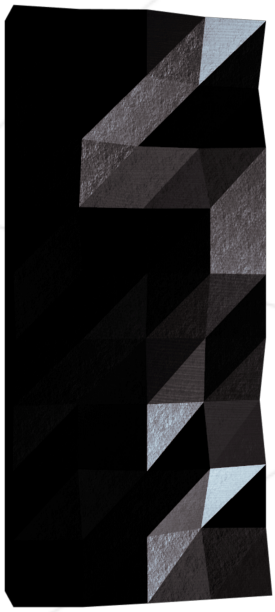

Ocaml

Imperative features - 1

Summary: The main theme of this module is related to features linked to imperative programming in ocaml.

Version: 1.00

Contents

Chapter I

General rules

- Your project must be realized in a virtual machine.
- Your virtual machine must have all the necessary software to complete your project. These softwares must be configured and installed.
- You can choose the operating system to use for your virtual machine.
- You must be able to use your virtual machine from a cluster computer.
- You must use a shared folder between your virtual machine and your host machine.
- During your evaluations you will use this folder to share with your repository.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter II

Ocaml piscine, general rules

- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names, etc.
- Unless otherwise explicitly stated, the keywords `open`, `for` and `while` are forbidden. Their use will be flagged as cheating, no questions asked.
- Turn-in directories are `ex00/`, `ex01/`, ..., `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- Since you are allowed to use the `OCaml` syntaxes you learned about since the beginning of the piscine, you are not allowed to use any additional syntaxes, modules and libraries unless explicitly stated otherwise.
- The exercices must be done in order. The graduation will stop at the first failed exercice. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `ocamlopt`. When you are required to turn in a function, you must also include anything necessary to compile a full executable. That executable should display some tests that prove that you've done the exercise correctly.
- Remember that the special token `";;"` is only used to end an expression in the interpreter. Thus, it must never appear in any file you turn in. Regardless, the interpreter is a powerfull ally, learn to use it at its best as soon as possible!
- The subject can be modified up to 4 hours before the final turn-in time.
- In case you're wondering, no coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code he or she can't grade. As usual, big functions are a weak style.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the

exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.

- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter III

Day-specific rules

- For exercises 00, 01, 02 , 03 and 04, the `rec` keyword is forbidden and considered cheating. No questions asked. Yes, yes, I know. But still.
- For every exercise of today's subject, you are allowed to use every function in the Pervasives module. Each exercise's header specifies what you are allowed to use **aside** from these functions.
- For today et **only** for today, the keywords `for` and `while` are allowed.
- Any warning at compilation or uncaught exception means your exercise is non-functional and you will therefore earn no points for it.
- Some exercises require you to write a full program, others just specify a function. If you are just asked to write a function, you still need to turn in a full program, including a `let ()` definition with examples to show that your function is working properly. You can use any functions you want or see fit to use in your `let ()` definition, as long as you don't have to link an external library.
- If you are required to provide your own examples, remember that your tests must be comprehensive. A non-tested feature is a non-functional feature.

Chapter IV

Acknowledgements

Exercises 05, 06, 07 and 08 on Machine Learning use the Ionosphere Data Set from UC Irvine's Machine Learning Repository, and I would like to thank them for making these data public. Check the ressources to access the datasets.

Chapter V

Exercise 00: Micro-nap

	Exercise 00
A tribute to polyph...poly...you know, that polysleep thingie.	
Turn-in directory : *ex*00/	
Files to turn in : `micronap.ml`	
Allowed functions : `The Sys and Array modules`	

You will write a program which takes an integer command line argument. This argument will be the number of seconds your program will wait before exiting. Invalid or missing argument quits the program immediatly, no specific output is expected.

You must use the following function `my_sleep` to do the actual wait:

```
let my_sleep () = Unix.sleep 1
```

You will turn in this function along with your work. Feel free to sleep while your program is running, if you need to.

`You might have to do something "special" to compile this exercise.`

Chapter VI

Exercise 01: ft_ref

	Exercise : 01
A tribute to something you should normally never use.	
Turn-in directory : $ex01/$	
Files to turn in : `ft_ref.ml`	
Forbidden functions : `ref`	

Crate a type `ft_ref` to reproduce the `ref` type, and implement the following functions:

- `return: 'a -> 'a ft_ref`: creates a new reference.
- `get: 'a ft_ref -> 'a`: Dereferences a reference.
- `set: 'a ft_ref -> 'a -> unit`: Assign a reference's value.
- `bind: 'a ft_ref -> ('a -> 'b ft_ref) -> 'b ft_ref`: This one is a bit more complicated. It applies a function to a reference to transform it. You can see it as a more complicated `set` function.

The use of the standard type `ref` is obviously forbidden, but playing with it in the interpreter should tell you how it is implemented internally. Your goal is to do the same thing. Oh, and by the way, after this exercise, you will have done your first monad. Monads are some kind of ancient black magic you'll get to play with soon enough. See you on d09...

Chapter VII

Exercise 02: Bad jokes

	Exercise : 02
What's red and goes up and down?	
Turn-in directory : $ex02/$	
Files to turn in : `jokes.ml`	
Forbidden functions : `None`	

You will write a program to print a joke on the standard output, followed by an end-line character.

Your jokes can be whatever you want, but you get bonus points if they are bad. The only restriction is, you will store them in an array and there must be at least **five (5)** of them. Your program will randomly pick a joke from this array and print it to the standard output.

`A joke is bad if your grader wants to slap you after reading it.`

Chapter VIII

Exercise 03: Bad jokes, improved.

	Exercise : 03
A tomato in an elevator.	
Turn-in directory : $ex03/$	
Files to turn in : `jokes.ml, and any jokes file you want.`	
Forbidden functions : `None`	

In this exercise you will take the `jokes.ml` file from the previous exercise and make some improvements to it.

Your program will now load its jokes from a file. You can organise your file however you like, as long as anyone can modify it and add/remove jokes from it if given short instructions; if I need a 2-hour course on how to edit your file, there's obviously a problem.

The name of the jokes file will be provided to your program through a command line argument. You are not required to handle non-compliant/weird files, as your program will only be tested with your file, or any file that fully complies to your formatting rules. **IMPORTANT**: your jokes **MUST** be different from the previous exercice in order to earn bonus points!

What does not change, is that your jokes still have to be stored in an array, they still have to be bad, and your program still has to pick one randomly from that array.

Chapter IX

Exercise 04: Sum

	Exercise 04
Seriously, just a sum. There's no catch.	
Turn-in directory : $ex04/$	
Files to turn in : `sum.ml`	
Allowed functions : `None`	

You will write a function named `sum` which takes two floating-point numbers and adds one to the other. Yes. That's it.

Your function's type will be `float -> float -> float`.

`Don't forget to turn in a full program with examples to show your work is functional. This is harder than it seems.`

Chapter X

Exercise 05: You are (not) alone.

	Exercise 05
You didn't actually think I was that kind, did you?	
Turn-in directory : $ex05/$	
Files to turn in : `eu_dist.ml`	
Allowed functions : `The Array module`	

I mean come on, you know I'm much meaner than that. Now let's do some funny stuff. In the next series of exercises we'll try to do some machine learning. If you don't know what machine learning is, look it up on Wikipedia or ask your hipster entrepreneur NodeJS friend.

But first, we need to do the basic things. You will write a function named `eu_dist` which takes two points and calculates the Euclidian distance between them. If you don't know what the Euclidian distance is, here it is: if we consider a a point as an array of coordinates $a_1, a_2, a_3...a_n$ and b another point as an array of coordinates $b_1, b_2, b_3...b_n$, the Euclidian distance between a and b is:

$$eu\_dist(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

Our model for a point will be an `array` of floating-point numbers, with each cell containing the coordinate in a given dimension.

Your function's domain will be: $eu\_dist : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}^+, D \in \mathbb{N}^*$.

Your function's type will be: `float array -> float array -> float`. You don't have to handle cases with two vectors having different lengths.

Okay, now you should start to understand that machine learning is not just a buzz word. It's mostly math. And it's just the beginning. Still with me ?

Chapter XI

Exercise 06: You can (not) advance.

	Exercise : 06
This is boring. But you have to do it anyway.	
Turn-in directory : *ex*06/	
Files to turn in : `examples_of_file.ml, *.csv`	
Forbidden functions : `None`	

I said we would be doing some machine learning today; it is now time to give you some more details about what we will be doing. What we will implement is an algorithm for *supervised classification*. If you don't know what "supervised" and "classification" means, look it up on the Internet because your hipster entrepreneur NodeJS friend probably doesn't know either.

In this exercise you will write a function named `examples_of_file` which takes a path to a file as argument, and returns a set of examples read from the file input, which is formatted as csv.

Each line in the input describes a radar used to detect free electrons in the ionosphere. A radar is described by a set of `float` fields, which are some complicated stats I don't understand, and a letter at the end of the line to specify if the radar could detect evidence of free electrons in the ionosphere or not. Anyway, you just have to remember that a radar is defined by a bunch of complicated stats which form a vector, and a class under the form of a character. If you don't know what a vector is, just ask your 15-year old sibling. Chances are he or she knows.

In other words, the type of an example will be `float array * string`, and your function's type will be `string -> (float array * string) list`.

For instance, `1.0,0.5,0.3,g` will be converted to `([|1.0; 0.5 ;0.3 |], "g")`.

You can use the files named ionosphere.test.csv and/or ionosphere.train.csv for your tests, or you can use any file you want, as long as it has the same format (but it doesn't have to have the same number of float columns).

Chapter XII

Exercise 07: You can (not) redo.

	Exercise : 07
This is very interesting! But you have to do it anyway.	
Turn-in directory : $ex07/$	
Files to turn in : `one_nn.ml`	
Forbidden functions : `None`	

Here we go! Now that we have our examples, we can do some prediction. This exercise is where you're going to implement the `K-nearest` neighbours algorithm — or `K-nn` for short.

Do you remember our radars? Our radars can be either good or bad. Our objective will be to use the stats describing the radars: let's say you're trying to guess the type of a radar `A`. You know that `A` is bad. You know a radar named `B` that looks a lot like `A`. That means `B` is probably bad, right? That's the spirit of the K-nn algorithm. You pick the K nearest radars to the one you're trying to guess, and you can say `good` or `bad` depending on whether there's more good radars or bad radars.

And how do you know two radars are close to each other? DUH! You know how to compute an Euclidian distance, right? ... Right?

But right now, that sounds like a lot to do. It's complicated. And I know you're tired. So we're going to implement that with just **ONE** nearest neighbour. 1-nn. Your function will do just that: guess if the radar you give to it is good or bad, using the type of the nearest radar.

That means your function's type will be `radar list -> radar -> string`, with `type radar = float array * string`. But I bet you already figured that for yourself. You are not required to handle the case when the radars have different vector lengths, or when the train set is empty.

As usual, don't forget your tests. It could be interesting to show that your one-nn can guess correctly, but also make mistakes! Feel free to use your examples_of_file function to provide examples, or write in-memory examples if you couldn't solve the previous exercise.

Your one-nn has to be able to handle any class (not just g or b) and any vector length, as long as it's always the same for all radars.

This exercise is not mandatory.

Chapter XIII

Exercise 08: This is (not) the end.

	Exercise : 08
Let's put everything together!	
Turn-in directory : $ex08/$	
Files to turn in : `k_nn.ml`	
Forbidden functions : `None`	

Up for another challenge? Cool! :) So now we're going to really implement a `K-nn` algorithm, using what you already know, especially your `one_nn`.

The K-nn algorithm works like your `one_nn`, except you're going to pick the K nearest radars and return the class that's more represented in those K radars. In other words, you're making a generalization of `one_nn` to implement your K-nn.

By the way, K is called a `hyperparameter`, but that's just vocabulary. It'll be another argument for your function, which means its type will be `radar list -> int -> radar -> string`.

In case your K hyperparameter is even and you have a tie, do something smart. You won't be deducted if you randomly pick either choice, but really that's a shame.

`As usual, don't forget your tests. Bonus points if your tests can run a full test set and measure your K-nn classifier's accuracy.`

`This exercise is not mandatory.`

Chapter XIV

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.

`The evaluation process will happen on the computer of the evaluated group.`