✓ 100 XP

Exercise - Train and evaluate multiclass classification models

10 minutes

Sandbox activated! Time remaining: 1 hr 24 min

You have used 3 of 10 sandboxes for today. More sandboxes will be available tomorrow.

learn-notebooks-e3dfdcb1-99d0-4066-b50a-c4ff... Py38_default

Multiclass Classification

In the last notebook, we looked at binary classification. This works well when the data observations belong to one of two classes or categories, such as "True" or "False". When the data can be categorized into more than two classes, you must use a multiclass classification algorithm.

Multiclass classification can be thought of as a combination of multiple binary classifiers. There are two ways in which you approach the problem:

- **One vs Rest (OVR)**, in which a classifier is created for each possible class value, with a positive outcome for cases where the prediction is *this* class, and negative predictions for cases where the prediction is any other class. A classification problem with four possible shape classes (*square*, *circle*, *triangle*, *hexagon*) would require four classifiers that predict:
  - *square* or not
  - *circle* or not
  - *triangle* or not
  - *hexagon* or not
- **One vs One (OVO)**, in which a classifier for each possible pair of classes is created. The classification problem with four shape classes would require the following binary classifiers:
  - *square* or *circle*
  - *square* or *triangle*

- *square* or *hexagon*
- *circle* or *triangle*
- *circle* or *hexagon*
- *triangle* or *hexagon*

In both approaches, the overall model that combines the classifiers generates a vector of predictions in which the probabilities generated from the individual binary classifiers are used to determine which class to predict.

Fortunately, in most machine learning frameworks, including scikit-learn, implementing a multiclass classification model is not significantly more complex than binary classification - and in most cases, the estimators used for binary classification implicitly support multiclass classification by abstracting an OVR algorithm, an OVO algorithm, or by allowing a choice of either.

> **More Information**: To learn more about estimator support for multiclass classification in Scikit-Learn, see the Scikit-Learn documentation .

Explore the data

Let's start by examining a dataset that contains observations of multiple classes. We'll use a dataset that contains observations of three different species of penguin.

> **Citation**: The penguins dataset used in the this exercise is a subset of data collected and made available by Dr. Kristen Gorman and the Palmer Station, Antarctica LTER , a member of the Long Term Ecological Research Network .

```
import pandas as pd

# load the training dataset
!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to
penguins = pd.read_csv('penguins.csv')

# Display a random sample of 10 observations
sample = penguins.sample(10)
sample
```

[1] ✓ 4 sec

```
--2022-03-13 00:32:38--
https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-
machine-learning/main/Data/ml-basics/penguins.csv
Resolving raw.githubusercontent.com... 185.199.108.133, 185.199.111.133,
185.199.110.133, ...
Connecting to raw.githubusercontent.com|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7086 (6.9K) [text/plain]
Saving to: ‘penguins.csv’
```

```
penguins.csv        100%[===================>]   6.92K  --.-KB/s    in 0s


2022-03-13 00:32:38 (55.2 MB/s) - ‘penguins.csv’ saved [7086/7086]
```

The dataset contains the following columns:

- **CulmenLength**: The length in mm of the penguin's culmen (bill).
- **CulmenDepth**: The depth in mm of the penguin's culmen.
- **FlipperLength**: The length in mm of the penguin's flipper.
- **BodyMass**: The body mass of the penguin in grams.
- **Species**: An integer value that represents the species of the penguin.

The **Species** column is the label we want to train a model to predict. The dataset includes three possible species, which are encoded as 0, 1, and 2. The actual species names are revealed by the code below:

```
penguin_classes = ['Adelie', 'Gentoo', 'Chinstrap']
print(sample.columns[0:5].values, 'SpeciesName')
for index, row in penguins.sample(10).iterrows():
    print('[',row[0], row[1], row[2], row[3], int(row[4]),']',penguin_class
```

[2] ✓ <1 sec

```
['CulmenLength' 'CulmenDepth' 'FlipperLength' 'BodyMass' 'Species']

SpeciesName

[ 46.5 14.4 217.0 4900.0 1 ] Gentoo
[ 44.1 18.0 210.0 4000.0 0 ] Adelie
[ 45.6 19.4 194.0 3525.0 2 ] Chinstrap
[ 36.0 17.8 195.0 3450.0 0 ] Adelie
[ 46.5 14.5 213.0 4400.0 1 ] Gentoo
[ 38.7 19.0 195.0 3450.0 0 ] Adelie
[ 42.2 18.5 180.0 3550.0 0 ] Adelie
[ 49.4 15.8 216.0 4925.0 1 ] Gentoo
[ 42.8 18.5 195.0 4250.0 0 ] Adelie
[ 35.6 17.5 191.0 3175.0 0 ] Adelie
```

Now that we know what the features and labels in the data represent, let's explore the dataset. First, let's see if there are any missing (*null*) values.

[3]

```
# Count the number of null values for each column
penguins.isnull().sum()
```

✓ <1 sec

```
CulmenLength     2
CulmenDepth      2
FlipperLength    2
BodyMass         2
Species          0
dtype: int64
```

It looks like there are some missing feature values, but no missing labels. Let's dig a little deeper and see the rows that contain nulls.

[4]

```
# Show rows containing nulls
penguins[penguins.isnull().any(axis=1)]
```

✓ <1 sec

There are two rows that contain no feature values at all (*NaN* stands for "not a number"), so these won't be useful in training a model. Let's discard them from the dataset.

```
# Drop rows containing NaN values
penguins=penguins.dropna()
#Confirm there are now no nulls
penguins.isnull().sum()
```

[5] ✓ <1 sec

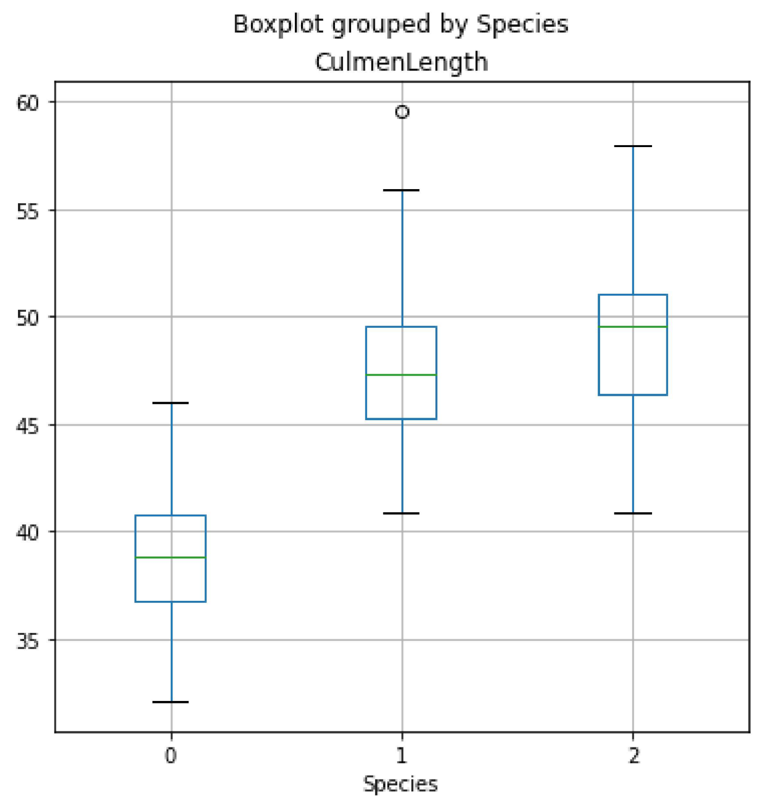
```
CulmenLength     0
CulmenDepth      0
FlipperLength    0
BodyMass         0
Species          0
dtype: int64
```

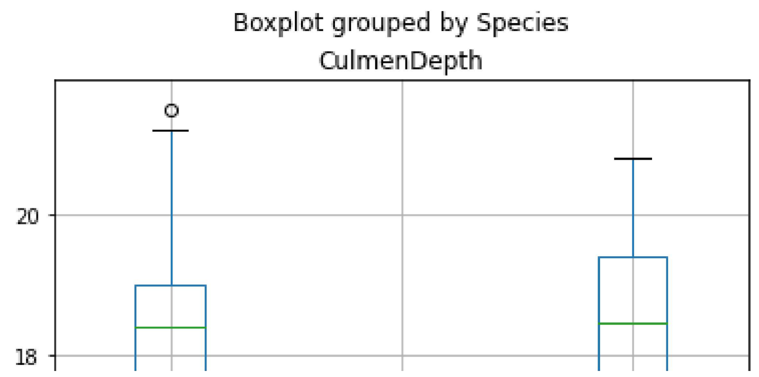
Now that we've dealt with the missing values, let's explore how the features relate to the label by creating some box charts.

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

penguin_features = ['CulmenLength','CulmenDepth','FlipperLength','BodyMass'
penguin_label = 'Species'
for col in penguin_features:
    penguins.boxplot(column=col, by=penguin_label, figsize=(6,6))
    plt.title(col)
plt.show()
```

[6] ✓ 6 sec





From the box plots, it looks like species 0 and 2 (Adelie and Chinstrap) have similar data profiles for culmen depth, flipper length, and body mass, but Chinstraps tend to have longer culmens. Species 1 (Gentoo) tends to have fairly clearly differentiated features from the others; which should help us train a good classification model.

Prepare the data

Just as for binary classification, before training the model, we need to separate the features and label, and then split the data into subsets for training and validation. We'll also apply a *stratification* technique when splitting the data to maintain the proportion of each label value in the training and validation datasets.

```
from sklearn.model_selection import train_test_split

# Separate features and labels
penguins_X, penguins_y = penguins[penguin_features].values, penguins[pengui

# Split data 70%-30% into training set and test set
x_penguin_train, x_penguin_test, y_penguin_train, y_penguin_test = train_te
```

```
print ('Training Set: %d, Test Set: %d \n' % (x_penguin_train.shape[0], x_p
```

✓ 2 sec

[7]

...

```
Training Set: 239, Test Set: 103
```

+ Code + Markdown

Train and evaluate a multiclass classifier

Now that we have a set of training features and corresponding training labels, we can fit a multiclass classification algorithm to the data to create a model. Most scikit-learn classification algorithms inherently support multiclass classification. We'll try a logistic regression algorithm.

```
from sklearn.linear_model import LogisticRegression

# Set regularization rate
reg = 0.1

# train a logistic regression model on the training set
multi_model = LogisticRegression(C=1/reg, solver='lbfgs', multi_class='autc
print (multi_model)
```

✓ 1 sec

[8]

```
LogisticRegression(C=10.0, max_iter=10000)
```

Now we can use the trained model to predict the labels for the test features, and compare the predicted labels to the actual labels:

```
penguin_predictions = multi_model.predict(x_penguin_test)
print('Predicted labels: ', penguin_predictions[:15])
print('Actual labels   : ' ,y_penguin_test[:15])
```

✓ <1 sec

[9]

```
Predicted labels:  [0 1 0 2 2 1 1 1 0 2 2 1 2 1 2]
Actual labels   :  [0 1 2 2 2 1 1 1 0 2 2 1 2 1 2]
```

Let's look at a classification report.

[10]

```
from sklearn. metrics import classification_report

print(classification_report(y_penguin_test, penguin_predictions))
```

✓ <1 sec

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97        45
           1       1.00      1.00      1.00        37
           2       0.95      0.90      0.93        21

    accuracy                           0.97       103
   macro avg       0.97      0.96      0.96       103
weighted avg       0.97      0.97      0.97       103
```

As with binary classification, the report includes *precision* and *recall* metrics for each class. However, while with binary classification we could focus on the scores for the *positive* class; in this case, there are multiple classes so we need to look at an overall metric (either the macro or weighted average) to get a sense of how well the model performs across all three classes.

You can get the overall metrics separately from the report using the scikit-learn metrics score classes, but with multiclass results you must specify which average metric you want to use for precision and recall.

[11]

```
from sklearn.metrics import accuracy_score, precision_score, recall_score

print("Overall Accuracy:",accuracy_score(y_penguin_test, penguin_prediction
print("Overall Precision:",precision_score(y_penguin_test, penguin_predicti
print("Overall Recall:",recall_score(y_penguin_test, penguin_predictions, a
```

✓ <1 sec

```
Overall Accuracy: 0.970873786407767
Overall Precision: 0.9688405797101449
Overall Recall: 0.9608465608465608
```

Now let's look at the confusion matrix for our model:

```
from sklearn.metrics import confusion_matrix
```

```
# Print the confusion matrix
mcm = confusion_matrix(y_penguin_test, penguin_predictions)
print(mcm)
```

[12] ✓ <1 sec

```
[[44  0  1]
 [ 0 37  0]
 [ 2  0 19]]
```

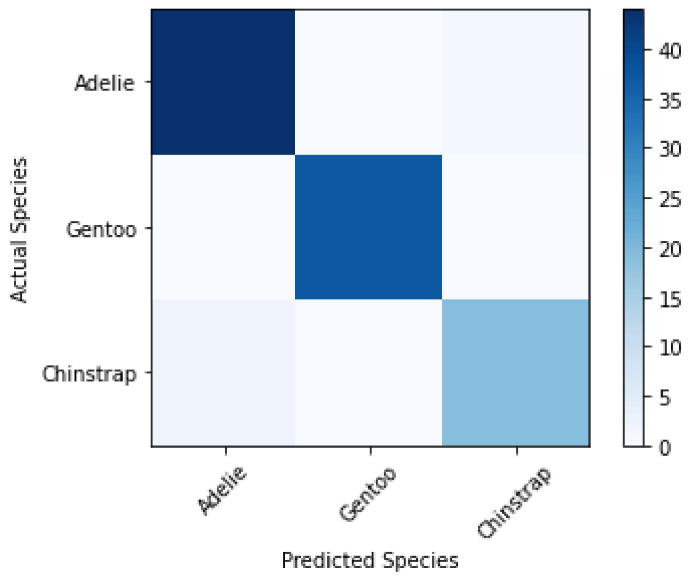
The confusion matrix shows the intersection of predicted and actual label values for each class - in simple terms, the diagonal intersections from top-left to bottom-right indicate the number of correct predictions.

When dealing with multiple classes, it's generally more intuitive to visualize this as a heat map, like this:

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

plt.imshow(mcm, interpolation="nearest", cmap=plt.cm.Blues)
plt.colorbar()
tick_marks = np.arange(len(penguin_classes))
plt.xticks(tick_marks, penguin_classes, rotation=45)
plt.yticks(tick_marks, penguin_classes)
plt.xlabel("Predicted Species")
plt.ylabel("Actual Species")
plt.show()
```

[13] ✓ 1 sec



The darker squares in the confusion matrix plot indicate high numbers of cases, and you can hopefully see a diagonal line of darker squares indicating cases where the predicted and actual label are the same.

In the case of a multiclass classification model, a single ROC curve showing true positive rate vs false positive rate is not possible. However, you can use the rates for each class in a One vs Rest (OVR) comparison to create a ROC chart for each class.

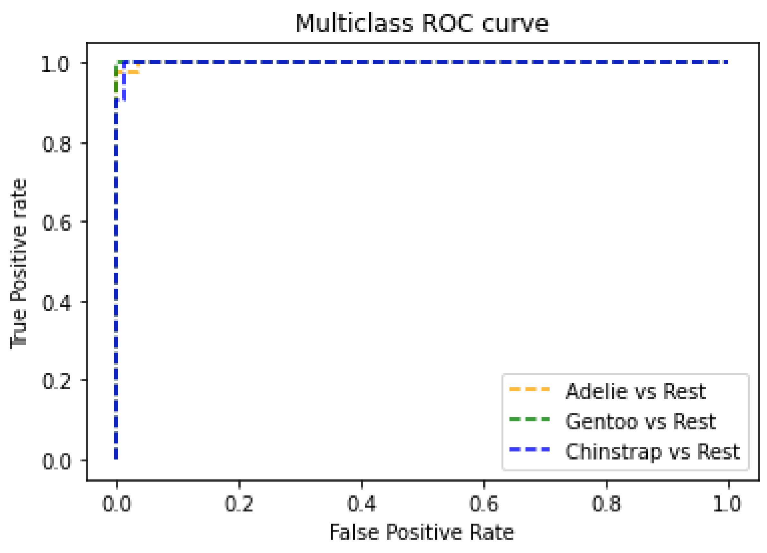
```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

# Get class probability scores
penguin_prob = multi_model.predict_proba(x_penguin_test)

# Get ROC metrics for each class
fpr = {}
tpr = {}
thresh ={}
for i in range(len(penguin_classes)):
    fpr[i], tpr[i], thresh[i] = roc_curve(y_penguin_test, penguin_prob[:,i]

# Plot the ROC chart
plt.plot(fpr[0], tpr[0], linestyle='--',color='orange', label=penguin_class
plt.plot(fpr[1], tpr[1], linestyle='--',color='green', label=penguin_classe
plt.plot(fpr[2], tpr[2], linestyle='--',color='blue', label=penguin_classes
plt.title('Multiclass ROC curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive rate')
plt.legend(loc='best')
plt.show()
```

[14] ✓ 1 sec



To quantify the ROC performance, you can calculate an aggregate area under the curve score that is averaged across all of the OVR curves.

```
auc = roc_auc_score(y_penguin_test,penguin_prob, multi_class='ovr')
```

```
print('Average AUC:', auc)
```

[15] ✓ <1 sec

```
Average AUC: 0.9993574254297553
```

Preprocess data in a pipeline

Again, just like with binary classification, you can use a pipeline to apply preprocessing steps to the data before fitting it to an algorithm to train a model. Let's see if we can improve the penguin predictor by scaling the numeric features in a transformation steps before training. We'll also try a different algorithm (a support vector machine), just to show that we can!

```
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC

# Define preprocessing for numeric columns (scale them)
feature_columns = [0,1,2,3]
feature_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())
    ])

# Create preprocessing steps
preprocessor = ColumnTransformer(
    transformers=[
        ('preprocess', feature_transformer, feature_columns)])

# Create training pipeline
pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                           ('regressor', SVC(probability=True))])


# fit the pipeline to train a linear regression model on the training set
multi_model = pipeline.fit(x_penguin_train, y_penguin_train)
print (multi_model)
```

[16] ✓ <1 sec

```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('preprocess',
                                                  Pipeline(steps=[('scaler',

StandardScaler())]),
                                                  [0, 1, 2, 3])])),
                ('regressor', SVC(probability=True))])
```

Now we can evaluate the new model.

```
# Get predictions from test data
penguin_predictions = multi_model.predict(x_penguin_test)
penguin_prob = multi_model.predict_proba(x_penguin_test)

# Overall metrics
print("Overall Accuracy:",accuracy_score(y_penguin_test, penguin_prediction
print("Overall Precision:",precision_score(y_penguin_test, penguin_predicti
print("Overall Recall:",recall_score(y_penguin_test, penguin_predictions, a
print('Average AUC:', roc_auc_score(y_penguin_test,penguin_prob, multi_clas

# Confusion matrix
plt.imshow(mcm, interpolation="nearest", cmap=plt.cm.Blues)
plt.colorbar()
tick_marks = np.arange(len(penguin_classes))
plt.xticks(tick_marks, penguin_classes, rotation=45)
plt.yticks(tick_marks, penguin_classes)
plt.xlabel("Predicted Species")
plt.ylabel("Actual Species")
plt.show()
```

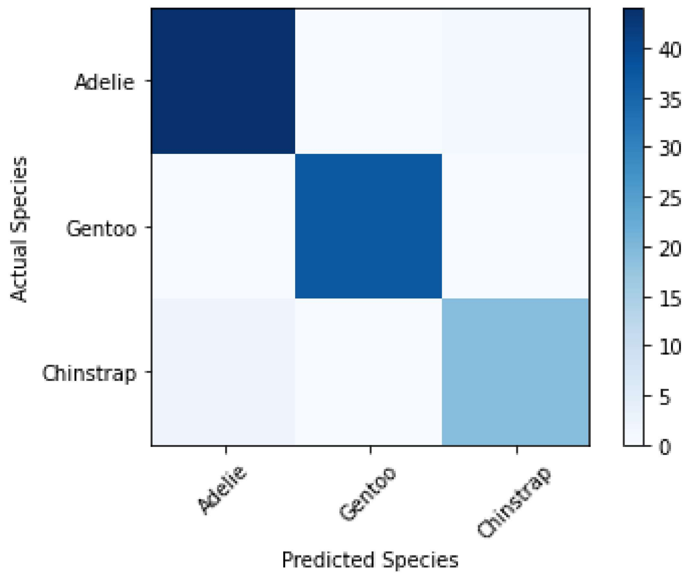
[17] ✓ 1 sec

```
Overall Accuracy: 0.9805825242718447

Overall Precision: 0.9767195767195768

Overall Recall: 0.9767195767195768

Average AUC: 0.9990361381446328
```



Use the model with new data observations

Now let's save our trained model so we can use it again later.

```
import joblib
```

```
# Save the model as a pickle file
filename = './penguin_model.pkl'
joblib.dump(multi_model, filename)
```

[18] ✓ <1 sec

```
['./penguin_model.pkl']
```

OK, so now we have a trained model. Let's use it to predict the class of a new penguin observation:

```
# Load the model from the file
multi_model = joblib.load(filename)

# The model accepts an array of feature arrays (so you can predict the clas
# We'll create an array with a single array of features, representing one p
x_new = np.array([[50.4,15.3,224,5550]])
print ('New sample: {}'.format(x_new[0]))

# The model returns an array of predictions - one for each set of features
# In our case, we only submitted one penguin, so our prediction is the firs
penguin_pred = multi_model.predict(x_new)[0]
print('Predicted class is', penguin_classes[penguin_pred])
```

[19] ✓ <1 sec

```
New sample: [  50.4   15.3  224.  5550. ]
Predicted class is Gentoo
```

You can also submit a batch of penguin observations to the model, and get back a prediction for each one.

```
# This time our input is an array of two feature arrays
x_new = np.array([[49.5,18.4,195, 3600],
         [38.2,20.1,190,3900]])
print ('New samples:\n{}'.format(x_new))

# Call the web service, passing the input data
predictions = multi_model.predict(x_new)

# Get the predicted classes.
for prediction in predictions:
    print(prediction, '(' + penguin_classes[prediction] +')')
```

[20] ✓ <1 sec

```
New samples:
[[  49.5   18.4  195.  3600. ]
 [  38.2   20.1  190.  3900. ]]
```

```
2 (Chinstrap)
0 (Adelie)
```

Summary

Classification is one of the most common forms of machine learning, and by following the basic principles we've discussed in this notebook you should be able to train and evaluate classification models with scikit-learn. It's worth spending some time investigating classification algorithms in more depth, and a good starting point is the Scikit-Learn documentation .

[ ] Press shift + enter to run

Next unit: Knowledge check

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆