Problem Set 1 Solution

Cell-1 (required library installation)

```
%pip install timed-decorator
```

the line above is a shell command that should be run in terminal but can be executed in jupyter notebook with the help of

1. `%` **(IPython magic command)** : This is the **recommended** way to install packages. it ensures that the library is installed into the **current** Python kernel environment.
2. `!` **(Shell Command) :** It tells the notebook to send the command following the `!` explicitly to the underlying operating system's shell (like Bash or PowerShell), exactly as if you typed it in a terminal. but can sometimes install packages into a different Python environment than the one your current notebook kernel is using.
   There are many we will see others when we will encounter them.

---

okay the question arises

1. What is a IPython kernel ?

```
is the actual process that runs your code. It is the "backend" of a Jupyter
Notebook. When you press `Shift + Enter` in a cell, the Notebook (frontend)
sends that code to the Kernel (backend) to be executed.
**Purpose**:  It handles the interactive features of a notebook, like seeing
variable outputs immediately, plotting graphs inline, and using magic
commands
```

2. What is a python environment ?

is an isolated folder on your computer that contains a specific version of the Python interpreter and a specific set of libraries (like `pandas`, `numpy`, or `timed-decorator`).

Purpose: To prevent "dependency hell." For example, Project A might need `Django 2.0`, while Project B needs `Django 4.0`. By using two separate environments, you can work on both projects without them crashing each other.
There are Common tools you might know : `venv`, `conda` , etc.

3. What is timed-decorator's work ?

it acts like a **stopwatch** for your code.
When you put the `@timed_decorator` tag above a function, it automatically tracks exactly how long that function takes to run, without you having to write any timer code yourself.

How it works (The Logic)

It wraps your function in a "layer" of extra code that does three things in order:

1. **Start the Clock:** Records the exact time just before your function begins.
2. **Run the Function:** Executes your code normally.
3. **Stop the Clock:** Records the time immediately after your function finishes and calculates the difference (`End Time` - `Start Time`).

4. What is a decorator function ?

A **decorator function** is a design pattern in Python that allows you to modify the behavior of a function or class without permanently changing its source code.

How it is built (The Anatomy)

A decorator is a function that takes another function as an argument. It has three distinct parts:

1. **The Decorator (Outer Function):** Accepts the function you want to modify.
2. **The Wrapper (Inner Function):** Where the new logic lives. It runs code before/after the original function.
3. **The Return:** It returns the *wrapper* function to replace the original.

Let's understand this with an example

```
# 1. The Decorator Function
def my_logger_decorator(original_func):
    # 2. The Wrapper (The "Security Guard")
    def wrapper():
        print(f"LOG: About to run {original_func.__name__}...") # Extra code
BEFORE
        original_func()  # <--- The Original Function runs here
        print(f"LOG: Finished running {original_func.__name__}.") # Extra
code AFTER
    return wrapper # 3. Return the new wrapped version
```

is a decorator function to be used as wrapper for a new function with `@` sign

```
@my_logger_decorator
def say_hello():
```

```
    print("Hello World!")
# When you call the function now...
say_hello()
```

Output:

```
LOG: About to run say_hello...
Hello World!
LOG: Finished running say_hello.
```

those 2 lines (1 and 3) are executed because of the decorator function

Why the library is being used here ?

to tell you the execution time difference between sklearn code and numpy code for the same logic

Cell-2 (import required library)

```
import numpy as np
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score
from timed_decorator.simple_timed import timed
from typing import Tuple
```

Cell-3 (Dummy data to use in exercise)

```
predicted = np.array([
    1,1,1,0,1,0,1,1,0,0
])

actual = np.array([
    1,1,1,1,0,0,1,0,0,0
])
big_size = 500000
big_actual = np.repeat(actual, big_size)
big_predicted = np.repeat(predicted, big_size)
```

Cell-4(Exercise-1)

Description: Implement a method to retrieve the confusion matrix values using numpy operations. Aim to make your method faster than the sklearn implementation.

```
@timed(use_seconds=True, show_args=True)
def tp_fp_fn_tn_sklearn(gt: np.ndarray, pred: np.ndarray) -> Tuple[int,
...]:
    tn, fp, fn, tp = confusion_matrix(gt, pred).ravel()
    return tp, fp, fn, tn

@timed(use_seconds=True, show_args=True)
def tp_fp_fn_tn_numpy(gt: np.ndarray, pred: np.ndarray) -> Tuple[int, ...]:
    raise NotImplementedError()

assert tp_fp_fn_tn_sklearn(actual, predicted) == tp_fp_fn_tn_numpy(actual,
predicted)
```

few questions before solving

1. what could be sklearn library logic for confusion matrix ?
2. what ravel() do here ?
3. How we should implement in numpy ? (what should be our though process ?)

(Easy question first) What `ravel()` doing here ?

it is a numpy function (aha! so sklearn confusion matrix return a numpy array) which is used to flatten the array. implies confusion matrix should be giving nd array.

actually it's true right as it give 2 $\times$ 2 matrix

```
[[tn, fp],
 [fn, tp]]
```

so ravel make it 1D array

```
[tn, fp, fn, tp]
```

our implementation in numpy

Rules:

1. write in natural english the logics then code

```
@timed(use_seconds=True, show_args=True)
def tp_fp_fn_tn_numpy(gt: np.ndarray, pred: np.ndarray) -> Tuple[int, ...]:
    # True Positive: gt=1 and pred=1
    tp = np.sum((gt == 1) & (pred == 1))
```

```
    # False Positive: gt=0 but pred=1
    fp = np.sum((gt == 0) & (pred == 1))
    # False Negative: gt=1 but pred=0
    fn = np.sum((gt == 1) & (pred == 0))
    # True Negative: gt=0 and pred=0
    tn = np.sum((gt == 0) & (pred == 0))
    return tp, fp, fn, tn
```

as `(gt == 1) & (pred == 1)` will give a matrix with 0 and 1 and `np.sum()` will add the matrix to have the respective numbers

and yes i forgot one thing. it's a new syntax as well right ?
the

```
def function (parameter: datatype ) -> return datatype :
    pass
```

this is python3 syntax a bit faster and type safe. it now knows what the function will return and what are the parameters type to check while compiling.

coming to the topic, the output is

```
tp_fp_fn_tn_sklearn(ndarray(10,), ndarray(10,)) -> total time: 0.015788836s
tp_fp_fn_tn_numpy(ndarray(10,), ndarray(10,)) -> total time: 0.000119890s
```

so the Numpy code is nearly 10 times faster.

and **sklearn uses numpy then why not the same speed ?**
because sklearn is doing a lot more than “just count matches.” It wraps robust, general, user-friendly behavior around the core counting step, and those extra checks and generalizations cost time. You can get raw, minimal NumPy code that’s faster for your very specific case — but sklearn trades raw speed for safety, flexibility and convenience.

we will talk about it in upcoming implementations

Cell-5 (Exercise - 2)

Description: Implement a method to retrieve the calculate the accuracy using numpy operations.

```
@timed(use_seconds=True, show_args=True)
def accuracy_sklearn(gt: np.ndarray, pred: np.ndarray) -> float:
    return accuracy_score(gt, pred)
```

```
@timed(use_seconds=True, show_args=True)
def accuracy_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    raise NotImplementedError()

assert accuracy_sklearn(actual, predicted) == accuracy_numpy(actual,
predicted)
```

Solution:

```
@timed(use_seconds=True, show_args=True)
def accuracy_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    # get the respective counts
    tp, fp, fn, tn = tp_fp_fn_tn_numpy(gt, pred)
    # implement the accuracy formula
    return (tp + tn) / (tp + fp + fn + tn)
```

Cell-6 (Exercise - 3)

Description: Implement a method to calculate the F1-Score using numpy operations. Be careful at corner cases (divide by 0).

```
@timed(use_seconds=True, show_args=True)

def f1_score_sklearn(gt: np.ndarray, pred: np.ndarray) -> float:
    return f1_score(gt, pred)

@timed(use_seconds=True, show_args=True)
def f1_score_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    raise NotImplementedError()

assert f1_score_sklearn(actual, predicted) == f1_score_numpy(actual,
predicted)
```

Precision = $\frac{TP}{TP+FP}$
Recall = $\frac{TP}{TP+FN}$
F1-Score = $2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$

```
@timed(use_seconds=True, show_args=True)
def f1_score_sklearn(gt: np.ndarray, pred: np.ndarray) -> float:
    return f1_score(gt, pred)
```

```
# precision function
def precision_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    tp, fp, fn, tn = tp_fp_fn_tn_numpy(gt, pred)
    return tp / (tp + fp)

# recall function
def recall_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    tp, fp, fn, tn = tp_fp_fn_tn_numpy(gt, pred)
    return tp / (tp + fn)

@timed(use_seconds=True, show_args=True)
def f1_score_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    precision = precision_numpy(gt, pred)
    recall = recall_numpy(gt, pred)
    return 2 * (precision * recall) / (precision + recall)

assert f1_score_sklearn(actual, predicted) == f1_score_numpy(actual,
predicted)
```

even if the code is write the assertion will be wrong as we are comparing 2 floats with `==` sign. why it is bad ? (search for the answer)

```
@timed(use_seconds=True, show_args=True)
def f1_score_sklearn(gt: np.ndarray, pred: np.ndarray) -> float:
    return float(f"{f1_score(gt, pred):.5f}")

# precision function
def precision_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    tp, fp, fn, tn = tp_fp_fn_tn_numpy(gt, pred)
    return tp / (tp + fp)

# recall function
def recall_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    tp, fp, fn, tn = tp_fp_fn_tn_numpy(gt, pred)
    return tp / (tp + fn)

@timed(use_seconds=True, show_args=True)
def f1_score_numpy(gt: np.ndarray, pred: np.ndarray) -> float:
    precision = precision_numpy(gt, pred)
    recall = recall_numpy(gt, pred)
    f1 = 2 * (precision * recall) / (precision + recall)
    return float(f"{f1:.5f}")
```

```
assert f1_score_sklearn(actual, predicted) == f1_score_numpy(actual,
predicted)
```

Here i trimmed to 5 decimal places and it accepted
Now it's
0.72727
0.72727

Earlier it was
0.7272727272727273
0.7272727272727272

Have Fun ;)