ↪colnames=['Predicted labels'])
```

[132]:
```
Predicted labels  abnormal  normal
True labels
abnormal                25       3
normal                  45     501
```

[133]:
```
accuracy_score(true_values_str, predictions_str)
```

[133]:
```
0.9163763066202091
```

[134]:
```
recall_score(true_values_str, predictions_str, average=None)
```

[134]:
```
array([0.89285714, 0.91758242])
```

2 Now, train a model (without class weighting) by first oversampling the *train* data using SMOTE.

-Train a model with the same architecture as the previous one. -Conduct your experiments below and compare the resuls between the weighted model and using SMOTE. Which method was better? Write your conclusions at the end.

[135]:
```
# YOUR CODE HERE

# Define the model.
model_smote = keras.Sequential([
    keras.layers.Dense(units = 16, input_shape=(8,), activation=tf.nn.relu),
    keras.layers.Dense(units = 8, activation=tf.nn.relu),
    keras.layers.Dense(units = 2, activation=tf.nn.softmax)
])
```

```
C:\Users\User\AppData\Roaming\Python\Python313\site-
packages\keras\src\layers\core\dense.py:93: UserWarning: Do not pass an
```

```
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[ ]:

[136]:
```
from imblearn.over_sampling import SMOTE
from tensorflow.keras.utils import to_categorical

sm = SMOTE(
    sampling_strategy="minority",
    random_state=42,
    k_neighbors=5,
)

X_resampled, y_resampled = sm.fit_resample(train_normalized, train_labels)

y_resampled = to_categorical(y_resampled, num_classes=2)
```

[137]:
```
# Define the optimizer. Stochastic Gradient Descent in this case.
optimizer = tf.keras.optimizers.SGD(learning_rate = 0.01)

model_smote.compile(optimizer = optimizer,
              loss = "categorical_crossentropy",
              metrics = ['accuracy'])

# Train the model.
history = model_smote.fit(X_resampled, y_resampled,
                    epochs = 100,
                    validation_split = 0.0,
                    batch_size = 256,
                    verbose = 1)
```

```
Epoch 1/100
5/5             2s 22ms/step -
accuracy: 0.6024 - loss: 0.5609
Epoch 2/100
5/5             0s 22ms/step -
accuracy: 0.6227 - loss: 0.5465
Epoch 3/100
5/5             0s 23ms/step -
accuracy: 0.6478 - loss: 0.5265
Epoch 4/100
5/5             0s 44ms/step -
accuracy: 0.6871 - loss: 0.4998
Epoch 5/100
5/5             0s 16ms/step -
accuracy: 0.7102 - loss: 0.4932
```

```
Epoch 6/100
5/5             0s 16ms/step -
accuracy: 0.7524 - loss: 0.4692
Epoch 7/100
5/5             0s 33ms/step -
accuracy: 0.7897 - loss: 0.4585
Epoch 8/100
5/5             0s 40ms/step -
accuracy: 0.8221 - loss: 0.4462
Epoch 9/100
5/5             0s 22ms/step -
accuracy: 0.8415 - loss: 0.4353
Epoch 10/100
5/5             0s 19ms/step -
accuracy: 0.8376 - loss: 0.4270
Epoch 11/100
5/5             0s 15ms/step -
accuracy: 0.8538 - loss: 0.4313
Epoch 12/100
5/5             0s 17ms/step -
accuracy: 0.8652 - loss: 0.4208
Epoch 13/100
5/5             0s 16ms/step -
accuracy: 0.8768 - loss: 0.4063
Epoch 14/100
5/5             0s 20ms/step -
accuracy: 0.8835 - loss: 0.4047
Epoch 15/100
5/5             0s 16ms/step -
accuracy: 0.8846 - loss: 0.4034
Epoch 16/100
5/5             0s 16ms/step -
accuracy: 0.8998 - loss: 0.3907
Epoch 17/100
5/5             0s 16ms/step -
accuracy: 0.8918 - loss: 0.3840
Epoch 18/100
5/5             0s 16ms/step -
accuracy: 0.8958 - loss: 0.3942
Epoch 19/100
5/5             0s 22ms/step -
accuracy: 0.8983 - loss: 0.3838
Epoch 20/100
5/5             0s 21ms/step -
accuracy: 0.9143 - loss: 0.3680
Epoch 21/100
5/5             0s 20ms/step -
accuracy: 0.9044 - loss: 0.3710
```

```
Epoch 22/100
5/5             0s 21ms/step -
accuracy: 0.9113 - loss: 0.3571
Epoch 23/100
5/5             0s 14ms/step -
accuracy: 0.9149 - loss: 0.3591
Epoch 24/100
5/5             0s 16ms/step -
accuracy: 0.9074 - loss: 0.3678
Epoch 25/100
5/5             0s 15ms/step -
accuracy: 0.9084 - loss: 0.3561
Epoch 26/100
5/5             0s 16ms/step -
accuracy: 0.9164 - loss: 0.3502
Epoch 27/100
5/5             0s 24ms/step -
accuracy: 0.9147 - loss: 0.3499
Epoch 28/100
5/5             0s 25ms/step -
accuracy: 0.9126 - loss: 0.3535
Epoch 29/100
5/5             0s 28ms/step -
accuracy: 0.9282 - loss: 0.3302
Epoch 30/100
5/5             0s 35ms/step -
accuracy: 0.9286 - loss: 0.3276
Epoch 31/100
5/5             0s 19ms/step -
accuracy: 0.9266 - loss: 0.3280
Epoch 32/100
5/5             0s 16ms/step -
accuracy: 0.9246 - loss: 0.3223
Epoch 33/100
5/5             0s 27ms/step -
accuracy: 0.9277 - loss: 0.3248
Epoch 34/100
5/5             0s 16ms/step -
accuracy: 0.9293 - loss: 0.3241
Epoch 35/100
5/5             0s 16ms/step -
accuracy: 0.9306 - loss: 0.3203
Epoch 36/100
5/5             0s 16ms/step -
accuracy: 0.9311 - loss: 0.3177
Epoch 37/100
5/5             0s 24ms/step -
accuracy: 0.9331 - loss: 0.3109
```

```
Epoch 38/100
5/5             0s 14ms/step -
accuracy: 0.9305 - loss: 0.3156
Epoch 39/100
5/5             0s 22ms/step -
accuracy: 0.9324 - loss: 0.3029
Epoch 40/100
5/5             0s 26ms/step -
accuracy: 0.9334 - loss: 0.3029
Epoch 41/100
5/5             0s 25ms/step -
accuracy: 0.9309 - loss: 0.2992
Epoch 42/100
5/5             0s 25ms/step -
accuracy: 0.9293 - loss: 0.3004
Epoch 43/100
5/5             0s 25ms/step -
accuracy: 0.9343 - loss: 0.2869
Epoch 44/100
5/5             0s 25ms/step -
accuracy: 0.9364 - loss: 0.2884
Epoch 45/100
5/5             0s 25ms/step -
accuracy: 0.9364 - loss: 0.2855
Epoch 46/100
5/5             0s 28ms/step -
accuracy: 0.9331 - loss: 0.2804
Epoch 47/100
5/5             0s 31ms/step -
accuracy: 0.9403 - loss: 0.2774
Epoch 48/100
5/5             0s 25ms/step -
accuracy: 0.9392 - loss: 0.2780
Epoch 49/100
5/5             0s 22ms/step -
accuracy: 0.9416 - loss: 0.2635
Epoch 50/100
5/5             0s 16ms/step -
accuracy: 0.9341 - loss: 0.2759
Epoch 51/100
5/5             0s 16ms/step -
accuracy: 0.9362 - loss: 0.2729
Epoch 52/100
5/5             0s 16ms/step -
accuracy: 0.9299 - loss: 0.2782
Epoch 53/100
5/5             0s 15ms/step -
accuracy: 0.9289 - loss: 0.2772
```

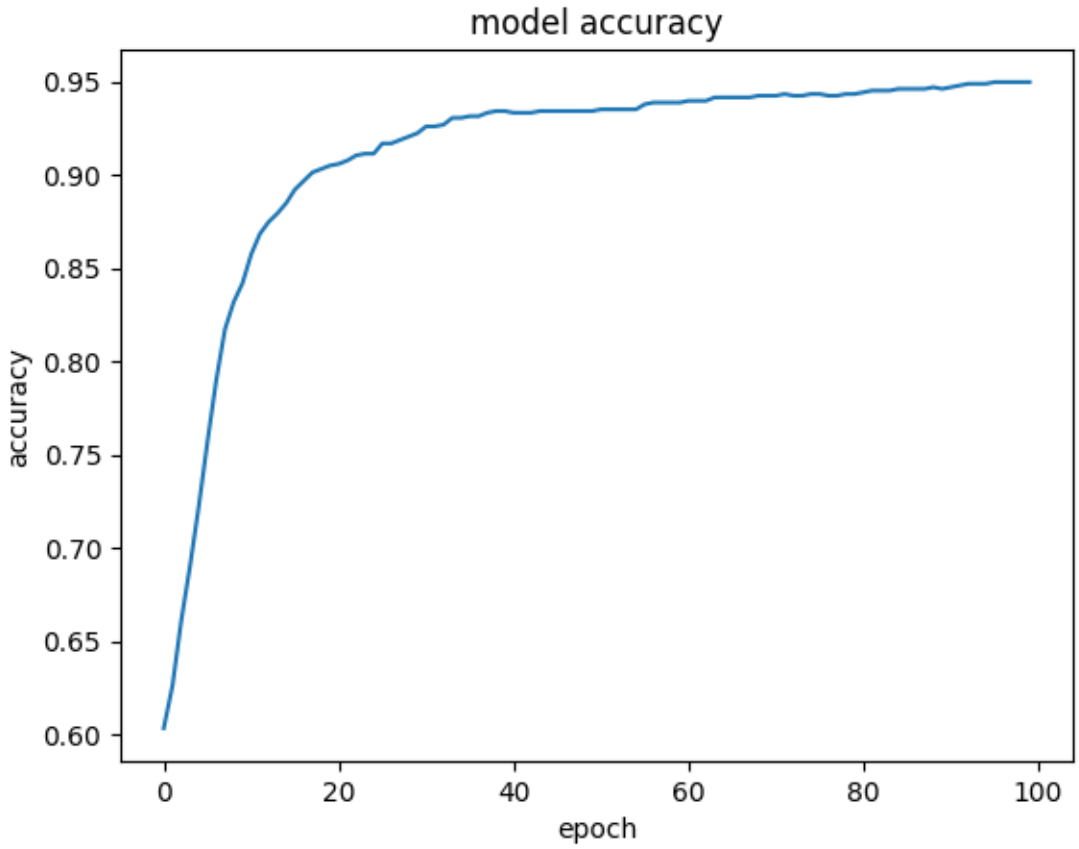
```
Epoch 54/100
5/5             0s 26ms/step -
accuracy: 0.9388 - loss: 0.2584
Epoch 55/100
5/5             0s 28ms/step -
accuracy: 0.9347 - loss: 0.2637
Epoch 56/100
5/5             0s 19ms/step -
accuracy: 0.9374 - loss: 0.2630
Epoch 57/100
5/5             0s 18ms/step -
accuracy: 0.9319 - loss: 0.2678
Epoch 58/100
5/5             0s 22ms/step -
accuracy: 0.9381 - loss: 0.2586
Epoch 59/100
5/5             0s 31ms/step -
accuracy: 0.9404 - loss: 0.2511
Epoch 60/100
5/5             0s 35ms/step -
accuracy: 0.9391 - loss: 0.2505
Epoch 61/100
5/5             0s 19ms/step -
accuracy: 0.9348 - loss: 0.2626
Epoch 62/100
5/5             0s 18ms/step -
accuracy: 0.9389 - loss: 0.2487
Epoch 63/100
5/5             0s 19ms/step -
accuracy: 0.9373 - loss: 0.2500
Epoch 64/100
5/5             0s 19ms/step -
accuracy: 0.9377 - loss: 0.2479
Epoch 65/100
5/5             0s 16ms/step -
accuracy: 0.9426 - loss: 0.2375
Epoch 66/100
5/5             0s 22ms/step -
accuracy: 0.9388 - loss: 0.2414
Epoch 67/100
5/5             0s 29ms/step -
accuracy: 0.9435 - loss: 0.2349
Epoch 68/100
5/5             0s 25ms/step -
accuracy: 0.9415 - loss: 0.2338
Epoch 69/100
5/5             0s 27ms/step -
accuracy: 0.9457 - loss: 0.2255
```

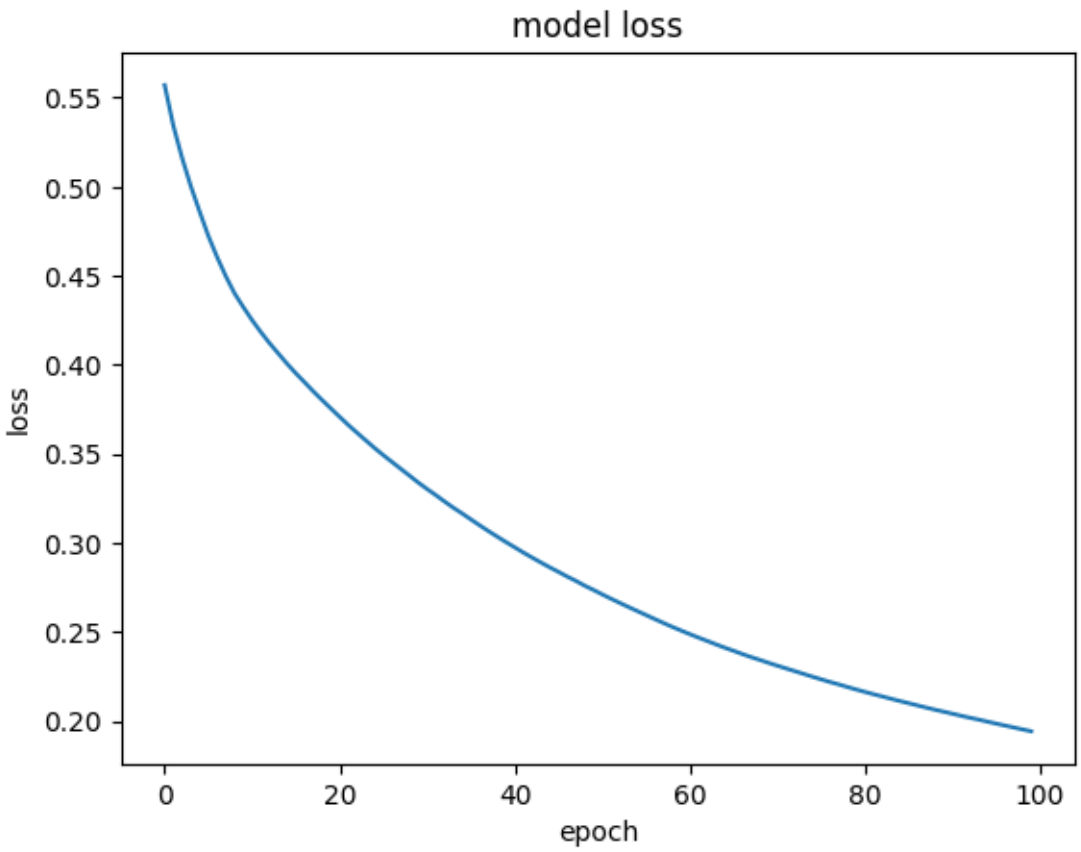
```
Epoch 70/100
5/5             0s 29ms/step -
accuracy: 0.9443 - loss: 0.2276
Epoch 71/100
5/5             0s 31ms/step -
accuracy: 0.9441 - loss: 0.2309
Epoch 72/100
5/5             0s 32ms/step -
accuracy: 0.9345 - loss: 0.2466
Epoch 73/100
5/5             0s 17ms/step -
accuracy: 0.9334 - loss: 0.2371
Epoch 74/100
5/5             0s 16ms/step -
accuracy: 0.9455 - loss: 0.2209
Epoch 75/100
5/5             0s 16ms/step -
accuracy: 0.9439 - loss: 0.2300
Epoch 76/100
5/5             0s 27ms/step -
accuracy: 0.9439 - loss: 0.2213
Epoch 77/100
5/5             0s 45ms/step -
accuracy: 0.9447 - loss: 0.2229
Epoch 78/100
5/5             0s 13ms/step -
accuracy: 0.9391 - loss: 0.2208
Epoch 79/100
5/5             0s 20ms/step -
accuracy: 0.9415 - loss: 0.2235
Epoch 80/100
5/5             0s 28ms/step -
accuracy: 0.9446 - loss: 0.2171
Epoch 81/100
5/5             0s 30ms/step -
accuracy: 0.9429 - loss: 0.2120
Epoch 82/100
5/5             0s 29ms/step -
accuracy: 0.9497 - loss: 0.2154
Epoch 83/100
5/5             0s 29ms/step -
accuracy: 0.9408 - loss: 0.2163
Epoch 84/100
5/5             0s 32ms/step -
accuracy: 0.9449 - loss: 0.2161
Epoch 85/100
5/5             0s 26ms/step -
accuracy: 0.9459 - loss: 0.2094
```

```
Epoch 86/100
5/5             0s 16ms/step -
accuracy: 0.9451 - loss: 0.2128
Epoch 87/100
5/5             0s 31ms/step -
accuracy: 0.9554 - loss: 0.1958
Epoch 88/100
5/5             0s 27ms/step -
accuracy: 0.9468 - loss: 0.2082
Epoch 89/100
5/5             0s 30ms/step -
accuracy: 0.9519 - loss: 0.2009
Epoch 90/100
5/5             0s 26ms/step -
accuracy: 0.9470 - loss: 0.2080
Epoch 91/100
5/5             0s 30ms/step -
accuracy: 0.9504 - loss: 0.2013
Epoch 92/100
5/5             0s 30ms/step -
accuracy: 0.9469 - loss: 0.2118
Epoch 93/100
5/5             0s 18ms/step -
accuracy: 0.9524 - loss: 0.1997
Epoch 94/100
5/5             0s 18ms/step -
accuracy: 0.9514 - loss: 0.2013
Epoch 95/100
5/5             0s 18ms/step -
accuracy: 0.9497 - loss: 0.1985
Epoch 96/100
5/5             0s 17ms/step -
accuracy: 0.9497 - loss: 0.1994
Epoch 97/100
5/5             0s 14ms/step -
accuracy: 0.9508 - loss: 0.1979
Epoch 98/100
5/5             0s 29ms/step -
accuracy: 0.9487 - loss: 0.1887
Epoch 99/100
5/5             0s 27ms/step -
accuracy: 0.9478 - loss: 0.1963
Epoch 100/100
5/5             0s 14ms/step -
accuracy: 0.9507 - loss: 0.1934
```

[138]:
```
# Plot accuracy and loss curves

import matplotlib.pyplot as plt
%matplotlib inline
# summarize history for accuracy
plt.plot(history.history['accuracy'])
#plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
#plt.legend(['train', 'validation'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
#plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
#plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```





[139]:
```
# Evaluate the model on the test set and print the loss and accuracy.
model_smote.evaluate(test_normalized, test_labels) # [loss, accuracy]
```

```
18/18             1s 7ms/step -
accuracy: 0.9367 - loss: 0.2388
```

[139]:
```
[0.2789207100868225, 0.9198606014251709]
```

[140]:
```
predictions = model_smote.predict(test_normalized)
```

```
18/18             0s 10ms/step
```

[141]:
```
# Get the column index with max probability from predictions.
predictions_int = np.argmax(predictions, axis=1)

# Ground truth
true_values_int = np.argmax(test_labels, axis=1)
```

[142]:
```
# Convert back to strings
predictions_str = le.inverse_transform(predictions_int)
```

```
true_values_str = le.inverse_transform(true_values_int)
```

[143]:
```
pd.crosstab(true_values_str, predictions_str, rownames=['True labels'],␣
↪colnames=['Predicted labels'])
```

[143]:
```
Predicted labels  abnormal  normal
True labels
abnormal                23       5
normal                  41     505
```

[144]:
```
accuracy_score(true_values_str, predictions_str)
```

[144]:
```
0.9198606271777003
```

[145]:
```
recall_score(true_values_str, predictions_str, average=None)
```

[145]:
```
array([0.82142857, 0.92490842])
```

3 Conclusions

El uso de SMOTE permitió incrementar artificialmente la cantidad de muestras de la clase minoritaria mediante interpolación sintética, logrando un conjunto de entrenamiento completamente balanceado. Dado que el modelo original utiliza categorical_crossentropy, fue necesario convertir las etiquetas generadas por SMOTE (que son enteros 0/1) nuevamente a formato one-hot para mantener compatibilidad con la arquitectura de salida del clasificador.

En resumen, la integración de SMOTE con la red neuronal se realizó correctamente, preservando la estructura y los requisitos del código original. En cuanto a la comparación entre las técnicas de class weighting y SMOTE, la segunda obtuvo un resultado ligeramente mejor (aunque casi insignificante) en términos de accuracy. Sin embargo, en términos de recall, SMOTE produjo un desempeño apenas superior para la clase mayoritaria, pero notablemente inferior para la clase minoritaria. Esto incrementó la diferencia entre los recalls de ambas clases, reduciendo el balance del modelo al clasificar los casos minoritarios, por lo que preferimos utilizar class weighting para esta situación.