Coursework 3

Step 1: Obtain the Code

In your base repo:

```
git checkout lab3
./refresh.sh
```

Install the requirements

```
pip install -r requirements.txt
```

Notice the location in which `pip` installs the `pytest` tool which you'll need to run. The easiest way to access `pytest` will be to add that directory to your `PATH`. On Ubuntu lab machines you can achieve this with the command

```
export PATH=~/.local/bin:"$PATH"
```

Familiarise yourself with the code. in the directory, you will find the subdirectory `learning` which contains the source code (stubs). You will implement the decision tree learner in `learning/attr_learner.py` and FOIL in `learning/rule_learner.py`. The files contain a skeleton of the ready algorithm with method and function stubs that you will need to implement. Do not change their signature, as tests found in the `tests/` directory will be used to evaluate your solutions automatically. Obviously, also do not change the tests.

Step 2: Run the tests

You will have to implement your solutions against public and hidden test cases. Public solutions are available to you and you can look at them in the folder `tests/public/`. Hidden solutions test your implementation further, for some interesting input configurations or edge cases and are not available to you. To run all tests, run

```
pytest
```

To run all tests in a file, e.g. `tests/public/test_attr_learner.py`, run

```
pytest tests/public/test_attr_learner.py
```

To run a specific test case in the file, e.g. `test_overall_public` run

```
pytest tests/public/test_attr_learner.py::test_overall_public
```

Pytest captures your console output by default. If you don't want that, you can append the `--capture=no` flag

```
pytest --capture=no [...]
```

And if you - as you should - use the python `logging` stdlib instead of `print`, you can even control the granularity of messages `pytest` will let through with

```
pytest --log-cli-level=LEVEL [...]
```

where `LEVEL` is `DEBUG`, `INFO`, `WARN`, etc.

Exercise 1: Decision tree learning [7pt]

In this exercise, you will implement the Decision Tree Learner as presented in the lecture. The skeleton is located in `learning/attr_learner.py`. Tests for this exercise can be found in `tests/public/test_attr_learner.py` It will take a dataset of examples that are represented by a combination of values of a fixed set of attributes and produce a learned hypothesis of the following format:

```
ATTR_1_NAME = ATTR_1_VAL_1 AND ... AND ATTR_N_NAME = ATTR_N_VAL_K
OR
...
OR
ATTR_1_NAME = ATTR_1_VAL_N AND ... AND ATTR_N_NAME = ATTR_M_VAL_K
```

The dataclasses `AttrLogicExpression`, `Conjunction` and `Disjunction` represent the hypotheses above in python code.

If you are unfamiliar with dataclasses, click here for a quick refresher.

`AttrLogicExpression` is an abstract base class, representing a logic expression. `Conjunction` is a conjunction of specific attribute values. the field `attribute_confs` takes a `dict` mapping from attribute names to their values. In other words a `Conjunction c` will be true for an example `e` iff

```
all(c.attr_values[x] == e[x]
    for x in k in set(self.attribute_confs).intersection(example))
```

This functionality is already implemented. You can test this by calling the conjunction, like so:

```
c(example)
```

Similar is the case with disjunctions, a `Disjunction d` has a field `conjunctions` which is a list of conjunctions. A `Disjunction d` is true for an example `e` if all its conjunctions are true. Similarly, you can test this with `d(e)`.

Other relevant dataclasses are the abstract base class `Tree` representing a Decision Tree element, `Node` representing a decision and `Leaf` representing a classification. More details can be seen in the source code documentation.

Finally, type abbreviations used are `Example = Dict[str, Any]` which means that one single example is a `dict` mapping from attribute names to their values and `Examples = List[Example]` represents a `list` of examples.

A special attribute is `target` which represents the classification of the example. For now, you can assume that `target` is boolean, all attributes are the same for all `Examples` and all values are given (no `NULL` values) for all examples.

Implementing the termination [1p]

Like you have seen in the lecture, the termination cases of the learning algorithm include checking whether all remaining examples have the same classification and what the majority value is. For the former, you should implement the function `same_target(examples: Examples) -> bool` that takes a list of examples and returns True if they have the same `target` attribute. For the latter you should implement the function `plurality_value(examples: Examples) -> bool`, which returns the `target` value that appears most in the given dataset.

The tests `test_same_target_public` and `test_plurality_value_public` should succeed now.

Implementing Binary Entropy [1p]

Now you should implement the binary entropy as shown in the lecture, in the function `binary_entropy(examples: Examples) -> float`. The function will take `Examples` and return the binary entropy with regard to the classification.

Watch out for the edge cases!

The test `test_binary_entropy_public` should succeed now.

Implementing information gain [2p]

With binary entropy implemented, you can implement the information gain now. The method is `DecisionTreeLearner.information_gain(self, examples: Examples, attribute: str) -> float`. Like in the lecture videos, the method should take a dataset `examples` and an attribute name `attribute`, and compute the information gain by splitting the dataset on that attribute. With the information gain, you are also be able to implement `DecisionTreeLearner.get_most_important_attribute(self, attributes: List[str], examples: Examples) -> str`: that takes a list of attribute names and a dataset and returns the attribute with the highest information gain

Watch out for edge cases here as well!

The following tests should succeed now:

- `test_information_gain_public`
- `test_get_most_important_attribute_public`

Implementing Tree to Logic Expression [2p]

In the lecture we mentioned that a decision tree can be converted to an equivalent logic expression. Implement the function `to_logic_expression(tree: Tree) -> AttrLogicExpression` that takes a decision tree `tree` and converts it into an `AttrLogicExpression` which, as described above is a disjunction of conjunctions of attribute configurations. Note that the conjunctions should be ordered such

that the topmost attribute in a path in the decision tree should appear leftmost in the conjunction. The order of the disjunctions is irrelevant. Again, you won't need to do the string conversion, just represent the tree as a combination of `Conjunction` and `Disjunction` python objects, they already implement the corresponding `__str__` methods to convert them a string of the form shown above.

Implementing The full algorithm [1p]

Now you're equipped to bring the pieces together and implement the full algorithm as presented in the lecture.

The recursive function `decision_tree_learning(self, examples: Examples, attributes: List[str], parent_examples: Examples) -> Tree` should be implemented to take a dataset, a list of attributes that describe examples of that dataset and all examples from the parent decision step and return an induced decision tree.

Make sure to watch out for edge cases and implement all terminating cases correctly!

The test `test_overall_public` should succeed now.

Exercise 2: FOIL [9p]

In this exercise, we will make use of pyswip, a bridge between Python and Prolog. Yes, we're going to call Prolog from Python. Make sure, that `swipl` is accessible from your path by testing whether you can run `swipl` successfully in your environment.

The overall layout will be the following: given a kb from a text file and positive and negative examples of the target relation to be learned, the goal is to learn the relation as a disjunction of horn clauses.

The skeleton implementation as well as the Python objects representing elements of Prolog can be found in `learning/rule_learner.py`. Tests are located in `tests/public/test_rule_learner.py`.

Take a look at the documentation of the Python logic objects to understand how they are working.

Note: The `pyswip` bridge can be wonky at times, not clearing the knowledge base between test runs. I recommend to run tests involving the KB in isolation rather than all together. This is especially relevant for the last two tasks in this exercise.

Get predicates from the KB [1p]

Implement `_FOIL.get_predicates(self) -> List[Predicate]` which will extract all relevant predicates from the kb.

Hint 1: to access Prolog from within `get_predicates`, use `self.prolog`

Hint 2: to access the KB file path, use `self.dataset.kb`.

Hint 3: Prolog's built-in predicate `predicate_property/2` might come in handy! Test it and pay attention to the second argument. How can you make use of it to effectively filter for relevant predicates?

The test `test_get_predicates_public` should succeed now.

Implement covers [1p]

To check whether an example is covered by a Horn clause, implement `covers(self, clause: HornClause, example: Example) -> bool`. This will take a horn clause and an example and return true if the example is covered by the clause. Similar to above, you can make use of Prolog here, to make your life easier.

The test `test_rule_covers_public` should succeed now.

Generate Candidates [2p]

Just as discussed in the lecture, the function

```
_FOIL.generate_candidates(self,
                          clause: HornClause,
                          predicates: List[Predicate]
                          ) -> Generator[Expression, None, None]
```

shall take a horn clause and return a python generator that contains all (reasonable) literals which specialise the clause. Follow the lecture video, but for now, don't worry about negation or any of the other more advanced use cases such as recursion, (dis)equality or arithmetics.

When introducing literals with new variables, make sure that the variables are named uniquely! Implement the function `_FOIL.unique_var(self)` to return the next unique variable every time it's called. For simplicity (and to comply with the test cases later), the variables should be named `V_i` where `i` is an integer starting from `0` and increasing by `1` with every subsequent call.

The test `test_generate_candidates` should succeed now.

Note on python generators: If you don't feel comfortable using generators it's also okay to return lists. The tests should not fail because of this. You might just get annoying type checking warnings if you use an IDE/editor with type checking support.

Extending examples [1p]

To extend an example with all valid substitutions according to the knowledge base and subject to a given expression, as seen in the lecture videos, implement

`extend_example(self, example: Example, new_expr: Expression) -> Generator[Example, None, None]`. This function shall take an example and an expression, and return a python generator of examples (that means potentially more than one) that represent a valid substitution. See lecture video for an example.

Hint: Use Prolog here. Think about how to formulate a Prolog query to get the substitutions.

The test `test_extend_example_public` should pass now.

Foil information gain [2p]

Now you're in the position to implement the method that guides the search for the specialisations: `_FOIL.foil_information_gain(self, candidate: Expression, pos_ex: Examples, neg_ex: Examples) -> float`. This method shall take a candidate literal (as generated by `generate_candidates`), the dataset consisting of positive and negative examples and evaluate the information gain from the candidate. See lecture video for an example.

Make sure to implement `is_represented_by(example: Example, examples: Examples) -> bool` to calculate the factor `t` of the FOIL information gain formula.

Once again, pay attention to edge cases!

The tests `test_foil_ig_public` and `test_is_represented_by_public` should succeed now.

Putting it all together [1p]

Now you're equipped to put everything together. Implement

```
_FOIL.foil(self,
          positive_examples: Examples,
          negative_examples: Examples,
          predicates: List[Predicate],
          target: Literal) -> List[HornClause]
```

which shall be the outer loop of the FOIL algorithm and

```
_FOIL.new_clause(self,
                 positive_examples: Examples,
                 negative_examples: Examples,
                 predicates: List[Predicate],
                 target: Literal) -> HornClause
```

which shall be the inner loop of the algorithm according to the pseudo-code presented in the lecture.

Make sure that your code terminates!

The following tests should succeed:

- `test_new_clause_public`
- `test_overall_public`
- `test_overall_public_2`

Recursive case [1p]

Our FOIL algorithm isn't quite good so far. Make it learn recursive relations!

Hint: you will want to modify which predicates are allowed at which point.

The test `test_recursive_public` should run now. (It takes a while)

Exercise 3: Make it better [4p]

Well done, if you've arrived here, all your tests should succeed.

However, the implemented algorithms so far are pretty basic. It's time to make them better! You can choose one (or multiple) algorithms and improve them along the following metrics:

- Accuracy with noisy data
- Accuracy with less training examples
- Training time

Alternatively, you can make conceptual improvements to the algorithms (e.g. such as introducing recursion in the example above). In that case you should also provide a corresponding dataset where the base algorithm fails, but your implementation succeeds.

Make sure to provide explanations for what you did in a separate `Exercise3.md` file.

Note that in order to get full marks here, you will need to implement multiple "impressive" improvements. Since FOIL can be seen as conceptually more challenging, the improvements on FOIL will be seen as more "impressive".

The functionality to compare algorithms along the metrics and to generate datasets is provided. To find out how to use it and how to implement your own improved algorithms, make sure to check out the video explaining the course work and `readme-scripts.MD`.

Make sure to document well what you have done! Try to describe *what* you did, *why* you did it what you *hypothesise* will happen and whether experiments *confirm* or *reject* your hypothesis.

Note that a well documented failure will probably give you a better score than a "mystic success".

Submission

Submission follows the usual mechanism. Add and commit your code, then and tag it with the `./submit.sh` command. For exercise 1 and 2 you will need submit your (documented) code. For exercise 3, you will need to submit your code as well as a readme labelled `Exercise3.md` that describes the changes you made, how they improve upon the algorithms, and the commands to replicate your evidence in support of your arguments, i.e: what do the markers need to run to see the improvements for themselves?

Marking scheme

The weight of the coursework for the final course grade is equal to that of the previous assignments.

The rationale behind the marking scheme is the following: The test cases released to you represent a bare minimum example of the intended functionality that you need to implement. Further *hidden* test cases might check for additional interesting scenarios where the implemented methods/functions are supposed to work, for example some edge cases. These will not be released, but will be taken into consideration when determining the final grade. Feedback will be provided to you based on whether the tests succeed or fail. Furthermore, you're expected to understand the functionality of your code as evidenced by an appropriate documentation of the functionality/comments in the code. This results in the following marking scheme for all Exercises 1 and 2:

Grade	Explanation
0	No reasonable attempt to solve the exercise has been made
0.25	A reasonable attempt has been made, but public tests fail
0.5	Public tests succeed but all hidden tests fail
0.75	Public tests succeed and some hidden test fails
1	All (public+hidden) tests succeed

If the code is not sufficiently documented, the overall grade will be reduced up to 1p per exercise at the discretion of the markers.

Example for insufficient explanation/documentation:

```
def some_func(text):
    text = str(text)
    # get the name
    name = text[:text.find("(")].strip()
    # get the arity
    arity = len(re.findall("Variable", text))
    # if arity zero
    if arity == 0:
```

```
        # find arity
        arity = len(re.findall(",", text)) + 1
    # return predicate
    return Predicate(name, arity)
```

Example for exhaustive explanation/documentation (Note that you should aim for sufficient rather than exhaustive):

```
def some_func(text):
    # assuming input is a single predicate with its
    # arguments in valid prolog syntax
    text = str(text)
    # then name of predicate comes before
    # the first bracket in expression
    name = text[:text.find("(")].strip()
    #  and arity is either defined by number
    #  of Variable in expression
    # (as returned by predicate_property/2)
    arity = len(re.findall("Variable", text))
    # or, if not result of that query
    if arity == 0:
        # then we just count the commas that separate
        # the arguments
        arity = len(re.findall(",", text)) + 1
    return Predicate(name, arity)
```

Note that implementing a function/method to directly (and only) to return the expected test value will automatically and inevitably result in 0 points for that exercise. To illustrate, the following test/implementation example will give 0 marks although the test is passing:

```
def test_foo():
    assert foo("bar") == 5


def foo(arg):
    return 5
```

Exercises worth more than one mark are scaled accordingly.

For Exercise 3, the mile stones for marks are set as follows:

Grade	Explanation
0	No reasonable attempt to improve any of the algorithms has been made
1	A reasonable attempt to improve one of the algorithms has been made, but without any visible impact

Grade	Explanation
2	A good attempt to improve one of the algorithms has been made, the improvements are visible
3	An impressive improvement to one the algorithms have been made, the improvements are visible and well documented. Alternatively, a new algorithm has been implemented and its advantages have been demonstrated clearly
4	Like above but more than one improvement/algorithm