

# AI4M\_C3\_M1\_lecture\_nb\_sklearn

September 10, 2021

## 0.1 AI for Medicine Course 3 Week 1 lecture notebook - Model Training/Tuning Basics with Sklearn

Welcome to this exercise! You're going to be exploring the sklearn library, including an overview of its underlying data types and methods for tweaking a model's hyperparameters. You'll be using the same data from the previous lecture notebook. Let's get started!

### 0.1.1 Packages

First import all the packages that you need for this assignment.

- pandas is what you'll use to manipulate your data
- numpy is a library for mathematical and scientific operations
- sklearn has many efficient tools for machine learning and statistical modeling
- itertools helps with hyperparameter (grid) searching

### 0.1.2 Import Packages

Run the next cell to import all the necessary packages.

```
In [1]: # Import packages
import pandas as pd
import numpy as np
import itertools

# Set the random seed for consistent output
np.random.seed(18)

# Read in the data
data = pd.read_csv("dummy_data.csv", index_col=0)
```

### 0.1.3 Train/Test Split

```
In [2]: # Import module to split data
from sklearn.model_selection import train_test_split

# Get the label
y = data.outcome
```

```

# Get the features
X = data.drop('outcome', axis=1)

# Get training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
print(f"Number of observations for training: {y_train.size}")
print(f"Number of observations for testing: {y_test.size}")

```

Number of observations for training: 37

Number of observations for testing: 13

### 0.1.4 Model Fit and Prediction

Let's fit a logistic regression to the training data. Sklearn allows you to provide arguments that override the defaults.

The default solver is lbfgs.

- Lbfgs stands for '[Limited Memory BFGS](#)', and is an efficient and popular method for fitting models.
- The solver is set explicitly here for learning purposes; if you do not set the solver parameter explicitly, the [LogisticRegression](#) function will use its default solver, which is lbfgs as well.

```
In [3]: from sklearn.linear_model import LogisticRegression
```

```

lr = LogisticRegression(solver='lbfgs')
lr.fit(X_train, y_train)

```

```
Out[3]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
    penalty='l2', random_state=None, solver='lbfgs', tol=0.0001,
    verbose=0, warm_start=False)
```

```
In [4]: lr.get_params()
```

```
Out[4]: {'C': 1.0,
    'class_weight': None,
    'dual': False,
    'fit_intercept': True,
    'intercept_scaling': 1,
    'max_iter': 100,
    'multi_class': 'ovr',
    'n_jobs': 1,
    'penalty': 'l2',
    'random_state': None,
    'solver': 'lbfgs',
    'tol': 0.0001,
    'verbose': 0,
    'warm_start': False}
```

When it fits the training data, sklearn also prints out the model's hyperparameters. - Here, these are the default hyperparameters for sklearn's logistic regression classifier. - Another way to check these parameters is the `get_params()` method of the classifier.

You should spend some time checking out the [documentation](#) to get a deeper understanding of what's going on. One important thing to note is that each classifier has different hyperparameters.

### 0.1.5 Prediction

To predict with the classifier, use the `predict()` method. - This returns a numpy array containing the predicted class for each observation in the test set, as you can see by running the next cell:

```
In [5]: # Use the trained model to predict labels from the features of the test set
        predictions = lr.predict(X_test)

        # View the prediction type, shape, and print out a sample prediction
        print(f"predictions is of type: {type(predictions)}")
        print(f"predictions has shape: {predictions.shape}")
        print(f"predicted class for 10th element in test set: {predictions[9]}")
```

```
predictions is of type: <class 'numpy.ndarray'>
predictions has shape: (13,)
predicted class for 10th element in test set: 0
```

### 0.1.6 Prediction probability

When a model predicts that a label is 1 rather than 0, it may help you to know if the model was predicting 1 with a 51% probability or 90% probability; in other words, how confident is that prediction?

You can get the model's probability of predicting each of the class. - To do this, use the `predict_proba()` method. - The resulting array will have a shape that matches the number of classes for the target variable.

```
In [8]: prediction_probs = lr.predict_proba(X_test)
        print(f"prediction_probs is of type: {type(prediction_probs)}")
        print(f"prediction_probs has shape: {prediction_probs.shape}")
        print(f"probabilities for first element in test set: {prediction_probs[0]}")
```

```
prediction_probs is of type: <class 'numpy.ndarray'>
prediction_probs has shape: (13, 2)
probabilities for first element in test set: [0.42348297 0.57651703]
```

There are 13 patients in the test set. Each patient's label could be either 0 or 1, so the prediction probability has 13 rows and 2 columns. To know which column refers to label 0 and which refers to label 1, you can check the `.classes_` attribute.

```
In [9]: lr.classes_
```

```
Out[9]: array([0, 1])
```

Since the order of the `classes_` array is 0, then 1, column 0 of the prediction probabilities has label 0, and column 1 has label 1.

Let's print these for the first 5 elements of the dataset:

```
In [10]: for i in range(5):
          print(f"Element number: {i}")
          print(f"Predicted class: {predictions[i]}")
          print(f"Probability of predicting class 0: {prediction_probs[i][0]}")
          print(f"Probability of predicting class 1: {prediction_probs[i][1]}\n")
```

```
Element number: 0
Predicted class: 1
Probability of predicting class 0: 0.42348296737845204
Probability of predicting class 1: 0.576517032621548
```

```
Element number: 1
Predicted class: 1
Probability of predicting class 0: 0.4914968703166631
Probability of predicting class 1: 0.5085031296833369
```

```
Element number: 2
Predicted class: 1
Probability of predicting class 0: 0.483088763245346
Probability of predicting class 1: 0.516911236754654
```

```
Element number: 3
Predicted class: 0
Probability of predicting class 0: 0.86953653498578
Probability of predicting class 1: 0.13046346501422001
```

```
Element number: 4
Predicted class: 0
Probability of predicting class 0: 0.8470774295731573
Probability of predicting class 1: 0.15292257042684265
```

You can see here that the predicted class matches the class with a higher probability of being predicted. Since you're dealing with numpy arrays, you can simply slice them and get specific information, such as the probability of predicting class 1 for all elements in the test set:

```
In [11]: # Retrieve prediction probabilities for label 1, for all patients
          prediction_probs[:, 1]
```

```
Out[11]: array([0.57651703, 0.50850313, 0.51691124, 0.13046347, 0.15292257,
                 0.26162479, 0.50831618, 0.3190805 , 0.37250246, 0.47736442,
                 0.15743244, 0.51193665, 0.26832495])
```

### 0.1.7 Tuning the Model

Most of the time, the predictive power of a classifier can be increased if a good set of hyperparameters is defined. This is known as model tuning.

For this process, you'll need a classifier, an appropriate evaluation metric, and a set of parameters to test. Since this is a dummy example, you'll use the default metric for the logistic regression classifier: the **mean accuracy**.

### 0.1.8 Mean Accuracy

Mean Accuracy is the number of correct predictions divided by total predictions. This can be computed with the `score()` method.

Let's begin by checking the performance of your out-of-the-box logit classifier:

```
In [12]: lr.score(X_test, y_test)
```

```
Out[12]: 0.6153846153846154
```

Let's say you want to tweak this model's default parameters. You can pass a dictionary containing the values you specify to the classifier when you instantiate it. Notice that these must be passed as keyword arguments, or *kwargs*, which are created by using the `**` prefix:

```
In [13]: # Choose hyperparameters and place them as key-value pairs in a dictionary
        params = {
            'solver': 'liblinear',
            'fit_intercept': False,
            'penalty': 'l1',
            'max_iter': 500
        }

        # Pass in the dictionary as keyword arguments to the model
        lr_tweaked = LogisticRegression(**params)

        # Train the model
        lr_tweaked.fit(X_train, y_train)

        # View hyper-parameters
        print(f"Tweaked hyperparameters: {lr_tweaked.get_params()}\n")

        # Evaluate the model with the mean accuracy
        print(f"Mean Accuracy: {lr_tweaked.score(X_test, y_test)}")
```

```
Tweaked hyperparameters: {'C': 1.0, 'class_weight': None, 'dual': False, 'fit_intercept': False,
```

```
Mean Accuracy: 0.5384615384615384
```

The model with the tweaked parameters is worse than the original! However, there might still be some combination of parameters that increase the predictive power of your logit classifier.

### 0.1.9 Try different hyperparameters

Testing this can be daunting considering all the possible parameter combinations. Let's try something

To get started, you'll apply `itertools.product()` to create all the combinations of parameters. - Notice that the iterable (in this case a list of the lists of parameters) must be passed as `*args` to the `product()` function.

```
In [14]: # Choose hyperparameters and place in a dictionary
hyperparams = {
    'solver': ["liblinear"],
    'fit_intercept': [True, False],
    'penalty': ["l1", "l2"],
    'class_weight': [None, "balanced"]
}
# Get the values of hyperparams and convert them to a list of lists
hp_values = list(hyperparams.values())
hp_values

Out[14]: [['liblinear'], [True, False], ['l1', 'l2'], [None, 'balanced']]

In [15]: # Get every combination of the hyperparameters
for hp in itertools.product(*hp_values):
    print(hp)

('liblinear', True, 'l1', None)
('liblinear', True, 'l1', 'balanced')
('liblinear', True, 'l2', None)
('liblinear', True, 'l2', 'balanced')
('liblinear', False, 'l1', None)
('liblinear', False, 'l1', 'balanced')
('liblinear', False, 'l2', None)
('liblinear', False, 'l2', 'balanced')

In [16]: # Loop through the combinations of hyperparams
for hp in itertools.product(*hp_values):

    # Create the model with the hyperparams
    estimator = LogisticRegression(solver=hp[0],
                                   fit_intercept=hp[1],
                                   penalty=hp[2],
                                   class_weight=hp[3])

    # Fit the model
    estimator.fit(X_train, y_train)
    print(f"Parameters used: {hp}")
    print(f"Mean accuracy of the model: {estimator.score(X_test, y_test)}\n")

Parameters used: ('liblinear', True, 'l1', None)
Mean accuracy of the model: 0.5384615384615384
```

```
Parameters used: ('liblinear', True, 'l1', 'balanced')
Mean accuracy of the model: 0.46153846153846156
```

```
Parameters used: ('liblinear', True, 'l2', None)
Mean accuracy of the model: 0.38461538461538464
```

```
Parameters used: ('liblinear', True, 'l2', 'balanced')
Mean accuracy of the model: 0.46153846153846156
```

```
Parameters used: ('liblinear', False, 'l1', None)
Mean accuracy of the model: 0.5384615384615384
```

```
Parameters used: ('liblinear', False, 'l1', 'balanced')
Mean accuracy of the model: 0.46153846153846156
```

```
Parameters used: ('liblinear', False, 'l2', None)
Mean accuracy of the model: 0.3076923076923077
```

```
Parameters used: ('liblinear', False, 'l2', 'balanced')
Mean accuracy of the model: 0.46153846153846156
```

Note that in the graded assignment, you will take a more generalizable approach that doesn't require you to explicitly specify each hyperparameter.

That is, instead of:

```
LogisticRegression(solver=hp[0], fit_intercept=hp[1], ...
```

You'll be able to write:

```
LogisticRegression(**params)
```

Looks like none of these models beats the original! This won't always be the case, so next time the opportunity arises, you'll be able to check this for yourself.

### 0.1.10 Grid Search

This is essentially grid search. You'll be implementing customized grid search in the graded assignment.

- Note that even though sci-kit learn provides a grid search function, it uses K-fold cross validation, which you won't want to do in the assignment, which is why you will implement grid search yourself.

### 0.1.11 Congratulations on completing this lecture notebook!

By now, you should feel more comfortable with the sklearn library and how it works. You also created a grid search from scratch by leveraging the itertools library. Nice work!