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 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

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`.

each

1.3.1 Additional properties

The class ACTS, other than instances and properties, contains other two properties: `lam, probas`:

- lam : value of λ used in the modeling part (exponential distribution).
- probas : values of $p_1, \ldots, p_L$ used in the modeling part (multinomial distribution).

2 Implementation

In this section, we'll show how the first part of the algorithm is actually implemented.

As we said previosly, the objective of this part is to integrate the new instances and update the patterns.

The steps of the algorithm are the following:

- When new labelled instances are passed, they are added to the existing instances.
- New patterns or *shapelets* are computed
- Each instance is assigned to a pattern
- All the parameters for the modeling part are updated with respect to the new patterns

For the first part, not much has to be explained, the algorithm is called with the arguments `DL, L, Li`, which represent respectively the whole set of labelled instances, their labels and their indexes.

At each iteration, the algorithm looks for new indices inside `Li` and adds them to the existing instances in the correct format, the instances are stored inside a dataframe with the following structure:

- `key`: containing indices of the instances, in integer, datetime, or float format
- `ts`: containing arrays with actual instances
- `label`: integer
- `near_pt` : containing the key of the pattern each instance is assigned to

To compute new shapelets for the dataset, the algorithm makes use of two libraries: `pyts` and `sklearn`. In particular, we make use of the transformation module for the first and of the DecisionTree module for the second.

First we compute the *shapelet transform* of each labelled instance. This procedure finds a set of shapelets that might be useful to classify data inside the labelled instances, then, it calculates the distance between the instances and each shapelet.

This operation is called *shapelet transform*, since we transform each instance into a set of feature. Where each feature is the distance between the instance and each shapelet found. It is useful to say though that this transform *does not substitute* the instances, but it's only an intermediate step.

Once the shapelet transform of each instance has been calculated, we need to find the most useful shapelets, that will constitute our **patterns**.

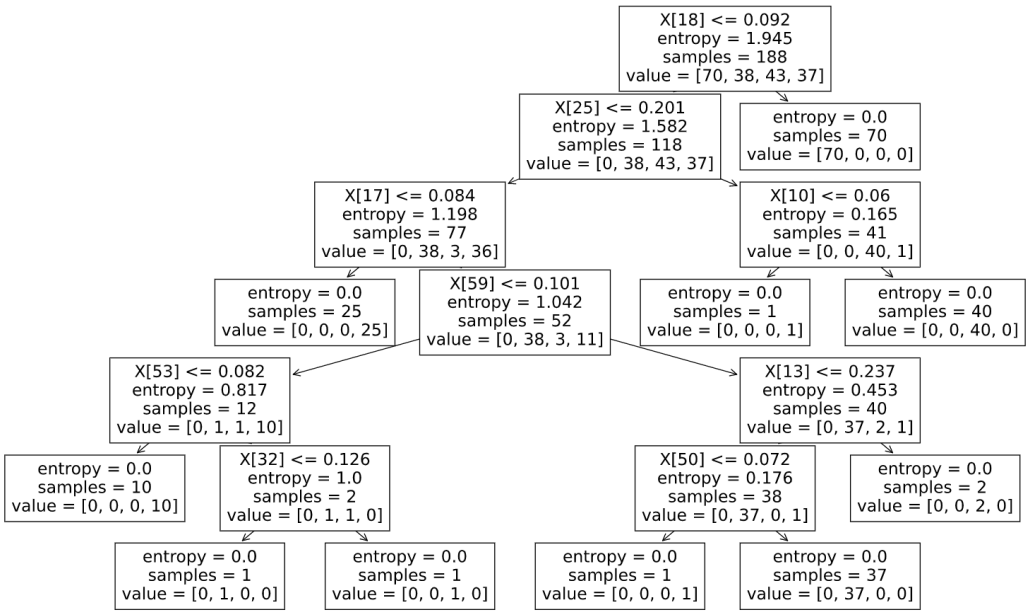

Figure 1: An example of tree structure acting on the shapelet transorms

To do this, we make use of a decision tree, from the library `sklearn`.

The decision tree acts on the shapelet transform of the data and finds the best splits according to the *information gain*.

We can see an example in Figure (1): each feature is a distance between shapelet and instance. The first node splits on shapelet 18. 70 instances of the same label are grouped. They are assigned to shapelet 18, which will be the first pattern found. A completely pure pattern.

Going down the tree, we apply the same logic to every leaf node, by assigning all the instances in it to the decision node just above.

Another example: there is a decision node which splits on feature 59, but it has no leaf node, so no instance will be assigned to pattern 59.

In this way each instance is assigned to a pattern in order to maximize the information gain.

Once the patterns have been computed and each instance has been assigned to a certain pattern, we start the modeling part.

The objective is to calculate some useful parameters by MLE.

The first parameter is λ. The algorithm model the following probability with an exponential distribution:

$$P(X_i|pt) = \exp(-\lambda_{pt} Dis(X_i, pt)) \tag{2}$$

Where X_i is a time series and pt is a pattern. Dis is the distance function between the two.

We have a λ parameter for each pattern, that we have to estimate via MLE.

The results are saved inside a the column `lambda` inside the pattern dataframe.

The second set of parameters to be estimated is the probabilities:

$$P(pt|l) = Multi(p_1, \dots, p_l) \tag{3}$$

Where *multi* stands for multinomial. These parameters are also saved in a column of the patterns dataframe, called `l_probas`.

3 Second part

The second part of the algorithm aims at calculating, for each unlabelled instance, its **uncertainty** and **utility**.

First we'll define some thing that we have until now:

- Instances. The labelled examples until now, we'll refer to these instances as Y_i, they are contained inside D_L, each one of these instances has a label and an index, we'll refer to these as $Y_i.l$, $Y_i.i$. Also, each instance is assigned to a pattern, we'll refer to this patterns as $Y_i.pt$.
- Patterns (or shapelets). Short subsequences that represents part of the instances, each instance is assigned to a pattern. Each pattern has its own λ parameter, which is used to calculate the quantity $P(Y|pt)$. Also, each parameter has its own $P(pt|y = l)$ for each l inside the possible labels.
- Unlabelled instances. We'll refer to these as X_i, contained inside D_U.
- We also have a distance function between time series, or time series and patterns: `_dis(Y, X)`.
- We also have a **label set** L, i.e. the list of all possible labels inside the dataset. If the classification is binary the label set will be $L = \{0, 1\}$, for example.

3.1 Uncertainty calculation

The first quantity we have to measure is the uncertainty related to an instance $X \in D_U$.

The steps are: **for each instance X**:

- Calculate its k-nearest neighbors in D_L. We'll refer to these as $\{Y_j\}_{j \in 1 \dots k}$, so $Y_1, \dots, Y_k$ where Y_1 is the closest instance and Y_k the farthest of the knn. From this, we also need to retrieve the following quantities:

$$d_1 = dis(X, Y_1) \tag{4}$$

$$d_k = dis(X, Y_k) \tag{5}$$

- Calculate, for all possible $l \in L$ the quantity:

$$\bar{P}(y = l|X) = \sum_{Y_j \in \{Y_1, \dots, Y_k\}} P(X|Y_j.pt) \cdot P(Y_j.pt|y = l) \tag{6}$$

  Where the both term are available: the function `calculate_probax(X, pt)` calculates the first part, while the second term is contained in the `l_probas` column of the *patterns* dataframe. It is an array where the first entry is $P(Y_j.pt|y = 1)$ and so on for each possible label.

- Then we calculate the normalizer:

$$Z = \sum_{l \in L} \bar{P}(y = l|X) \tag{7}$$

- Then we normalize the quantities we just calculated with Z, obtaining:

$$\hat{P}(y = l|X) = \frac{1}{Z} \bar{P}(y = l|X) \tag{8}$$

- Then finally we calculate the uncertainty for the unlabelled instance X:

$$Uncr(X) = \sum_{l \in L} \hat{P}(y = l|X) \log\left(\hat{P}(y = l|X)\right) \frac{d_1}{d_k} \tag{9}$$

3.2 Utility calculation

The utility needs to be calculated for each instance $X_i \in D_u$, but needs the so called *reverse nearest neighbors* of X, which are the instances $\{Y_1, \ldots, Y_n\}$, that have X_i as a k nearest neighbor.

To do this:

- We calculate, for each instance $Y_i \in D_L$, it's k nearest neighbors in D_U.
- For each instance $X \in D_U$, we check all the instances $Y_j \in D_L$ that have X_i as a knn

This way we obtain the set of reverse nearest neighbors of X_i, which we'll call $RN(X_i)$. We'll identify the instances in this set with $\{Y_1, \ldots, Y_k\}$.

Now, we calculate the following quantities:

$$dis(X_i, Y_j) \qquad \text{for } Y_j \in RN(X_i) \tag{10}$$

Of which we'll calculate:

$$mDis(X_i) = \max_j dis(X_i, Y_j) \qquad \text{for } Y_j \in RN(X_i) \tag{11}$$

And then:

$$SimD(X_i, Y_j) = 1 - \frac{dis(X_i, Y_j)}{mDis(X_i)} \qquad \text{for } Y_j \in RN(X_i) \tag{12}$$

Now the first part of the utility is done, we have to calculate the second part.

To do this, we'll calculate the nearest neighbors of X_i in D_L. We'll refer to these as $\{Y_1, \ldots, Y_k\} \in LN(X_i)$.

Now, we have to calculate the follwing quantity for each possible pattern:

$$\Psi(X_i, pt) = \sum_{Y_j \in LN(X_i)} P(X_i|Y_j.pt) I(Y_j.pt = pt) \qquad \text{for } pt \in patterns \tag{13}$$

Where the first term is the same of Equation 6 first part, so available with the function `calculate_probax(X, pt)`, where pt will be the pattern assigned to Y_j.

The second part, on the other hand is defined as follows:

$$I(Y_j.pt = pt) \begin{cases} 0 & \text{if } Y_j.pt \text{ is not the same as } pt \\ 1 & \text{if } Y_j.pt \text{ is the same as } pt \end{cases} \tag{14}$$

For the values of $\Psi(X_i, pt)$ calculated for each pt, we calculate:

$$Z(X_i|PT) = \sum_{pt} \Psi(X_i, pt) \tag{15}$$

Now we have that:

$$P(X_i|pt) = Z(X_i|PT)^{-1}\Psi(X_i, pt) \tag{16}$$

This element is the basic element for the calculation of the utility, it must be defined also for a labelled instance $Y_i \in D_L$, so, to calculate it:

- Find the knn of Y_j in D_L, which will be referred to as $LN(Y_j)$.
- Calculate:

$$\Psi(Y_j, pt) = \sum_{Y_k \in LN(Y_j)} P(Y_j|Y_k.pt)I(Y_j.pt = pt) \quad \text{for all } pt \in patterns \tag{17}$$

- Calculate:

$$Z(Y_j|PT) = \sum_{pt} \Psi(Y_j, pt) \tag{18}$$

- Calculate:

$$P(Y_j|pt) = Z(Y_j|PT)^{-1}\Psi(Y_j, pt) \tag{19}$$

Now, let suppose that all the possible patterns are $[pt_1, pt_2, \ldots, pt_n]$, we define the following arrays as:

$$P(X_i|PT) = [P(X_i|pt_1), \ldots, P(X_i|pt_n)] \tag{20}$$
$$P(Y_j|PT) = [P(Y_j|pt_1), \ldots, P(Y_j|pt_n)] \tag{21}$$

And we introduce the following quantity:

$$SimP(X_i, Y_j) = 1 - JSD(P(X_i|PT), P(Y_i|PT)) \tag{22}$$

Where JSD is the Jensen Shannon Distance between the arrays, available in `scipy`.

Then:

$$Sim(X_i, Y_j) = SimD(X_i, Y_j) \cdot SimP(X_i, Y_j) \tag{23}$$

Lastly, once we defined everything, and remembering that $RN(X_i)$ is the set of reverse k nearest neighbors in D_L of $X_i \in D_U$, we define the utility of an instance

$$Uti(X_i) = \sum_{Y_j \in RN(X_i)} Sim(X_i, Y_j) \tag{24}$$

And its question informativeness:

$$QI(X_i) = Uti(X_i) + Uncr(X_i) \tag{25}$$

We select the n most informative instances to query.

4 Differences between original proposal and ACTS

We can summarize the differences between this implementation of ACTS and the one proposed in the paper as:

- Loop, scope of the code
- Automatic addition of certain instances to the labelled data is not present here
- Pattern creation and assigning

4.1 Loop, scope of the code

For this first aspect, we would like to remark that this implementation of ACTS is built to be used as a query strategy inside va builder.

The code in the paper describes a full active learning loop. Starting from a few labelled instances, calculate uncertainty and utility, then query, etc.. Then repeat until you have enough labels.

This code does not deal with the active learning loop, but only performs a query given a certain set of labelled and unlabelled data.

So given DL, X, L, Li as in documentation, returns the most useful instances.

4.2 Automatic addition of certain instances

In the paper it is said that if the uncertainty of an instance is equal to 0, then this is added to the labelled data with the predicted label, since there's no uncertainty on it.

This feature is not present here. This is because of the particular structure of the code.

ACTS is implemented as a query strategy, so it does not interact with the active learning manager, and thus it cannot add a certain instance to the labelled dataset.

4.3 Pattern creation and assigning

Another difference is in how the patterns (or shapelets) are found. And how each instance is assigned to a pattern.

In the original paper, the method to find patterns in the dataset is the one described in Time series shapelets: a new primitive for Data Mining.

This is basically a brute force algorithm. For each possible subsequence in the dataset, the algorithms tries to determine the optimal split point and information gain.

After having checked all the possible candidates, it selects the best one.

Despite using early abandon strategies, this algorithm takes a long time.

When the patterns have been found, a binary tree is constructed. To assign an instance to a pattern it is necessary to run down the tree. As shown in the paper above (figure 13).

Inside the acts paper, the logic is the following:

- Start from a small number of labelled instances. At first, each instance is considered as a pattern.
- Once other labelled instances arrive, they are assigned to a pattern.
- If instances of different labels are assigned to the same pattern, apply pattern splitting to that particular pattern

This is not the procedure applied in the code. We used a different approach, that has the same logic and should be equivalent.

To retrieve the patterns, we don't use a brute force algorithm, but a rather more recent algorithm, described in Learning Time Series shapelets.

This is an iterative algorithm that generates some *candidate patterns*, meaning a series of patterns that might be helpful in discriminating the classes in the dataset.

So the logic that we used is the following:

- Compute pattern candidates
- Calculate for each labelled instance the shapelet transform: vector of distances between the instance and each pattern candidate.
- Run a decision tree on these shapelet transform to find the most important patterns.

The decision tree has the same exact function of the binary tree of the original algorithm.

The only difference is in how the shapelets are computed.

Also, in our case, since the search for pattern candidates is less prohibitive in time, we do not perform initialization and pattern splitting strategy described earlier.

Instead, we look for the best patterns in all the dataset at each call of the function.

That is, we don't perform the sequence of operations described in the first list of this subsection, but rather:

- When new labelled instances arrive, we add them to the pool of labelled instances
- Delete previous patterns
- Look for new patterns using all the available data

When training the decision tree, not all pattern candidates are used. Some of them might not be used by the tree. They are removed in "drop_empty_pattern".