1 Sketching the overall structure of ACTS

As also stated in the Github Repo, ACTS is defined as a query strategy to be put inside `va_builder`.

Similarly to other query strategies, we defined ACTS as a wrapper class for the actual algorithm. The reason for this is that ACTS needs some elements to work, which are patterns and instances. The value of this elements must be stored after every call of the algorithm. Rather than defining function attributes as:

```
def ACTS():
    ...
    <insert algorithm here>
    ...

ACTS.patterns = set_patterns()
ACTS.instances = set_instances()
```

We chose to define a wrapper class whose constructor does not take any argument. The actual algorithm will be then put inside `__call__`.

So for example:

```
class ACTS:

    def __init__(self):
        self.patterns = None
        self.instances = None

    def __call__(self):
        ...
        <insert algorithm here>
        ...
```

1.1 Before The Algorithm

First, the ACTS must be defined and used as an argument inside the `almanager` constructor:

```
acts = ACTS()
alm = ALManager(..., query_strategy = acts, ...)
```

Then, the query strategy is used in the method `query` of class `ALManager`. First, a function called `select_query_strategy` is called. The function performs some controls on the state of the data and on the type of query strategy and returns something like: `partial(alm.query_strategy, ...additional arguments, depend on qs)`.

```
def select_query_strategy(alm):
    ...
    if <control> :
        return partial(...)
    elif <control> :
        return partial(...)

    ...
```

For ACTS, it is necessary do add an `elif` clause before the last one. After the clause, we return the object with partial:

```
    elif np.all(
        np.in1d(
            ["DL", "L", "Li"],
            getfullargspec(alm.query_strategy).args,
        )
    ):
        DL, L, Li = alm.dm.get_training_data()
        X, _ = alm.dm.get_X_pool()
        return partial(
            alm.query_strategy,
            X = X,
            DL = DL,
            L = L,
            Li = Li
        )
```

This should be enough to apply the actual algorithm, which is now integrated inside `modAL`.

1.2 The actual algorithm

Here a general schema of the first part of ACTS is given:

- When the object is called for the first time, a small initial training set is available, which is referred to as D_{L0}. Starting from this, patterns and instances are initialized. Each instance is a pattern by itself.
- From here on, each time the object is called, it comes with a new training set D_{Ln}, containing more labelled instances. New instances are assigned to their nearest pattern.
- Then, each pattern is checked, if a pattern contains more than one label (it's *mixed*), it must split into two, finding the one that achieves the optimal information gain.

  To do this, a total of 4 approaches is available:

  – Original method: brute force pruned. Always achieves optimal solution, might be slow, needs to be implemented.
  – Fast-shapelets: Novel method, slightly faster and usually finds the optimal solution. Needs to be implemented (available in cpp).
  – Ultra-fast shapelets: sacrifices accuracy for speed, might obtain suboptimal solutions. Needs to be implemented.
  – Gradient descent: slightly faster than the original method, has an hyperparameter that has to be tuned. Already implemented in `tslearn`.

  For now, we'll use the last method. If everything is good, the best option is fast-shapelets. (Boy it'll be fun).

- After splitting the patterns, reassign each instance to its nearest pattern.
- After this, we have to calculate the following quantity for each pattern pt and each label ℓ:

  $$P(pt|\ell) = Multi(p_1, \dots, p_L) \tag{1}$$

  Which is multinomial distribution whose parameters need to be estimated with maximum likelihood (needs to be implemented).

The second part of the algorithm aim at calculating uncertainty and utility of each instance.

1.3 Defining patterns and instances

When the algorithm is called for the first time, patterns and instances need to be initialized.

We'll now define the structure of this elements:

Instances Instances are labelled elements in the dataset. Every has a unique index (in the argument `Li`), and some values (an np array). When applying this algorithm we need additional information about the instances, so: **All the instances are collected inside a pandas dataframe**, the keys of this dataframe are the indexes of the instances, the columns are the following:

- `ts` is the actual time series (np.array).
- `label` is the label of the instance (int).
- `near_pt` is the key of the nearest pattern
- `pt_probas` is an array containing $P(X|pt)$ for each pt.

Patterns A pattern is a small sequence of values that represents a part of the dataset. Every pattern must contain some additional information, the same way for instances. So, **All the patterns are collected in a Pandas dataframe**. Each row has a key and some fields.

In this case patterns do not have a unique index, so the key is defined as the following: given a pattern $pt = [x_1, \dots, x_n]$, its key will be:

$$mean(pt) \longrightarrow round(\ ,6) \longrightarrow hash()$$

It is highly unlikable that two patterns will have the same mean value until the sixth decimal, also, if two patterns are the same, there is no need to keep them divided, they will be one pattern, so this seems reasonable.

Now that the key is defined, we define other fields.

- `ts` is the sequence of values.
- `inst_keys` array containing the keys of the instances assigned to that pattern. Basically earlier we assigned each instance to its nearest pattern. Here we collect all the instances with a certain pattern as nearest.

- `labels` is an array containing the labels of the instances in `inst_keys`.
- `l_proba` is an array containing the values $P(pt|\ell)$ for each label ℓ.