Bootcamp
Machine Learning

ARTIFICIAL INTELLIGENCE

Bootcamp ML

Day02 - Logistic Regression

Welcome to the day02 of the machine learning bootcamp ! Today you will see how to implement a logistic regression
class using a gradient descent algorithm. We will first go through a bit of maths, then we will implement our
class and to finish we will build our model evaluation functions.

Notions of the day

- Sigmoid
- Log loss
- Gradient descent
- Logistic regression
- Model evaluation
- Confusion matrix

General rules

- The version of Python to use is 3.7.x, you can check the version of Python with the following command: python -V
- The norm: during this bootcamp you will follow the Pep8 standards https://www.python.org/dev/peps/pep-0008/
- The function eval is never allowed.
- The exercises are ordered from the easiest to the hardest.
- Your exercises are going to be evaluated by someone else so make sure that variables and functions names are appropriated.
- Your man is internet.
- You can also ask question in the dedicated channel in Slack: 42-ai.slack.com.
- If you find any issue or mistakes in the subject please create an issue on our dedicated repository on Github: https://github.com/42-AI/bootcamp_machine-learning/issues.

Helper

Ensure that you have the right Python version.

```
> which python
/goinfre/miniconda/bin/python
> python -V
Python 3.7.*
> which pip
/goinfre/miniconda/bin/pip
```

Mathematical delights (continued)

Exercise 00 - Sigmoid

Exercise 01 - Logistic Loss Function

Exercise 02 - Logistic Gradient

Exercise 03 - Vectorized Logistic Loss Function

Exercise 04 - Vectorized Logistic Gradient

Algorithm

Exercise 05 - Logistic Regression

Model Evaluation

Exercise 06 - Accuracy

Exercise 07 - Precision

Exercise 08 - Recall

Exercise 09 - F1 Score

Exercise 10 - Confusion matrix

Exercise 00 - Sigmoid

Turning directory :	ex00
Files to turn in :	sigmoid.py
Forbidden function :	None
Remarks :	n/a

Objectives:

The goal of this first exercise is that you understand what is the sigmoid function and why it is important for logistic regression.

Instructions:

You must implement the following formula as a function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

where:

- x is a scalar or a vector
- f is a function applied to x

This function is called the sigmoid function also known as standard logistic sigmoid function.

It is a special case of the logistic function below, with $L = 1, k = 1$ and $x_0 = 0$:

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

The sigmoid function transform an input into a probability value, i.e. a value between 0 and 1. This probability value will then be used to classify the input.

In the sigmoid.py file create the following function as per the instructions below:

```python
def sigmoid_(x):
    """
    Compute the sigmoid of a scalar or a list.
    Args:
        x: a scalar or list
    Returns:
        The sigmoid value as a scalar or list.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

Nota Bene: if your argument is a list, the function would be applied element-wise to this list and a list of the same shape would be returned.

Examples:

```python
from sigmoid import sigmoid_


x = -4
print(sigmoid_(x))
# 0.01798620996209156
x = 2
print(sigmoid_(x))
# 0.8807970779778823
x = [-4, 2, 0]
print(sigmoid_(x))
# [0.01798620996209156, 0.8807970779778823, 0.5]
```

Exercise 01 - Logistic Loss Function

Turning directory :	ex01
Files to turn in :	log_loss.py
Forbidden library :	Numpy
Remarks :	n/a

Objectives:

The goal of this exercise is that you understand the logistic loss function and why we use it over MSE loss function.

Instructions:

You must implement the following formula as a function:

$$J(\theta) = -\frac{1}{m} * [\sum_{i=1}^{m} y^{(i)} log(h_\theta(x^{(i)})) + (1 - y^{(i)}) log(1 - h_\theta(x^{(i)}))]$$

where:

- $J(\theta)$ is the cost function with theta being the weights.
- m is the length of y, i.e. the number of observations in our sample.
- $h_\theta(x)$ also called y_pred or y_hat, is the calculated hypothesis and it represents the predicted output (formula below).
- y, also called y_true, represents the desired output, either 1 or 0.

This function is called the Cross-Entropy loss or logistic loss.
We encourage you to get a look at this section of the Cross entropy Wikipedia.

Formula for hypothesis:

$$\hat{y} = h_\theta(x) = \frac{1}{1 + e^{-\theta \cdot x}}$$

As you may have noticed, this is our sigmoid function with $\theta \cdot x$ as argument.

In the log_loss.py file create the following function as per the instructions below:

```
def log_loss_(y_true, y_pred, m, eps=1e-15):
    """
    Compute the logistic loss value.
    Args:
        y_true: a scalar or a list for the correct labels
        y_pred: a scalar or a list for the predicted labels
        m: the length of y_true (should also be the length of y_pred)
        eps: epsilon (default=1e-15)
    Returns:
        The logistic loss value as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

Hint: the purpose of epsilon (eps) is to avoid log(0) errors, it is a very small residual value we add to y_pred.

N.B.: the length y_pred and y_true MUST be the same !

Examples:

```python
from sigmoid import sigmoid_
from log_loss import log_loss_


# Test n.1
x = 4
y_true = 1
theta = 0.5
y_pred = sigmoid_(x * theta)
m = 1 # length of y_true is 1
print(log_loss_(y_true, y_pred, m))
# 0.12692801104297152

# Test n.2
x = [1, 2, 3, 4]
y_true = 0
theta = [-1.5, 2.3, 1.4, 0.7]
x_dot_theta = sum([a*b for a, b in zip(x, theta)])
y_pred = sigmoid_(x_dot_theta)
m = 1
print(log_loss_(y_true, y_pred, m))
# 10.100041078687479

# Test n.3
x_new = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
y_true = [1, 0, 1]
theta = [-1.5, 2.3, 1.4, 0.7]
x_dot_theta = []
for i in range(len(x_new)):
    my_sum = 0
    for j in range(len(x_new[i])):
        my_sum += x_new[i][j] * theta[j]
    x_dot_theta.append(my_sum)
y_pred = sigmoid_(x_dot_theta)
m = len(y_true)
print(log_loss_(y_true, y_pred, m))
# 7.233346147374828
```

Exercise 02 - Logistic Gradient

Turning directory :	ex02
Files to turn in :	log_gradient.py
Forbidden library :	Numpy
Remarks :	n/a

Objectives:

The goal of this exercise is that you understand how to calculate the gradient and what it represents.

Instructions:

You must implement the following formula as a function:

$$\vec{\nabla} = \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

where:

- $\vec{\nabla}$ is the gradient
- x are the variables of your models
- m is the length of y, i.e. the number of observations in our sample
- $h_\theta(x)$, also called y_pred, is the calculated hypothesis and it represents the predicted output
- y, also called y_true, represents the desired output, either 1 or 0.

We will use the gradient later to update our weights (θ) with respect to the learning rate (α).

In the log_gradient.py file create the following function as per the instructions below:

```python
def log_gradient_(x, y_true, y_pred):
    """
    Compute the gradient.
    Args:
        x: a list or a matrix (list of lists) for the samples
        y_true: a scalar or a list for the correct labels
        y_pred: a scalar or a list for the predicted labels
    Returns:
        The gradient as a scalar or a list of the width of x.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

- x length (x.shape[0]) should match y_true and y_pred length, i.e. we should have the same number of observations.
- x width (x.shape[1]) will be the number of coefficients or the number of coefficients + 1 if you choose to add an intercept value and it should match theta length, but this is for later when we will update theta in our gradient descent algorithm.

Examples:

```python
from sigmoid import sigmoid_
from log_gradient import log_gradient_


# Test n.1
x = [1, 4.2] # 1 represent the intercept
y_true = 1
theta = [0.5, -0.5]
x_dot_theta = sum([a*b for a, b in zip(x, theta)])
y_pred = sigmoid_(x_dot_theta)
print(log_gradient_(x, y_pred, y_true))
# [0.8320183851339245, 3.494477217562483]

# Test n.2
x = [1, -0.5, 2.3, -1.5, 3.2]
y_true = 0
theta = [0.5, -0.5, 1.2, -1.2, 2.3]
x_dot_theta = sum([a*b for a, b in zip(x, theta)])
y_pred = sigmoid_(x_dot_theta)
print(log_gradient_(x, y_true, y_pred))
# [0.99999685596372, -0.49999842798186, 2.299992768716556,
-1.4999952839455801, 3.1999899390839044]

# Test n.3
x_new = [[1, 2, 3, 4, 5], [1, 6, 7, 8, 9], [1, 10, 11, 12, 13]]
# first column of x_new are intercept values initialized to 1
y_true = [1, 0, 1]
theta = [0.5, -0.5, 1.2, -1.2, 2.3]
x_new_dot_theta = []
for i in range(len(x_new)):
    my_sum = 0
    for j in range(len(x_new[i])):
        my_sum += x_new[i][j] * theta[j]
    x_new_dot_theta.append(my_sum)
y_pred = sigmoid_(x_new_dot_theta)
print(log_gradient_(x_new, y_true, y_pred))
# [0.9999445100449934, 5.999888854245219, 6.999833364290213,
7.999777874335206, 8.999722384380199]
```

Exercise 03 - Vectorized Logistic Loss Function

Turning directory :	ex03
Files to turn in :	vec_log_loss.py
Forbidden function :	None
Remarks :	n/a

Objectives:

Now that you understood how we can calculate the loss, you will see how to do it with vectorized operations.
The goal of this exercise is to produce the same result as in ex01 but this time you will use numpy ndarrays.

You must implement the following formula as a function:

$$J(\theta) = -\frac{1}{m} * [y \cdot log(h) + (1 - y) \cdot log(1 - h)]$$

where:

- $J(\theta)$ is the cost function with theta being the weights.
- m is the length of y, i.e. the number of observations in our sample.
- h also called y_pred or y_hat, is the calculated hypothesis and it represents the vector of predicted outputs (formula below).
- y, also called y_true, represents the vector of desired outputs, either 1 or 0.

Instructions:

In the vec_log_loss.py file create the following function as per the instructions below:

```python
def vec_log_loss_(y_true, y_pred, m, eps=1e-15):
    """
    Compute the logistic loss value.
    Args:
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
        m: the length of y_true (should also be the length of y_pred)
        eps: epsilon (default=1e-15)
    Returns:
        The logistic loss value as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

N.B: you might want to update your sigmoid function to work with numpy ndarrays ;) !

Examples:

```python
import numpy as np
from sigmoid import sigmoid_
from vec_log_loss import vec_log_loss_


# Test n.1
x = 4
y_true = 1
theta = 0.5
y_pred = sigmoid_(x * theta)
m = 1 # length of y_true is 1
print(vec_log_loss_(y_true, y_pred, m))
# 0.12692801104297152

# Test n.2
x = np.array([1, 2, 3, 4])
y_true = 0
theta = np.array([-1.5, 2.3, 1.4, 0.7])
y_pred = sigmoid_(np.dot(x, theta))
m = 1
print(vec_log_loss_(y_true, y_pred, m))
# 10.100041078687479

# Test n.3
x_new = np.arange(1, 13).reshape((3, 4))
y_true = np.array([1, 0, 1])
theta = np.array([-1.5, 2.3, 1.4, 0.7])
y_pred = sigmoid_(np.dot(x_new, theta))
m = len(y_true)
print(vec_log_loss_(y_true, y_pred, m))
# 7.233346147374828
```

Exercise 04 - Vectorized Logistic Gradient

Turning directory :	ex04
Files to turn in :	vec_log_gradient.py
Forbidden function :	None
Remarks :	n/a

Objectives:

Now that you understood how we can calculate the gradient, you will see how to do it with vectorized operations.
The goal of this exercise is to produce the same result as in ex02 but this time you will use numpy ndarrays.
You should only need one line of code to do this (**`return ______`**).

Instructions:

In the vec_log_gradient.py file create the following function as per the instructions below:

```
def vec_log_gradient_(x, y_true, y_pred):
"""
    Compute the gradient.
    Args:
        x: a 1d or 2d numpy ndarray for the samples
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
    Returns:
        The gradient as a scalar or a numpy ndarray of the width of x.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

- x length (x.shape[0] if it is a 2d ndarray) should match y_true and y_pred length, i.e. we should have the same number of observations.
- x width (x.shape[1] if it is a 2d ndarray) will be the number of variables of your model (add 1 if you choose to add an intercept value) and it should match theta's length, but this is for later when we will update theta in our gradient descent algorithm.

Examples:

```python
import numpy as np
from sigmoid import sigmoid_
from vec_log_gradient import vec_log_gradient_


# Test n.1
x = np.array([1, 4.2]) # x[0] represent the intercept
y_true = 1
theta = np.array([0.5, -0.5])
y_pred = sigmoid_(np.dot(x, theta))
print(vec_log_gradient_(x, y_pred, y_true))
# [0.83201839 3.49447722]

# Test n.2
x = np.array([1, -0.5, 2.3, -1.5, 3.2]) # x[0] represent the intercept
y_true = 0
theta = np.array([0.5, -0.5, 1.2, -1.2, 2.3])
y_pred = sigmoid_(np.dot(x, theta))
print(vec_log_gradient_(x, y_true, y_pred))
# [ 0.99999686 -0.49999843 2.29999277 -1.49999528 3.19998994]

# Test n.3
x_new = np.arange(2, 14).reshape((3, 4))
x_new = np.insert(x_new, 0, 1, axis=1)
# first column of x_new are now intercept values initialized to 1
y_true = np.array([1, 0, 1])
theta = np.array([0.5, -0.5, 1.2, -1.2, 2.3])
y_pred = sigmoid_(np.dot(x_new, theta))
print(vec_log_gradient_(x_new, y_true, y_pred))
# [0.99994451 5.99988885 6.99983336 7.99977787 8.99972238]
```

Exercise 05 - Logistic Regression

Turning directory :	ex05
Files to turn in :	log_reg.py
Forbidden function :	None
Remarks :	n/a

Objectives:

Now it is time to use everything you built so far to implement a logistic regression classifier using gradient descent algorithm.
You must have seen the power of numpy for vectorized operations. Well let's make something more concrete with that.

The most curious of you may notice that scikit-learn implementation of the logistic regression offers a lot of options.

The goal of this exercise is to make a simplified but nonetheless useful and powerful version with fewer options.

Instructions:

In the log_reg.py file create the following class with the following methods as per the instructions below:

```python
class LogisticRegressionBatchGd:
    def __init__(self, alpha=0.001, max_iter=1000, verbose=False,
learning_rate='constant'):
        self.alpha = alpha
        self.max_iter = max_iter
        self.verbose = verbose
        self.learning_rate = learning_rate # can be 'constant' or
'invscaling'
        self.thetas = []
        # Your code here (e.g. a list of loss for each epochs...)

    def fit(self, x_train, y_train):
        """
        Fit the model according to the given training data.
        Args:
            x_train: a 1d or 2d numpy ndarray for the samples
            y_train: a scalar or a numpy ndarray for the correct labels
        Returns:
            self : object
            None on any error.
        Raises:
            This method should not raise any Exception.
        """
        # Your code here

    def predict(self, x_train):
        """
        Predict class labels for samples in x_train.
        Arg:
            x_train: a 1d or 2d numpy ndarray for the samples
        Returns:
            y_pred, the predicted class label per sample.
            None on any error.
        Raises:
            This method should not raise any Exception.
        """
        # Your code here

    def score(self, x_train, y_train):
        """
        Returns the mean accuracy on the given test data and labels.
        Arg:
            x_train: a 1d or 2d numpy ndarray for the samples
            y_train: a scalar or a numpy ndarray for the correct labels
        Returns:
            Mean accuracy of self.predict(x_train) with respect to y_true
            None on any error.
        Raises:
            This method should not raise any Exception.
        """
        # Your code here
```

It would be a good idea to use the functions you created so far in the mathematical part and a good way to do that is to put them in your class as private methods.

Now that you have done this, you will test your model with a train dataset and a test dataset.

Some words about the datasets you will use:

- Resource links are in the resources folder.
- The dataset called 'train_dataset_clean.csv' is a modified version of the Census Income Dataset (https://archive.ics.uci.edu/ml/datasets/census+income). All the categorical variables have been modified to dummy variables, several columns were dropped ('fnlwgt' and 'education'), the numerical variables we already had ('age', 'education-num', 'capital-gain', 'capital-loss' and 'hours-per-week') were standardized by removing the mean and scaling to unit variance (see sklearn.preprocessing.StandardScaler) and missing values have been imputed taking the mode (the most frequent value).
- The dataset called 'test_dataset_clean.csv' is the same as the dataset 'train_dataset_clean.csv' but with fewer and most importantly different rows.

The goal is to give you two datasets on which you can just train and test your own logistic regression class but if you want to dive a bit further on how the preprocessing was done, I advise you to look at this article on TDS.

(While I'm here, TDS is a good website with great articles if you wish to learn more about data science or machine learning. I would strongly advise you to put it on your top list of sites you have to visit everyday if you wish to work on a data related position in the future. PS: private navigation in your browser is the way to go for unlimited reading)

Regarding both datasets:

- The 1st column correspond to the 'income' variable : 0 if '<=50K' and 1 if '>50k'.
- The 2nd to the 6th columns correspond to the standardized numerical variables : 'age', 'education-num' 'capital-gain', 'capital-loss' and 'hours-per-week'.
- The 7th to the 81th columns correspond to dummy variables for the following categorical variables : 'workClass', 'marital-status', 'occupation', 'relationship', 'race', 'sex' and 'native-country'.

So what we try to do here is to predict with the 81 variables (columns 1 to 81) if the income (column 0) of the person (one person is one row in our data) is lower or equal to 50k ('income' == 0) or is greater than 50k ('income' == 1).

Examples:

```python
import pandas as pd
import numpy as np
from log_reg import LogisticRegressionBatchGd


# We load and prepare our train and test dataset into x_train, y_train and
x_test, y_test
df_train = pd.read_csv('train_dataset_clean.csv', delimiter=',',
header=None, index_col=False)
x_train, y_train = np.array(df_train.iloc[:, 1:82]), df_train.iloc[:, 0]
df_test = pd.read_csv('test_dataset_clean.csv', delimiter=',', header=None,
index_col=False)
x_test, y_test = np.array(df_test.iloc[:, 1:82]), df_test.iloc[:, 0]

# We set our model with our hyperparameters : alpha, max_iter, verbose and
learning_rate
model = LogisticRegressionBatchGd(alpha=0.01, max_iter=1500, verbose=True,
learning_rate='constant')

# We fit our model to our dataset and display the score for the train and
test datasets
model.fit(x_train, y_train)
print(f'Score on train dataset : {model.score(x_train, y_train)}')
y_pred = model.predict(x_test)
print(f'Score on test dataset : {(y_pred == y_test).mean()}')

# epoch 0     : loss 2.711028065632692
# epoch 150 : loss 1.760555094793668
# epoch 300 : loss 1.165023422947427
# epoch 450 : loss 0.830808383847448
# epoch 600 : loss 0.652110347325305
# epoch 750 : loss 0.555867078788320
# epoch 900 : loss 0.501596689945403
# epoch 1050 : loss 0.469145216528238
# epoch 1200 : loss 0.448682476966280
# epoch 1350 : loss 0.435197719530431
# epoch 1500 : loss 0.425934034947101
# Score on train dataset : 0.7591904425539756
# Score on test dataset : 0.7637737239727289

# This is an example with verbose set to True, you could choose to display
your loss at the epochs you want.
# Here I choose to only display 11 rows no matter how many epochs I had.
# Your score should be pretty close to mine.
# Your loss may be quite different weither you choose different
hyperparameters, if you add an intercept to your x_train
# or if you shuffle your x_train at each epochs (this introduce
stochasticity !) etc...
# You might not get a score as good as
sklearn.linear_model.LogisticRegression because it uses a different
algorithm and
# more optimized parameters that would require more time to implement.
```

Optional part

To go further here is what you could do:

- Keep your loss(self.loss_list ?) at each epoch and plot it after you fitted the model to see how it decreases.
- Keep your learning-rate (self.alpha_list ?) at each epoch and plot it after you fitted the model to see if it has the good behavior (constant vs invscaling)

Exercise 06 - Accuracy

Turning directory :	ex06
Files to turn in :	accuracy.py
Forbidden function :	None
Remarks :	n/a

Objectives:

The goal of this exercise is to recreate the function accuracy_score of sklearn.metrics and to learn what represents the accuracy and how to measure it.

Instructions:

For the sake of simplicity, we will only ask you to have two parameters.

In the accuracy.py file create the following function as per the instructions below:

```python
def accuracy_score_(y_true, y_pred):
    """
    Compute the accuracy score.
    Args:
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
    Returns:
        The accuracy score as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

Examples:

```
import numpy as np
from accuracy import accuracy_score_
from sklearn.metrics import accuracy_score


# Test n.1
y_pred = np.array([1, 1, 0, 1, 0, 0, 1, 1])
y_true = np.array([1, 0, 0, 1, 0, 1, 0, 0])
print(accuracy_score_(y_true, y_pred))
print(accuracy_score(y_true, y_pred))
# 0.5
# 0.5

# Test n.2
y_pred = np.array(['norminet', 'dog', 'norminet', 'norminet', 'dog', 'dog',
'dog', 'dog'])
y_true = np.array(['dog', 'dog', 'norminet', 'norminet', 'dog', 'norminet',
'dog', 'norminet'])
print(accuracy_score_(y_true, y_pred))
print(accuracy_score(y_true, y_pred))
# 0.625
# 0.625
```

Exercise 07 - Precision

Turning directory :	ex07
Files to turn in :	precision.py
Forbidden function :	None
Remarks :	n/a

Objectives:

The goal of this exercise is to recreate the function precision_score of sklearn.metrics and to learn what represents the precision and how to measure it.

Instructions:

For the sake of simplicity, we will only ask you to have three parameters.

In the precision.py file create the following function as per the instructions below:

```
def precision_score_(y_true, y_pred, pos_label=1):
    """
    Compute the precision score.
    Args:
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
        pos_label: str or int, the class on which to report the
precision_score (default=1)
    Returns:
        The precision score as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

Examples:

```python
import numpy as np
from precision import precision_score_
from sklearn.metrics import precision_score


# Test n.1
y_pred = np.array([1, 1, 0, 1, 0, 0, 1, 1])
y_true = np.array([1, 0, 0, 1, 0, 1, 0, 0])
print(precision_score_(y_true, y_pred))
print(precision_score(y_true, y_pred))
# 0.4
# 0.4

# Test n.2
y_pred = np.array(['norminet', 'dog', 'norminet', 'norminet', 'dog', 'dog',
'dog', 'dog'])
y_true = np.array(['dog', 'dog', 'norminet', 'norminet', 'dog', 'norminet',
'dog', 'norminet'])
print(precision_score_(y_true, y_pred, pos_label='dog'))
print(precision_score(y_true, y_pred, pos_label='dog'))
# 0.6
# 0.6

# Test n.3
print(precision_score_(y_true, y_pred, pos_label='norminet'))
print(precision_score(y_true, y_pred, pos_label='norminet'))
# 0.6666666666666666
# 0.6666666666666666
```

Exercise 08 - Recall

Turning directory :	ex08
Files to turn in :	recall.py
Forbidden function :	None
Remarks :	n/a

Objectives:

The goal of this exercise is to recreate the function recall_score of sklearn.metrics and to learn what represents the recall and how to measure it.

Instructions:

For the sake of simplicity, we will only ask you to have three parameters.

In the recall.py file create the following function as per the instructions below:

```
def recall_score_(y_true, y_pred, pos_label=1):
    """
    Compute the recall score.
    Args:
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
        pos_label: str or int, the class on which to report the
precision_score (default=1)
    Returns:
        The recall score as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

Examples:

```python
import numpy as np
from recall import recall_score_
from sklearn.metrics import recall_score


# Test n.1
y_pred = np.array([1, 1, 0, 1, 0, 0, 1, 1])
y_true = np.array([1, 0, 0, 1, 0, 1, 0, 0])
print(recall_score_(y_true, y_pred))
print(recall_score(y_true, y_pred))
# 0.6666666666666666
# 0.6666666666666666

# Test n.2
y_pred = np.array(['norminet', 'dog', 'norminet', 'norminet', 'dog', 'dog',
'dog', 'dog'])
y_true = np.array(['dog', 'dog', 'norminet', 'norminet', 'dog', 'norminet',
'dog', 'norminet'])
print(recall_score_(y_true, y_pred, pos_label='dog'))
print(recall_score(y_true, y_pred, pos_label='dog'))
# 0.75
# 0.75

# Test n.3
print(recall_score_(y_true, y_pred, pos_label='norminet'))
print(recall_score(y_true, y_pred, pos_label='norminet'))
# 0.5
# 0.5
```

Exercise 09 - F1 Score

Turning directory :	ex09
Files to turn in :	f1_score.py
Forbidden function :	None
Remarks :	n/a

Objectives:

The goal of this exercise is to recreate the function recall_score of sklearn.metrics and to learn what represents the recall and how to measure it.

Instructions:

For the sake of simplicity, we will only ask you to have three parameters.

In the f1_score.py file create the following function as per the instructions below:

```
def f1_score_(y_true, y_pred, pos_label=1):
    """
    Compute the f1 score.
    Args:
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
        pos_label: str or int, the class on which to report the
precision_score (default=1)
    Returns:
        The f1 score as a float.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

Examples:

```python
import numpy as np
from f1_score import f1_score_
from sklearn.metrics import f1_score


# Test n.1
y_pred = np.array([1, 1, 0, 1, 0, 0, 1, 1])
y_true = np.array([1, 0, 0, 1, 0, 1, 0, 0])
print(f1_score_(y_true, y_pred))
print(f1_score(y_true, y_pred))
# 0.5
# 0.5

# Test n.2
y_pred = np.array(['norminet', 'dog', 'norminet', 'norminet', 'dog', 'dog',
'dog', 'dog'])
y_true = np.array(['dog', 'dog', 'norminet', 'norminet', 'dog', 'norminet',
'dog', 'norminet'])
print(f1_score_(y_true, y_pred, pos_label='dog'))
print(f1_score(y_true, y_pred, pos_label='dog'))
# 0.6666666666666665
# 0.6666666666666665

# Test n.3
print(f1_score_(y_true, y_pred, pos_label='norminet'))
print(f1_score(y_true, y_pred, pos_label='norminet'))
# 0.5714285714285715
# 0.5714285714285715
```

Exercise 10 - Confusion Matrix

Turning directory :	ex10
Files to turn in :	confusion_matrix.py
Forbidden function :	None
Remarks :	n/a

Objectives:

The goal of this exercise is to recreate the function confusion_matrix of sklearn.metrics and to learn what represents the confusion matrix.

Instructions:

For the sake of simplicity, we will only ask you to have three parameters.
Be careful to respect the order, true labels are rows and predicted labels are columns:

```
#                      [predicted labels]
#                      label_1 label_2
# [ true ] label_1        .       .
# [labels] label_2        .       .
```

In the confusion_matrix.py file create the following function as per the instructions below:

```
def confusion_matrix_(y_true, y_pred, labels=None):
    """
    Compute confusion matrix to evaluate the accuracy of a classification.
    Args:
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
        labels: optional, a list of labels to index the matrix. This may be
used to reorder or select a subset of labels. (default=None)
    Returns:
        The confusion matrix as a numpy ndarray.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

Examples:

```python
import numpy as np
from confusion_matrix import confusion_matrix_
from sklearn.metrics import confusion_matrix


y_pred = np.array(['norminet', 'dog', 'norminet', 'norminet', 'dog',
'bird'])
y_true = np.array(['dog', 'dog', 'norminet', 'norminet', 'dog', 'norminet'])
print(confusion_matrix_(y_true, y_pred))
print(confusion_matrix(y_true, y_pred))
# [[0 0 0]
# [0 2 1]
# [1 0 2]]
# [[0 0 0]
# [0 2 1]
# [1 0 2]]
print(confusion_matrix_(y_true, y_pred, labels=['dog', 'norminet']))
print(confusion_matrix(y_true, y_pred, labels=['dog', 'norminet']))
# [[2 1]
# [0 2]]
# [[2 1]
# [0 2]]
```

Optional part

Objective(s):

For a more visual version, you can add the option to your previous confusion_matrix_ function to return a pandas dataframe instead of a numpy array.

Instructions:

In the confusion_matrix.py file create the following function as per the instructions below:

```python
def confusion_matrix_(y_true, y_pred, labels=None, df_option=False):
    """
    Compute confusion matrix to evaluate the accuracy of a classification.
    Args:
        y_true: a scalar or a numpy ndarray for the correct labels
        y_pred: a scalar or a numpy ndarray for the predicted labels
        labels: optional, a list of labels to index the matrix. This may be
used to reorder or select a subset of labels. (default=None)
        df_option: optional, if set to True the function will return a
pandas dataframe instead of a numpy array. (default=False)
    Returns:
        The confusion matrix as a numpy ndarray or a pandas dataframe
according to df_option value.
        None on any error.
    Raises:
        This function should not raise any Exception.
    """
```

Examples:

```
import numpy as np
from confusion_matrix import confusion_matrix_


y_pred = np.array(['norminet', 'dog', 'norminet', 'norminet', 'dog',
'bird'])
y_true = np.array(['dog', 'dog', 'norminet', 'norminet', 'dog', 'norminet'])
print(confusion_matrix_(y_true, y_pred, df_option=True))
#          bird dog norminet
# bird        0   0        0
# dog         0   2        1
# norminet    1   0        2
print(confusion_matrix_(y_true, y_pred, labels=['bird', 'dog'],
df_option=True))
#          bird dog
# bird        0   0
# dog         0   2
```

N.B: if you fail this exercise on your first attempt, norminet will curse you forever. Yeah, you better do it right or you are in trouble my friend, big trouble !