# Multiclass Classfication

September 26, 2021



[ ]:
```
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

## 1 Multiclass Classification

We previously created a binary classification model that determined if a piece of fruit was an orange or a grapefruit. There are many problems where binary classification can provide impactful solutions: spam or not spam in an email classifier, hit or hold in a blackjack simulation, buy or not in a stock market analysis. The list is basically endless.

There are other cases, however, where we want to make a decision across three or more classes. This is multiclass classification.

For many applications, multiclass classification can be broken down into many binary classification problems. These models employ a one-vs-all or one-vs-one strategy to create many binary classification tasks that are then aggregated into a multiclass classification model. Neural networks, decision trees, and k-nearest neighbors models are all capable of performing multiclass classification directly.

### 1.1 The Dataset

For this unit we are going to use a classic machine learning dataset, the Iris flower dataset. This is a dataset that was used in 1936 by British biologist and statistician Ronald Fisher to classify iris flowers into one of three species based on four measurements:

- The length of the petals
- The width of the petals
- The length of the sepals (the green petal-looking bits that are found at the base of the petals)
- The width of the sepals

Conveniently, the iris dataset is built into the scikit-learn library, so it is readily available to us. Let's take a look:

[ ]:
```
from sklearn import datasets

iris_bunch = datasets.load_iris()
iris_bunch.keys()
```

[ ]:
```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
'filename', 'data_module'])
```

Scikit-learn datasets are usually delivered in the form of a dictionary-like object called a `Bunch`. This `Bunch` contains the following fields:

- *DESCR*: A string describing the dataset.
- *data*: An array containing the features we are using for classifying. In this case, it's the four measurements listed above for each of 150 plants.
- *feature_names*: Labels for the data.
- *filename*: the file that this data came from.
- *target*: the values that we are trying to classify these flowers into. In this case, since we are dealing with three species of iris, we use three numbers (0, 1 and 2) to identify each species.
- *target_names*: labels for the target values. In this case, 0 refers to the setosa species, 1 to the versicolor species, and 2 to the virginica species.

We'll create a list of columns that we'll use for our model.

[ ]:
```
FEATURES = iris_bunch['feature_names']
TARGET = 'species'

FEATURES, TARGET
```

[ ]:
```
(['sepal length (cm)',
  'sepal width (cm)',
  'petal length (cm)',
  'petal width (cm)'],
 'species')
```

Next we will load the feature and target data into a Pandas dataframe.

[ ]:
```
import pandas as pd

iris_df = pd.DataFrame(iris_bunch['data'], columns=FEATURES)
iris_df[TARGET] = iris_bunch['target']

iris_df.sample(10)
```

[ ]:

| | sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) | \ |
|---|---|---|---|---|---|
| 129 | 7.2 | 3.0 | 5.8 | 1.6 | |
| 138 | 6.0 | 3.0 | 4.8 | 1.8 | |
| 68 | 6.2 | 2.2 | 4.5 | 1.5 | |

```
97                6.2               2.9                4.3               1.3
70                5.9               3.2                4.8               1.8
85                6.0               3.4                4.5               1.6
83                6.0               2.7                5.1               1.6
17                5.1               3.5                1.4               0.3
10                5.4               3.7                1.5               0.2
37                4.9               3.6                1.4               0.1

     species
129        2
138        2
68         1
97         1
70         1
85         1
83         1
17         0
10         0
37         0
```

Let's take a look at a description of the data.

[ ]:
```
iris_df.describe()
```

[ ]:
```
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000
mean            5.843333          3.057333           3.758000
std             0.828066          0.435866           1.765298
min             4.300000          2.000000           1.000000
25%             5.100000          2.800000           1.600000
50%             5.800000          3.000000           4.350000
75%             6.400000          3.300000           5.100000
max             7.900000          4.400000           6.900000

       petal width (cm)     species
count        150.000000  150.000000
mean           1.199333    1.000000
std            0.762238    0.819232
min            0.100000    0.000000
25%            0.300000    0.000000
50%            1.300000    1.000000
75%            1.800000    2.000000
max            2.500000    2.000000
```

There are 150 data points. No columns seem to be missing data and no values seem to be too far out of expected ranges. For example, there are no zero or negative lengths or widths, and the length and width values fall well within what we'd expect for a tulip.

We are interested in using the measurement features to predict the species of an iris. Let's take a

closer look at the values we'll be predicting.

In this case we'll group by our 'species' feature and get a count of each species in our dataset.

[ ]:
```
iris_df.groupby(TARGET).agg('count')
```

[ ]:
```
         sepal length (cm)  sepal width (cm)  petal length (cm)  \
species
0                       50                50                 50
1                       50                50                 50
2                       50                50                 50

         petal width (cm)
species
0                      50
1                      50
2                      50
```

We have 50 examples of each species of iris, and overall we have only 150 samples. This presents two challenges. First, we don't have much data to actually build a model from. Second, the data that we do have is evenly distributed over class types. We might want to make sure that we train over the same distribution.

Luckily, there are solutions to both of these issues!

When we have data that has some weighted distribution across classes, we can do a **stratified split** to ensure that every class appears proportionally in our training data.

When we don't have enough data to properly train a model and don't feel that we can pull training data away, we can do a **k-fold cross validation** in order to utilize all of our data for training, while still trying to minimize model overfitting.

### 1.2 Stratified Split

Let's first split off a set of data to use for our final model testing. We can use scikit-learn's `train_test_split` function to do this.

Since we have so little data, we'll only hold out 10% of the data for the final test.

After we make the split, we can see how many data points we will train off of for each class.

[ ]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    iris_df[FEATURES],
    iris_df[TARGET],
    test_size=0.1,
    random_state=45)

y_train.groupby(y_train).count()
```

[ ]:
```
species
0    43
1    50
2    42
Name: species, dtype: int64
```

Yikes! We kept 50 data points for training class 1. That means we left none for final testing:

[ ]:
```
y_test.groupby(y_test).count()
```

[ ]:
```
species
0    7
2    8
Name: species, dtype: int64
```

### 1.2.1 Exercise 1: Stratified Train Test Split

We risk not holding out a data point for every class if we don't stratify our train test split. Rewrite the split above to create a stratified split. (If you don't remember how, try looking at the documentation for `train_test_split` and finding the argument that can be used to stratify the data.) When you are done, there should be 45 data points for each class in the training data and five data points for each class in the testing data. Print the counts to verify.

**Student Solution**

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(
    iris_df[FEATURES],
    iris_df[TARGET],
    test_size=0.1,
    random_state=45,
    stratify = iris_df[TARGET])

y_train.groupby(y_train).count()
```

[ ]:
```
species
0    45
1    45
2    45
Name: species, dtype: int64
```

---

## 1.3 Cross-Validation

Another problem we have is that we have very little data to work with. We only had 150 data points in total and are only going to train using 135 of those data points. If we are going to be hyperparameter tuning, we'll need a test and validation holdout, which will leave us very little data to train on.

One way to get around this is to use cross-validation. Cross-validation splits the data into a fixed number of tranches and trains on `n-1` of the tranches. Then it calculates a score using the holdout

tranche. It does this repeatedly, holding out one tranche of data for each training pass. By looking at the mean of the scores for each training pass, you can get an idea of how well your model performs without having to specify a test dataset.

The `cross_val_score` method is used to perform the cross-validation. In the example below, we divide the data into five tranches and get five scores.

Since we are cross-validating with a classifier, scikit-learn automatically performs stratified splits for us.

[ ]:
```
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import cross_val_score

estimator = SGDClassifier()

scores = cross_val_score(
    estimator,
    X_train[FEATURES],
    y_train,
    cv=5
)

scores
```

[ ]:
```
array([1.        , 0.74074074, 0.81481481, 0.88888889, 0.7037037 ])
```

We can now find the mean score.

[ ]:
```
scores.mean()
```

[ ]:
```
0.8296296296296296
```

What does this score represent, though? It turns out that it uses the default scoring method for the classifier that we used. In this case we used the `SGDClassifier`, which reports accuracy by default.

Also note that the estimator isn't trained after running cross-validation. You can run cross-validation to test different data preprocessing pipelines and hyperparameters. Once you are happy with a specific setup, you'll need to train the model with the chosen pipeline and parameters.

### 1.3.1 Exercise 2: F1 Scoring

What if we wanted to use F1 for our scoring metric instead of accuracy?

Run `cross_val_score` on an `SGDClassifier` and get the F1 score. Check out the documentation for `cross_val_score`, `make_scorer`, and `f1_score` for clues.

**Student Solution**

[ ]:
```
from sklearn.metrics import f1_score, make_scorer

estimator = SGDClassifier()
```

```
scores = cross_val_score(
    estimator,
    X_train[FEATURES],
    y_train,
    cv=5,
    scoring = make_scorer(f1_score, average ='weighted')
)

scores
```

[ ]:
```
array([0.55555556, 0.85311943, 0.67965368, 0.5280112 , 0.79364495])
```

---

## 2 The Model Pipeline

Since we are now using cross-validation to train the model, we can use our testing holdout data as a final validation. Let's make that clear by renaming the data.

[ ]:
```
X_validation = X_test
y_validation = y_test
```

Now we can work on tuning the model and the model pipeline.

Let's first look back at the data going into the model:

[ ]:
```
iris_df[FEATURES].describe()
```

[ ]:

| | sepal length (cm) | sepal width (cm) | petal length (cm) | \ |
|---|---|---|---|---|
| count | 150.000000 | 150.000000 | 150.000000 | |
| mean | 5.843333 | 3.057333 | 3.758000 | |
| std | 0.828066 | 0.435866 | 1.765298 | |
| min | 4.300000 | 2.000000 | 1.000000 | |
| 25% | 5.100000 | 2.800000 | 1.600000 | |
| 50% | 5.800000 | 3.000000 | 4.350000 | |
| 75% | 6.400000 | 3.300000 | 5.100000 | |
| max | 7.900000 | 4.400000 | 6.900000 | |

| | petal width (cm) |
|---|---|
| count | 150.000000 |
| mean | 1.199333 |
| std | 0.762238 |
| min | 0.100000 |
| 25% | 0.300000 |
| 50% | 1.300000 |
| 75% | 1.800000 |
| max | 2.500000 |

The data is all in the same order of magnitude, but columns like 'sepal length (cm)' are considerably larger than columns like 'petal width (cm)'.

We need to perform some preprocessing to get the data into a more uniform shape before feeding it to the model. To do that we'll use the `StandardScaler`, which removes the mean and subtracts the unit variance from each column of data.

To use the `StandardScaler`, we create the object, `fit()` the data, and then `transform()`.

[ ]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaler.fit(iris_df[FEATURES])

pd.DataFrame(
    scaler.transform(iris_df[FEATURES]),
    columns=FEATURES
).describe()
```

[ ]:
```
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count       1.500000e+02      1.500000e+02       1.500000e+02
mean       -1.690315e-15     -1.842970e-15      -1.698641e-15
std         1.003350e+00      1.003350e+00       1.003350e+00
min        -1.870024e+00     -2.433947e+00      -1.567576e+00
25%        -9.006812e-01     -5.923730e-01      -1.226552e+00
50%        -5.250608e-02     -1.319795e-01       3.364776e-01
75%         6.745011e-01      5.586108e-01       7.627583e-01
max         2.492019e+00      3.090775e+00       1.785832e+00

       petal width (cm)
count      1.500000e+02
mean      -1.409243e-15
std        1.003350e+00
min       -1.447076e+00
25%       -1.183812e+00
50%        1.325097e-01
75%        7.906707e-01
max        1.712096e+00
```

You can see in the output of `describe()` that the data now all has a standard deviation that approaches one.

We need to perform this preprocessing to features before training the model and before getting predictions. It can be error-prone to try to remember to do this. To make the task easier, we can create an estimator `Pipeline` that applies our transformations and calls our estimator.

[ ]:
```
from sklearn.pipeline import Pipeline

estimator = Pipeline(
```

```
  steps=[
    ['scale', StandardScaler()],
    ['classifier', SGDClassifier()],
  ]
)

scores = cross_val_score(
    estimator,
    X_train[FEATURES],
    y_train,
    cv=5,
)
scores.mean()
```

[ ]:
```
0.9185185185185185
```

Scaling gave us a considerable jump in accuracy score. Hopefully you see similar results.

### 2.0.1 Exercise 3: Final Validation

Our accuracy results were pretty good, so we aren't going to do any more hyperparameter tuning in this lab. Before we declare victory, though, we should find the F1 score of our validation data. Using our estimator pipeline, calculate the F1 score for `X_validation`.

[ ]:
```
estimator = Pipeline(
  steps=[
    ['scale', StandardScaler()],
    ['classifier', SGDClassifier()],
  ]
)

scores = cross_val_score(
    estimator,
    X_validation[FEATURES],
    y_validation,
    cv=5,
    scoring = make_scorer(f1_score, average ='weighted')
)

scores.mean()
```

[ ]:
```
0.8222222222222222
```

---

## 3 Exercise 4: Winemaker Identification

Scikit-learn comes prepackaged with many toy datasets. These can be found in the `sklearn.datasets` package. In this exercise we'll be working with the wine dataset.

The dataset contains information about the properties of wines produced by three different producers. The grapes that the producers used all come from the same region.

The columns are:

- alcohol
- malic_acid
- ash
- alcalinity_of_ash
- magnesium
- total_phenols
- flavanoids
- nonflavanoid_phenols
- proanthocyanins
- color_intensity
- hue
- od280/od315_of_diluted_wines
- proline

The target column is a 0, 1, or 2. Each number represents a different producer.

Your task in this exercise is to create a classifier that can identify the producer based on the wine properties.

Use as many code blocks as necessary to examine the data and build and validate your model. Document your process using text blocks and/or comments in your code.

**Student Solution**

**3.0.1**

[ ]:
```
wine_bunch = datasets.load_wine()

FEATURES_2 = wine_bunch['feature_names']
TARGET_2 = 'producers'

FEATURES_2, TARGET_2


wine_df = pd.DataFrame(wine_bunch['data'], columns=FEATURES_2)
wine_df[TARGET_2] =wine_bunch['target']

wine_df.sample(10)
```

[ ]:

| | alcohol | malic_acid | ash | alcalinity_of_ash | magnesium | total_phenols | \ |
|---|---|---|---|---|---|---|---|
| 22 | 13.71 | 1.86 | 2.36 | 16.6 | 101.0 | 2.61 | |
| 26 | 13.39 | 1.77 | 2.62 | 16.1 | 93.0 | 2.85 | |
| 57 | 13.29 | 1.97 | 2.68 | 16.8 | 102.0 | 3.00 | |
| 37 | 13.05 | 1.65 | 2.55 | 18.0 | 98.0 | 2.45 | |
| 49 | 13.94 | 1.73 | 2.27 | 17.4 | 108.0 | 2.88 | |
| 13 | 14.75 | 1.73 | 2.39 | 11.4 | 91.0 | 3.10 | |

```
169    13.40        4.60  2.86               25.0      112.0           1.98
7      14.06        2.15  2.61               17.6      121.0           2.60
103    11.82        1.72  1.88               19.5       86.0           2.50
39     14.22        3.99  2.51               13.2      128.0           3.00

     flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue  \
22         2.88                  0.27             1.69             3.80  1.11
26         2.94                  0.34             1.45             4.80  0.92
57         3.23                  0.31             1.66             6.00  1.07
37         2.43                  0.29             1.44             4.25  1.12
49         3.54                  0.32             2.08             8.90  1.12
13         3.69                  0.43             2.81             5.40  1.25
169        0.96                  0.27             1.11             8.50  0.67
7          2.51                  0.31             1.25             5.05  1.06
103        1.64                  0.37             1.42             2.06  0.94
39         3.04                  0.20             2.08             5.10  0.89

     od280/od315_of_diluted_wines  proline  producers
22                           4.00   1035.0          0
26                           3.22   1195.0          0
57                           2.84   1270.0          0
37                           2.51   1105.0          0
49                           3.10   1260.0          0
13                           2.73   1150.0          0
169                          1.92    630.0          2
7                            3.58   1295.0          0
103                          2.44    415.0          1
39                           3.53    760.0          0
```

---

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(
    wine_df[FEATURES_2],
    wine_df[TARGET_2],
    test_size=0.1,
    random_state=45,
    stratify = wine_df[TARGET_2])

y_train.groupby(y_train).count()
```

[ ]:
```
producers
0    53
1    64
2    43
Name: producers, dtype: int64
```

[ ]:
```
scaler = StandardScaler()

scaler.fit(wine_df[FEATURES_2])

pd.DataFrame(
    scaler.transform(wine_df[FEATURES_2]),
    columns=FEATURES_2
).describe()
```

[ ]:

| | alcohol | malic_acid | ash | alcalinity_of_ash |
|---|---|---|---|---|
| count | 1.780000e+02 | 1.780000e+02 | 1.780000e+02 | 1.780000e+02 |
| mean | 7.841418e-15 | 2.444986e-16 | -4.059175e-15 | -7.110417e-17 |
| std | 1.002821e+00 | 1.002821e+00 | 1.002821e+00 | 1.002821e+00 |
| min | -2.434235e+00 | -1.432983e+00 | -3.679162e+00 | -2.671018e+00 |
| 25% | -7.882448e-01 | -6.587486e-01 | -5.721225e-01 | -6.891372e-01 |
| 50% | 6.099988e-02 | -4.231120e-01 | -2.382132e-02 | 1.518295e-03 |
| 75% | 8.361286e-01 | 6.697929e-01 | 6.981085e-01 | 6.020883e-01 |
| max | 2.259772e+00 | 3.109192e+00 | 3.156325e+00 | 3.154511e+00 |

| | magnesium | total_phenols | flavanoids | nonflavanoid_phenols |
|---|---|---|---|---|
| count | 1.780000e+02 | 1.780000e+02 | 1.780000e+02 | 1.780000e+02 |
| mean | -2.494883e-17 | -1.955365e-16 | 9.443133e-16 | -4.178929e-16 |
| std | 1.002821e+00 | 1.002821e+00 | 1.002821e+00 | 1.002821e+00 |
| min | -2.088255e+00 | -2.107246e+00 | -1.695971e+00 | -1.868234e+00 |
| 25% | -8.244151e-01 | -8.854682e-01 | -8.275393e-01 | -7.401412e-01 |
| 50% | -1.222817e-01 | 9.595986e-02 | 1.061497e-01 | -1.760948e-01 |
| 75% | 5.096384e-01 | 8.089974e-01 | 8.490851e-01 | 6.095413e-01 |
| max | 4.371372e+00 | 2.539515e+00 | 3.062832e+00 | 2.402403e+00 |

| | proanthocyanins | color_intensity | hue |
|---|---|---|---|
| count | 1.780000e+02 | 1.780000e+02 | 1.780000e+02 |
| mean | -1.540590e-15 | -4.129032e-16 | 1.398382e-15 |
| std | 1.002821e+00 | 1.002821e+00 | 1.002821e+00 |
| min | -2.069034e+00 | -1.634288e+00 | -2.094732e+00 |
| 25% | -5.972835e-01 | -7.951025e-01 | -7.675624e-01 |
| 50% | -6.289785e-02 | -1.592246e-01 | 3.312687e-02 |
| 75% | 6.291754e-01 | 4.939560e-01 | 7.131644e-01 |
| max | 3.485073e+00 | 3.435432e+00 | 3.301694e+00 |

| | od280/od315_of_diluted_wines | proline |
|---|---|---|
| count | 1.780000e+02 | 1.780000e+02 |
| mean | 2.126888e-15 | -6.985673e-17 |
| std | 1.002821e+00 | 1.002821e+00 |
| min | -1.895054e+00 | -1.493188e+00 |
| 25% | -9.522483e-01 | -7.846378e-01 |
| 50% | 2.377348e-01 | -2.337204e-01 |
| 75% | 7.885875e-01 | 7.582494e-01 |

```
max                    1.960915e+00   2.971473e+00
```

[ ]:
```
X_validation = X_test
y_validation = y_test
```

[ ]:
```
estimator = Pipeline(
  steps=[
    ['scale', StandardScaler()],
    ['classifier', SGDClassifier()],
  ]
)

scores = cross_val_score(
    estimator,
    X_validation[FEATURES_2],
    y_validation,
    cv=5,
    scoring = make_scorer(f1_score, average ='weighted')
)

scores.mean()
```

[ ]:
```
0.8411111111111111
```

[ ]:
```
estimator = Pipeline(
  steps=[
    ['scale', StandardScaler()],
    ['classifier', SGDClassifier()],
  ]
)

scores = cross_val_score(
    estimator,
    X_validation[FEATURES_2],
    y_validation,
    cv=5,
    scoring = make_scorer(f1_score, average ='weighted')
)

scores.mean()
```

[ ]:
```
0.8111111111111111
```

[ ]:
```
from sklearn.svm import SVC
model = SVC(decision_function_shape='ovo')
model.fit(wine_df[FEATURES_2],wine_df[TARGET_2])
```

[ ]:
```
SVC(decision_function_shape='ovo')
```

[ ]:
```

```