activity 3.1 - abnormal_fish_oversampling

November 17, 2024

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.

[78]:
```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from sklearn.metrics import accuracy_score, recall_score
```

[79]:
```
# Path to the dataset.
filepath = "fishFeatures-4.csv"

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

[80]:
```
dataset.head()
```

[80]:
```
    id   label  f.meanSpeed  f.sdSpeed  f.minSpeed  f.maxSpeed  f.meanAcc  \
0  id1  normal     2.623236   2.228456    0.500000    8.225342  -0.053660
1  id2  normal     5.984859   3.820270    1.414214   15.101738  -0.038705
2  id3  normal    16.608716  14.502042    0.707107   46.424670  -1.000196
3  id5  normal     4.808608   4.137387    0.500000   17.204651  -0.281815
4  id6  normal    17.785747   9.926729    3.354102   44.240818  -0.537534

    f.sdAcc   f.minAcc   f.maxAcc
0  1.839475  -5.532760   3.500000
1  2.660073  -7.273932   7.058594
2 12.890386 -24.320298  30.714624
3  5.228209 -12.204651  15.623512
4 11.272472 -22.178067  21.768613
```

[81]:
```python
# remove id column
dataset = dataset.drop('id', axis=1)
dataset.head()
```

[81]:
```
    label  f.meanSpeed  f.sdSpeed  f.minSpeed  f.maxSpeed  f.meanAcc  \
0  normal     2.623236   2.228456    0.500000    8.225342  -0.053660
1  normal     5.984859   3.820270    1.414214   15.101738  -0.038705
2  normal    16.608716  14.502042    0.707107   46.424670  -1.000196
3  normal     4.808608   4.137387    0.500000   17.204651  -0.281815
4  normal    17.785747   9.926729    3.354102   44.240818  -0.537534

     f.sdAcc    f.minAcc   f.maxAcc
0   1.839475   -5.532760   3.500000
1   2.660073   -7.273932   7.058594
2  12.890386  -24.320298  30.714624
3   5.228209  -12.204651  15.623512
4  11.272472  -22.178067  21.768613
```

[82]:
```python
# Count labels
dataset['label'].value_counts()
```

[82]:
```
label
normal      1093
abnormal      54
Name: count, dtype: int64
```

[83]:
```python
# Shuffle the dataset
from sklearn.utils import shuffle

seed = 1234 #set seed for reproducibility

np.random.seed(seed)

dataset = shuffle(dataset)
```

[84]:
```python
#Select features and class
features = dataset.drop('label', axis=1)

labels = dataset[['label']]

features = features.values.astype(float)

labels = labels.values
```

[85]:
```python
features.shape
```

[85]:
```
(1147, 8)
```

[86]:
```python
# Convert labels to integers.
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
labels_int = le.fit_transform(labels.ravel())
```

[87]:
```python
print(labels[0])
print(labels_int[0])
```

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

[88]:
```python
# One hot encode labels using the to_categorical function of keras.
labels = tf.keras.utils.to_categorical(labels_int, num_classes = 2)
```

[89]:
```python
labels[0:10,:]
```

[89]:
```
array([[0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

[90]:
```python
# 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)
```

[91]:
```python
# 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
```

[92]:
```python
# 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

[93]:
```
# 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)
])
```

```
/opt/anaconda3/envs/tensorflow_env/lib/python3.11/site-
packages/keras/src/layers/core/dense.py:87: 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)
```

[94]:
```
print(model.summary())
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 16)	144
dense_13 (Dense)	(None, 8)	136
dense_14 (Dense)	(None, 2)	18

Total params: 298 (1.16 KB)

Trainable params: 298 (1.16 KB)

Non-trainable params: 0 (0.00 B)

```
None
```

[95]:

```
# 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
```

[96]:

```
# 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            0s 1ms/step -
accuracy: 0.3187 - loss: 2.5136
Epoch 2/100
3/3            0s 1ms/step -
accuracy: 0.2816 - loss: 1.7067
Epoch 3/100
3/3            0s 1ms/step -
accuracy: 0.2569 - loss: 1.0542
Epoch 4/100
3/3            0s 1ms/step -
accuracy: 0.2503 - loss: 0.8066
Epoch 5/100
3/3            0s 1ms/step -
accuracy: 0.2756 - loss: 0.6729
Epoch 6/100
3/3            0s 1ms/step -
accuracy: 0.3077 - loss: 0.5875
```

```
Epoch 7/100
3/3              0s 1ms/step -
accuracy: 0.3662 - loss: 0.5650
Epoch 8/100
3/3              0s 1ms/step -
accuracy: 0.4230 - loss: 0.5175
Epoch 9/100
3/3              0s 1ms/step -
accuracy: 0.4634 - loss: 0.5038
Epoch 10/100
3/3              0s 1ms/step -
accuracy: 0.4858 - loss: 0.4915
Epoch 11/100
3/3              0s 1ms/step -
accuracy: 0.5100 - loss: 0.4922
Epoch 12/100
3/3              0s 1ms/step -
accuracy: 0.5473 - loss: 0.4843
Epoch 13/100
3/3              0s 968us/step -
accuracy: 0.5780 - loss: 0.4769
Epoch 14/100
3/3              0s 1ms/step -
accuracy: 0.6087 - loss: 0.4774
Epoch 15/100
3/3              0s 1ms/step -
accuracy: 0.6124 - loss: 0.4574
Epoch 16/100
3/3              0s 1ms/step -
accuracy: 0.6342 - loss: 0.4611
Epoch 17/100
3/3              0s 1ms/step -
accuracy: 0.6635 - loss: 0.4452
Epoch 18/100
3/3              0s 1ms/step -
accuracy: 0.6738 - loss: 0.4355
Epoch 19/100
3/3              0s 1ms/step -
accuracy: 0.7047 - loss: 0.4372
Epoch 20/100
3/3              0s 1ms/step -
accuracy: 0.7152 - loss: 0.4371
Epoch 21/100
3/3              0s 1ms/step -
accuracy: 0.7227 - loss: 0.4222
Epoch 22/100
3/3              0s 1ms/step -
accuracy: 0.7412 - loss: 0.4196
```

```
Epoch 23/100
3/3            0s 1ms/step -
accuracy: 0.7522 - loss: 0.4068
Epoch 24/100
3/3            0s 1ms/step -
accuracy: 0.7505 - loss: 0.4113
Epoch 25/100
3/3            0s 1ms/step -
accuracy: 0.7594 - loss: 0.4007
Epoch 26/100
3/3            0s 1ms/step -
accuracy: 0.7613 - loss: 0.3986
Epoch 27/100
3/3            0s 1ms/step -
accuracy: 0.7727 - loss: 0.4040
Epoch 28/100
3/3            0s 1ms/step -
accuracy: 0.7800 - loss: 0.4074
Epoch 29/100
3/3            0s 1ms/step -
accuracy: 0.7854 - loss: 0.3929
Epoch 30/100
3/3            0s 1ms/step -
accuracy: 0.8016 - loss: 0.3982
Epoch 31/100
3/3            0s 1ms/step -
accuracy: 0.7976 - loss: 0.3934
Epoch 32/100
3/3            0s 1ms/step -
accuracy: 0.8055 - loss: 0.3832
Epoch 33/100
3/3            0s 1ms/step -
accuracy: 0.8059 - loss: 0.3764
Epoch 34/100
3/3            0s 1ms/step -
accuracy: 0.8130 - loss: 0.3759
Epoch 35/100
3/3            0s 1ms/step -
accuracy: 0.8057 - loss: 0.3674
Epoch 36/100
3/3            0s 1ms/step -
accuracy: 0.8174 - loss: 0.3689
Epoch 37/100
3/3            0s 1ms/step -
accuracy: 0.8065 - loss: 0.3725
Epoch 38/100
3/3            0s 973us/step -
accuracy: 0.8120 - loss: 0.3579
```

```
Epoch 39/100
3/3            0s 914us/step -
accuracy: 0.8288 - loss: 0.3514
Epoch 40/100
3/3            0s 1ms/step -
accuracy: 0.8262 - loss: 0.3447
Epoch 41/100
3/3            0s 1ms/step -
accuracy: 0.8327 - loss: 0.3443
Epoch 42/100
3/3            0s 1ms/step -
accuracy: 0.8325 - loss: 0.3479
Epoch 43/100
3/3            0s 1ms/step -
accuracy: 0.8420 - loss: 0.3438
Epoch 44/100
3/3            0s 1ms/step -
accuracy: 0.8453 - loss: 0.3377
Epoch 45/100
3/3            0s 1ms/step -
accuracy: 0.8526 - loss: 0.3298
Epoch 46/100
3/3            0s 1ms/step -
accuracy: 0.8612 - loss: 0.3184
Epoch 47/100
3/3            0s 1ms/step -
accuracy: 0.8634 - loss: 0.3265
Epoch 48/100
3/3            0s 1ms/step -
accuracy: 0.8575 - loss: 0.3326
Epoch 49/100
3/3            0s 1ms/step -
accuracy: 0.8574 - loss: 0.3246
Epoch 50/100
3/3            0s 1ms/step -
accuracy: 0.8594 - loss: 0.3326
Epoch 51/100
3/3            0s 1ms/step -
accuracy: 0.8550 - loss: 0.3295
Epoch 52/100
3/3            0s 1ms/step -
accuracy: 0.8584 - loss: 0.3140
Epoch 53/100
3/3            0s 1ms/step -
accuracy: 0.8545 - loss: 0.3213
Epoch 54/100
3/3            0s 1ms/step -
accuracy: 0.8647 - loss: 0.3091
```

```
Epoch 55/100
3/3            0s 1ms/step -
accuracy: 0.8621 - loss: 0.3235
Epoch 56/100
3/3            0s 1ms/step -
accuracy: 0.8649 - loss: 0.3046
Epoch 57/100
3/3            0s 1ms/step -
accuracy: 0.8711 - loss: 0.3038
Epoch 58/100
3/3            0s 1ms/step -
accuracy: 0.8677 - loss: 0.3025
Epoch 59/100
3/3            0s 1ms/step -
accuracy: 0.8711 - loss: 0.3040
Epoch 60/100
3/3            0s 1ms/step -
accuracy: 0.8714 - loss: 0.2969
Epoch 61/100
3/3            0s 1ms/step -
accuracy: 0.8717 - loss: 0.2988
Epoch 62/100
3/3            0s 1ms/step -
accuracy: 0.8698 - loss: 0.3034
Epoch 63/100
3/3            0s 1ms/step -
accuracy: 0.8713 - loss: 0.3007
Epoch 64/100
3/3            0s 1ms/step -
accuracy: 0.8712 - loss: 0.2955
Epoch 65/100
3/3            0s 1ms/step -
accuracy: 0.8736 - loss: 0.2933
Epoch 66/100
3/3            0s 1ms/step -
accuracy: 0.8716 - loss: 0.3064
Epoch 67/100
3/3            0s 1ms/step -
accuracy: 0.8769 - loss: 0.2969
Epoch 68/100
3/3            0s 1ms/step -
accuracy: 0.8769 - loss: 0.2960
Epoch 69/100
3/3            0s 1ms/step -
accuracy: 0.8725 - loss: 0.2996
Epoch 70/100
3/3            0s 1ms/step -
accuracy: 0.8789 - loss: 0.2861
```

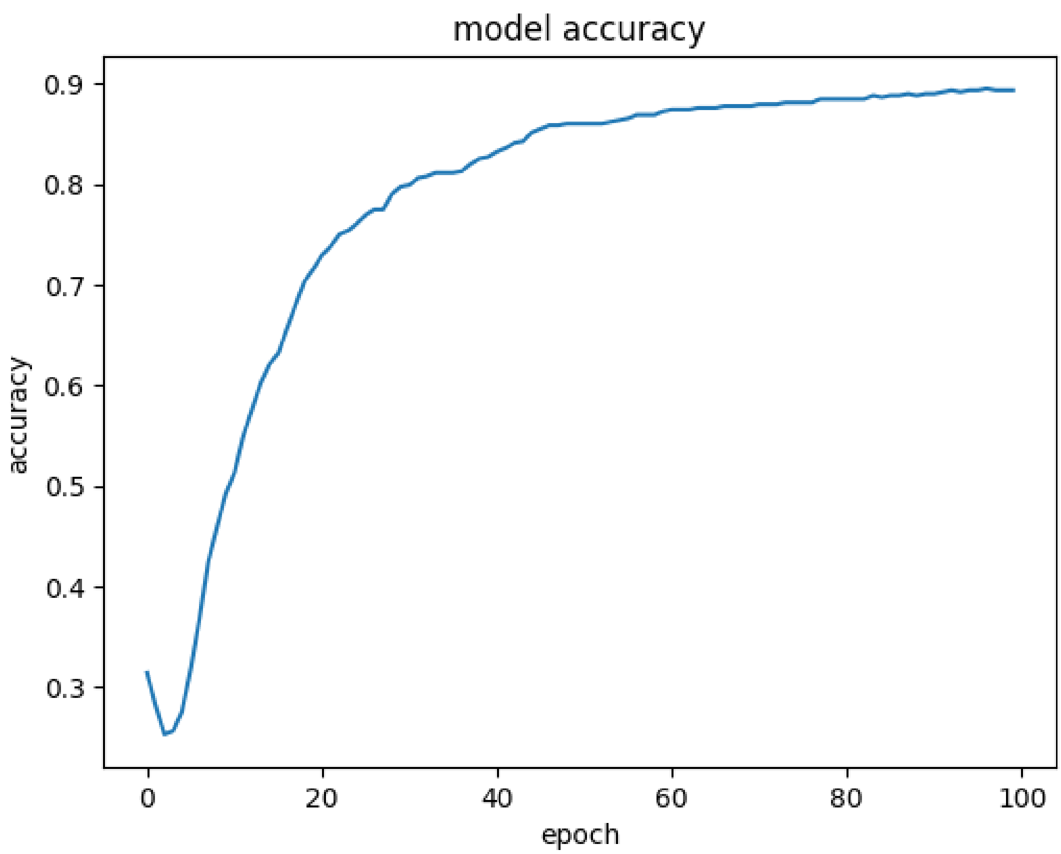
```
Epoch 71/100
3/3            0s 1ms/step -
accuracy: 0.8783 - loss: 0.2791
Epoch 72/100
3/3            0s 1ms/step -
accuracy: 0.8734 - loss: 0.2979
Epoch 73/100
3/3            0s 1ms/step -
accuracy: 0.8841 - loss: 0.2754
Epoch 74/100
3/3            0s 1ms/step -
accuracy: 0.8767 - loss: 0.2909
Epoch 75/100
3/3            0s 1ms/step -
accuracy: 0.8860 - loss: 0.2627
Epoch 76/100
3/3            0s 1ms/step -
accuracy: 0.8874 - loss: 0.2800
Epoch 77/100
3/3            0s 1ms/step -
accuracy: 0.8801 - loss: 0.2851
Epoch 78/100
3/3            0s 1ms/step -
accuracy: 0.8833 - loss: 0.2745
Epoch 79/100
3/3            0s 1ms/step -
accuracy: 0.8765 - loss: 0.2658
Epoch 80/100
3/3            0s 1ms/step -
accuracy: 0.8809 - loss: 0.2689
Epoch 81/100
3/3            0s 1ms/step -
accuracy: 0.8853 - loss: 0.2811
Epoch 82/100
3/3            0s 1ms/step -
accuracy: 0.8858 - loss: 0.2612
Epoch 83/100
3/3            0s 1ms/step -
accuracy: 0.8853 - loss: 0.2758
Epoch 84/100
3/3            0s 1ms/step -
accuracy: 0.8919 - loss: 0.2544
Epoch 85/100
3/3            0s 1ms/step -
accuracy: 0.8837 - loss: 0.2653
Epoch 86/100
3/3            0s 1ms/step -
accuracy: 0.8885 - loss: 0.2688
```

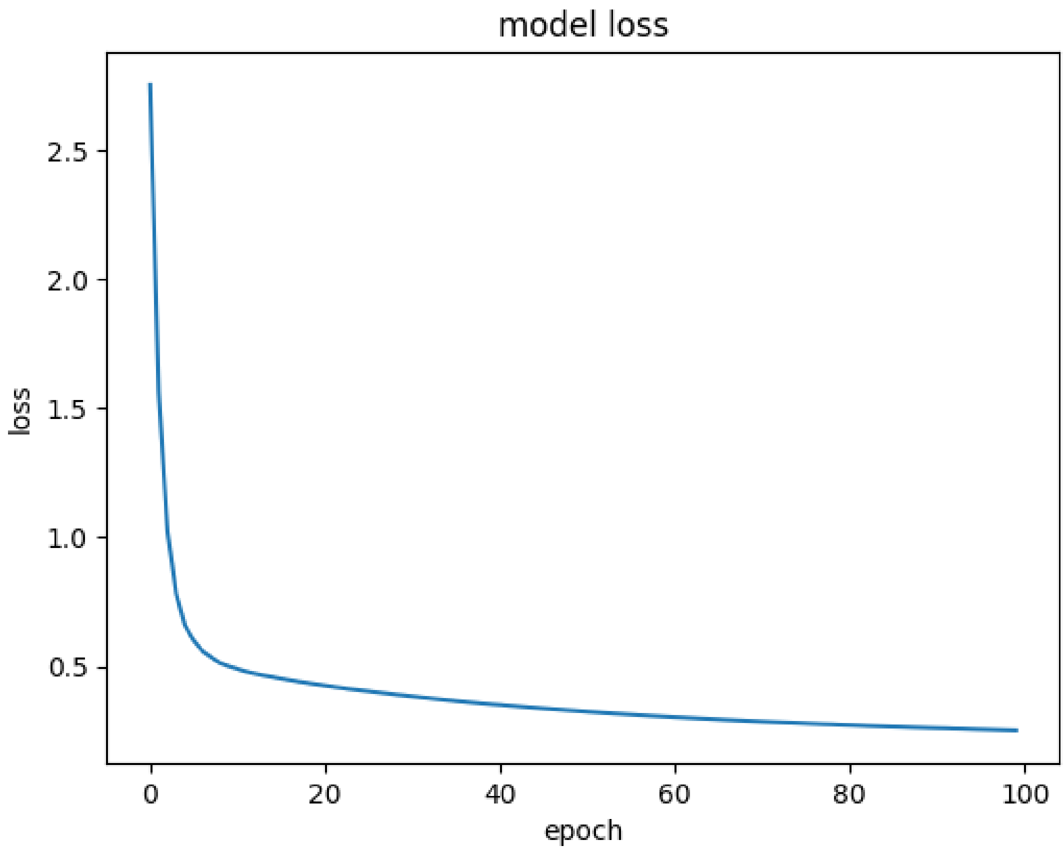
```
Epoch 87/100
3/3              0s 1ms/step -
accuracy: 0.8875 - loss: 0.2536
Epoch 88/100
3/3              0s 1ms/step -
accuracy: 0.8933 - loss: 0.2668
Epoch 89/100
3/3              0s 1ms/step -
accuracy: 0.8924 - loss: 0.2684
Epoch 90/100
3/3              0s 1ms/step -
accuracy: 0.8908 - loss: 0.2592
Epoch 91/100
3/3              0s 1ms/step -
accuracy: 0.8830 - loss: 0.2715
Epoch 92/100
3/3              0s 1ms/step -
accuracy: 0.8849 - loss: 0.2547
Epoch 93/100
3/3              0s 1ms/step -
accuracy: 0.8887 - loss: 0.2609
Epoch 94/100
3/3              0s 1ms/step -
accuracy: 0.8927 - loss: 0.2730
Epoch 95/100
3/3              0s 1ms/step -
accuracy: 0.8979 - loss: 0.2551
Epoch 96/100
3/3              0s 1ms/step -
accuracy: 0.8926 - loss: 0.2541
Epoch 97/100
3/3              0s 963us/step -
accuracy: 0.8949 - loss: 0.2486
Epoch 98/100
3/3              0s 1ms/step -
accuracy: 0.8911 - loss: 0.2582
Epoch 99/100
3/3              0s 1ms/step -
accuracy: 0.8896 - loss: 0.2496
Epoch 100/100
3/3              0s 1ms/step -
accuracy: 0.8979 - loss: 0.2493
```

[97]:
```python
# 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()
```





[98]:
```
# Evaluate the model on the test set and print the loss and accuracy.
model.evaluate(test_normalized, test_labels) # [loss, accuracy]
```

```
18/18                0s 555us/step -
accuracy: 0.9022 - loss: 0.2825
```

[98]:
```
[0.318112313747406, 0.891986072063446]
```

[99]:
```
# Make predictions on the test set.
predictions = model.predict(test_normalized)
```

```
18/18                0s 1ms/step
```

[100]:
```
# Print the first 5 predictions.
predictions[0:5]
```

[100]:
```
array([[0.12261198, 0.8773881 ],
       [0.34875438, 0.65124565],
       [0.9490538 , 0.05094618],
       [0.07122894, 0.928771  ],
       [0.07884924, 0.92115074]], dtype=float32)
```

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

[101]:
```
# 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)
```

[102]:
```
# Convert back to strings
predictions_str = le.inverse_transform(predictions_int)

true_values_str = le.inverse_transform(true_values_int)
```

[103]:
```
pd.crosstab(true_values_str, predictions_str, rownames=['True labels'],␣
 ↪colnames=['Predicted labels'])
```

[103]:
```
Predicted labels  abnormal  normal
True labels
abnormal                25       3
normal                  59     487
```

[104]:
```
accuracy_score(true_values_str, predictions_str)
```

[104]: 0.89198606271777

[105]:
```
recall_score(true_values_str, predictions_str, average=None)
```

[105]: array([0.89285714, 0.89194139])

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.

[106]:
```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=1234)
X_train_smote, y_train_smote = smote.fit_resample(train_features, train_labels)
```

[107]:
```
# YOUR CODE HERE

# Define the model.
model2 = 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)
])
```

```
/opt/anaconda3/envs/tensorflow_env/lib/python3.11/site-
packages/keras/src/layers/core/dense.py:87: 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)
```

[108]:

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

model2.compile(optimizer = optimizer,
              loss = "sparse_categorical_crossentropy",
              metrics = ['accuracy'])

# Train the model.
history = model2.fit(X_train_smote, y_train_smote,
                    epochs = 100,
                    validation_split = 0.0,
                    batch_size = 256,
                    verbose = 1)
```

```
Epoch 1/100
5/5               0s 888us/step -
accuracy: 0.5235 - loss: 3.2685
Epoch 2/100
5/5               0s 819us/step -
accuracy: 0.4960 - loss: 0.9806
Epoch 3/100
5/5               0s 825us/step -
accuracy: 0.4949 - loss: 0.5769
Epoch 4/100
5/5               0s 820us/step -
accuracy: 0.5149 - loss: 0.5436
Epoch 5/100
5/5               0s 763us/step -
accuracy: 0.5477 - loss: 0.5311
Epoch 6/100
5/5               0s 813us/step -
accuracy: 0.5603 - loss: 0.5320
Epoch 7/100
5/5               0s 788us/step -
accuracy: 0.5937 - loss: 0.5264
Epoch 8/100
5/5               0s 781us/step -
accuracy: 0.6274 - loss: 0.5353
```

```
Epoch 9/100
5/5            0s 842us/step -
accuracy: 0.6222 - loss: 0.5212
Epoch 10/100
5/5            0s 723us/step -
accuracy: 0.6539 - loss: 0.5218
Epoch 11/100
5/5            0s 807us/step -
accuracy: 0.6643 - loss: 0.5077
Epoch 12/100
5/5            0s 800us/step -
accuracy: 0.7018 - loss: 0.5034
Epoch 13/100
5/5            0s 980us/step -
accuracy: 0.7112 - loss: 0.5034
Epoch 14/100
5/5            0s 819us/step -
accuracy: 0.7230 - loss: 0.5007
Epoch 15/100
5/5            0s 799us/step -
accuracy: 0.7229 - loss: 0.4869
Epoch 16/100
5/5            0s 823us/step -
accuracy: 0.7493 - loss: 0.4906
Epoch 17/100
5/5            0s 751us/step -
accuracy: 0.7541 - loss: 0.4788
Epoch 18/100
5/5            0s 702us/step -
accuracy: 0.7763 - loss: 0.4800
Epoch 19/100
5/5            0s 762us/step -
accuracy: 0.7348 - loss: 0.4767
Epoch 20/100
5/5            0s 699us/step -
accuracy: 0.7670 - loss: 0.4790
Epoch 21/100
5/5            0s 688us/step -
accuracy: 0.7678 - loss: 0.4664
Epoch 22/100
5/5            0s 773us/step -
accuracy: 0.7705 - loss: 0.4599
Epoch 23/100
5/5            0s 686us/step -
accuracy: 0.7965 - loss: 0.4595
Epoch 24/100
5/5            0s 811us/step -
accuracy: 0.8024 - loss: 0.4605
```

```
Epoch 25/100
5/5             0s 809us/step -
accuracy: 0.7902 - loss: 0.4529
Epoch 26/100
5/5             0s 693us/step -
accuracy: 0.8114 - loss: 0.4462
Epoch 27/100
5/5             0s 702us/step -
accuracy: 0.8155 - loss: 0.4406
Epoch 28/100
5/5             0s 728us/step -
accuracy: 0.8371 - loss: 0.4390
Epoch 29/100
5/5             0s 715us/step -
accuracy: 0.8463 - loss: 0.4347
Epoch 30/100
5/5             0s 789us/step -
accuracy: 0.8365 - loss: 0.4390
Epoch 31/100
5/5             0s 769us/step -
accuracy: 0.8427 - loss: 0.4320
Epoch 32/100
5/5             0s 752us/step -
accuracy: 0.8520 - loss: 0.4363
Epoch 33/100
5/5             0s 772us/step -
accuracy: 0.8523 - loss: 0.4269
Epoch 34/100
5/5             0s 751us/step -
accuracy: 0.8690 - loss: 0.4183
Epoch 35/100
5/5             0s 725us/step -
accuracy: 0.8486 - loss: 0.4207
Epoch 36/100
5/5             0s 661us/step -
accuracy: 0.8516 - loss: 0.4229
Epoch 37/100
5/5             0s 795us/step -
accuracy: 0.8540 - loss: 0.4094
Epoch 38/100
5/5             0s 789us/step -
accuracy: 0.8823 - loss: 0.4106
Epoch 39/100
5/5             0s 774us/step -
accuracy: 0.8755 - loss: 0.4082
Epoch 40/100
5/5             0s 797us/step -
accuracy: 0.8689 - loss: 0.4107
```

```
Epoch 41/100
5/5            0s 723us/step -
accuracy: 0.8565 - loss: 0.4042
Epoch 42/100
5/5            0s 716us/step -
accuracy: 0.8690 - loss: 0.4050
Epoch 43/100
5/5            0s 776us/step -
accuracy: 0.8713 - loss: 0.3981
Epoch 44/100
5/5            0s 724us/step -
accuracy: 0.8746 - loss: 0.4002
Epoch 45/100
5/5            0s 877us/step -
accuracy: 0.8804 - loss: 0.3929
Epoch 46/100
5/5            0s 874us/step -
accuracy: 0.8687 - loss: 0.3858
Epoch 47/100
5/5            0s 674us/step -
accuracy: 0.8789 - loss: 0.3850
Epoch 48/100
5/5            0s 835us/step -
accuracy: 0.8907 - loss: 0.3891
Epoch 49/100
5/5            0s 956us/step -
accuracy: 0.9069 - loss: 0.3802
Epoch 50/100
5/5            0s 856us/step -
accuracy: 0.8792 - loss: 0.3771
Epoch 51/100
5/5            0s 730us/step -
accuracy: 0.8826 - loss: 0.3786
Epoch 52/100
5/5            0s 874us/step -
accuracy: 0.8924 - loss: 0.3778
Epoch 53/100
5/5            0s 703us/step -
accuracy: 0.8921 - loss: 0.3756
Epoch 54/100
5/5            0s 943us/step -
accuracy: 0.8865 - loss: 0.3693
Epoch 55/100
5/5            0s 800us/step -
accuracy: 0.8914 - loss: 0.3720
Epoch 56/100
5/5            0s 748us/step -
accuracy: 0.8957 - loss: 0.3756
```

```
Epoch 57/100
5/5            0s 885us/step -
accuracy: 0.9135 - loss: 0.3613
Epoch 58/100
5/5            0s 802us/step -
accuracy: 0.9092 - loss: 0.3648
Epoch 59/100
5/5            0s 753us/step -
accuracy: 0.8791 - loss: 0.3693
Epoch 60/100
5/5            0s 777us/step -
accuracy: 0.9038 - loss: 0.3561
Epoch 61/100
5/5            0s 758us/step -
accuracy: 0.9003 - loss: 0.3624
Epoch 62/100
5/5            0s 814us/step -
accuracy: 0.8975 - loss: 0.3539
Epoch 63/100
5/5            0s 873us/step -
accuracy: 0.9196 - loss: 0.3613
Epoch 64/100
5/5            0s 797us/step -
accuracy: 0.9142 - loss: 0.3560
Epoch 65/100
5/5            0s 896us/step -
accuracy: 0.8988 - loss: 0.3611
Epoch 66/100
5/5            0s 733us/step -
accuracy: 0.9076 - loss: 0.3504
Epoch 67/100
5/5            0s 797us/step -
accuracy: 0.9206 - loss: 0.3425
Epoch 68/100
5/5            0s 4ms/step -
accuracy: 0.9231 - loss: 0.3474
Epoch 69/100
5/5            0s 840us/step -
accuracy: 0.8960 - loss: 0.3480
Epoch 70/100
5/5            0s 874us/step -
accuracy: 0.9033 - loss: 0.3444
Epoch 71/100
5/5            0s 824us/step -
accuracy: 0.9035 - loss: 0.3426
Epoch 72/100
5/5            0s 803us/step -
accuracy: 0.9100 - loss: 0.3401
```

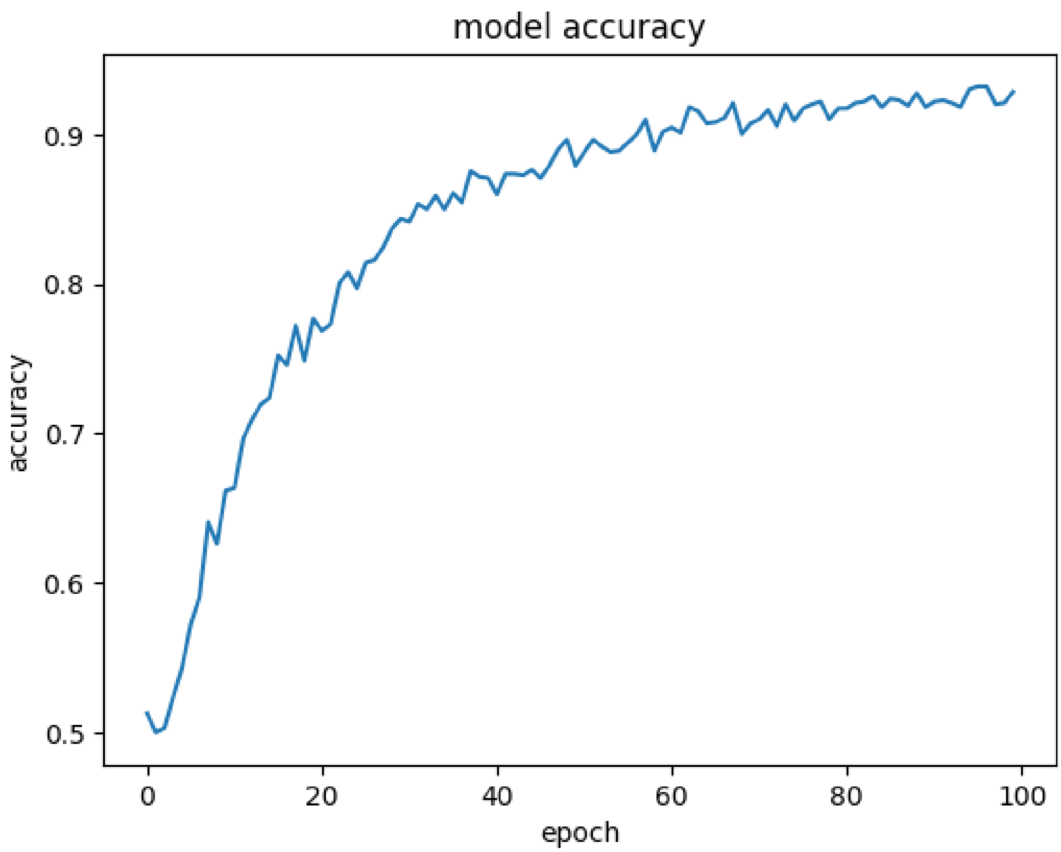
```
Epoch 73/100
5/5             0s 842us/step -
accuracy: 0.9018 - loss: 0.3335
Epoch 74/100
5/5             0s 792us/step -
accuracy: 0.9220 - loss: 0.3350
Epoch 75/100
5/5             0s 781us/step -
accuracy: 0.9092 - loss: 0.3324
Epoch 76/100
5/5             0s 946us/step -
accuracy: 0.9133 - loss: 0.3354
Epoch 77/100
5/5             0s 826us/step -
accuracy: 0.9271 - loss: 0.3313
Epoch 78/100
5/5             0s 827us/step -
accuracy: 0.9286 - loss: 0.3343
Epoch 79/100
5/5             0s 809us/step -
accuracy: 0.9046 - loss: 0.3226
Epoch 80/100
5/5             0s 825us/step -
accuracy: 0.9172 - loss: 0.3274
Epoch 81/100
5/5             0s 818us/step -
accuracy: 0.9212 - loss: 0.3214
Epoch 82/100
5/5             0s 807us/step -
accuracy: 0.9219 - loss: 0.3299
Epoch 83/100
5/5             0s 784us/step -
accuracy: 0.9249 - loss: 0.3246
Epoch 84/100
5/5             0s 795us/step -
accuracy: 0.9238 - loss: 0.3268
Epoch 85/100
5/5             0s 767us/step -
accuracy: 0.9150 - loss: 0.3232
Epoch 86/100
5/5             0s 825us/step -
accuracy: 0.9303 - loss: 0.3144
Epoch 87/100
5/5             0s 816us/step -
accuracy: 0.9244 - loss: 0.3093
Epoch 88/100
5/5             0s 787us/step -
accuracy: 0.9118 - loss: 0.3225
```

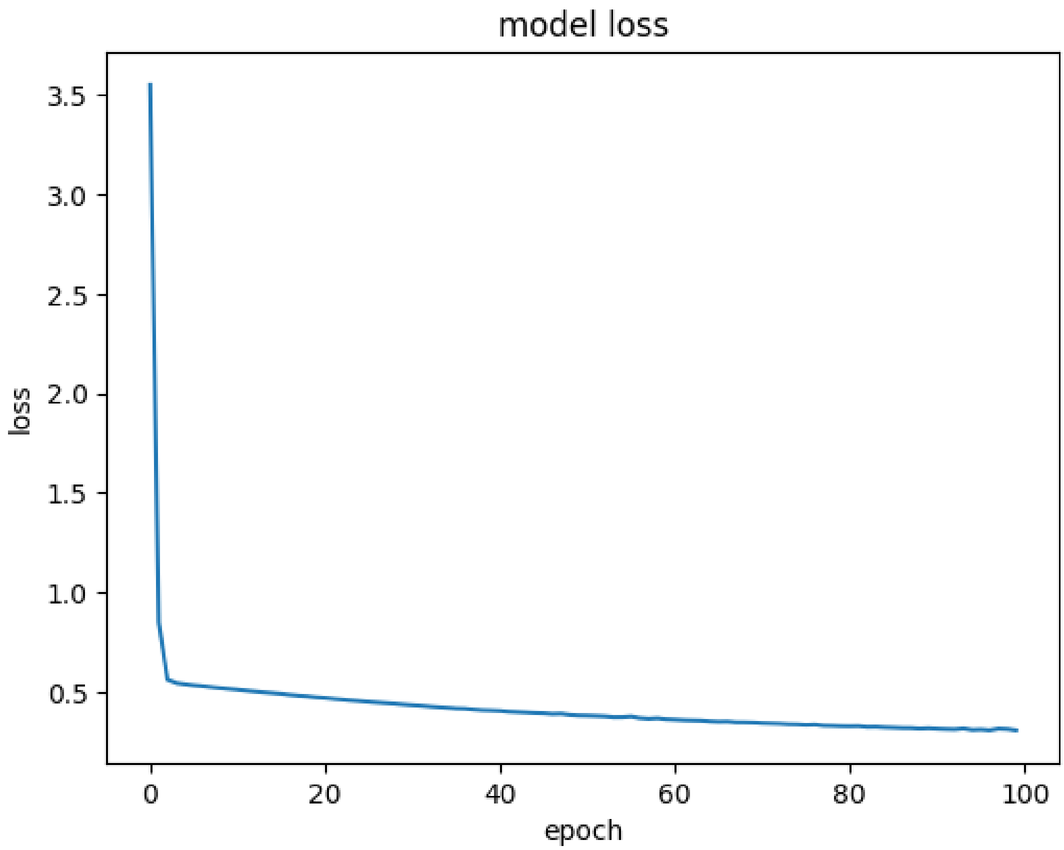
```
Epoch 89/100
5/5            0s 704us/step -
accuracy: 0.9244 - loss: 0.3204
Epoch 90/100
5/5            0s 741us/step -
accuracy: 0.9186 - loss: 0.3121
Epoch 91/100
5/5            0s 728us/step -
accuracy: 0.9207 - loss: 0.3087
Epoch 92/100
5/5            0s 761us/step -
accuracy: 0.9189 - loss: 0.3142
Epoch 93/100
5/5            0s 766us/step -
accuracy: 0.9198 - loss: 0.3071
Epoch 94/100
5/5            0s 784us/step -
accuracy: 0.9178 - loss: 0.3187
Epoch 95/100
5/5            0s 768us/step -
accuracy: 0.9313 - loss: 0.3148
Epoch 96/100
5/5            0s 768us/step -
accuracy: 0.9301 - loss: 0.3140
Epoch 97/100
5/5            0s 767us/step -
accuracy: 0.9328 - loss: 0.3029
Epoch 98/100
5/5            0s 728us/step -
accuracy: 0.9254 - loss: 0.3050
Epoch 99/100
5/5            0s 824us/step -
accuracy: 0.9143 - loss: 0.3221
Epoch 100/100
5/5            0s 744us/step -
accuracy: 0.9214 - loss: 0.3075
```

[109]:

```
# 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()
```





[110]:
```
test_labels_int = np.argmax(test_labels, axis=1)
model2.evaluate(test_normalized, test_labels_int) # [loss, accuracy]
```

```
18/18              0s 503us/step -
accuracy: 0.9653 - loss: 0.3006
```

[110]: [0.307842493057251, 0.9581881761550903]

[111]:
```
predictions = model2.predict(test_normalized)
```

```
18/18              0s 1ms/step
```

[112]:
```
# 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)
```

[113]:
```
# Convert back to strings
predictions_str = le.inverse_transform(predictions_int)
```

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

[114]:
```
pd.crosstab(true_values_str, predictions_str, rownames=['True labels'],␣
 ↪colnames=['Predicted labels'])
```

[114]:
```
Predicted labels  abnormal  normal
True labels
abnormal                 6      22
normal                   2     544
```

[115]:
```
accuracy_score(true_values_str, predictions_str)
```

[115]:
```
0.9581881533101045
```

[116]:
```
recall_score(true_values_str, predictions_str, average=None)
```

[116]:
```
array([0.21428571, 0.996337  ])
```

3 Conclusions

El trabajo realizado se enfocó en analizar comportamientos anormales de peces mediante un conjunto de datos desbalanceado, en el que predominaban ejemplos de comportamientos normales (1093) frente a anormales (54). Para abordar esta situación, se implementaron dos enfoques: la ponderación de clases y el oversampling mediante SMOTE, evaluando cuál ofrecía mejores resultados en términos de precisión y capacidad para identificar ambas clases.

Con el modelo ponderado, los pesos asignados equilibraron el impacto de ambas clases en la función de pérdida, permitiendo que el modelo lograra una precisión final del 89.2% en el conjunto de prueba. Este enfoque destacó por su balance entre clases, logrando un recall cercano al 89% tanto para comportamientos normales como anormales. Esto indica que el modelo fue capaz de identificar correctamente una proporción significativa de ejemplos en ambas categorías, un aspecto crucial cuando se busca tratar ambas clases con igual relevancia.

En contraste, el modelo que utilizó SMOTE generó datos sintéticos para equilibrar las clases antes del entrenamiento, lo que resultó en una precisión más alta, alcanzando un 95.8%. Sin embargo, este enfoque mostró un desequilibrio en el recall: mientras que para los comportamientos normales fue de un sobresaliente 99.6%, para los anormales fue considerablemente bajo, apenas alcanzando un 21.4%. Esto sugiere que el modelo tendió a favorecer los datos mayoritarios, subestimando la importancia de la clase minoritaria.

Los gráficos de precisión y pérdida reflejaron una convergencia adecuada en ambos casos, aunque el modelo entrenado con SMOTE mostró una curva de aprendizaje más rápida. Sin embargo, los resultados confirman que este método, aunque mejora la precisión global, no es óptimo cuando la identificación precisa de la clase minoritaria es prioritaria.

En conclusión, la elección del enfoque depende del objetivo del análisis. Si es fundamental identificar tanto los comportamientos normales como los anormales con igual importancia, la ponderación de clases es la opción más adecuada. Por otro lado, si el objetivo principal es maximizar la precisión general, el oversampling con SMOTE puede ser preferible, aunque a costa de sacrificar la capacidad para detectar eventos raros. Este trabajo evidencia cómo la elección de técnicas de preprocesamiento y entrenamiento afecta directamente el desempeño del modelo en escenarios desbalanceados, subrayando la importancia de considerar las prioridades del análisis antes de decidir un enfoque.

[ ]:

```

```