Question 2: Advanced Python

(a) In what situations are normalisation and canonicalisation useful? Why?

Normalisation is a technique that adjusts values that are in different scales to a common one. This can be useful in many scenarios, in the case of a histogram, normalising is a simple way of getting the percentages of each value. In a more general way, some machine learning algorithms work more efficiently if the training data has been previously normalised (e.g. for distance-based models such as K-NN neighbours).

Canonicalisation is a technique that groups values that represent the same concept to a common one. This helps humans and machines able to identify multiple different values from a database as the same one. This is important in many scenarios, in particular, when training data to a model we want to make sure that the data has been canonicalised before so the model has the right interpetation of the values.

(b) We have studied time complexity, this is how the time to run a piece of code will grow as the size of the input grows. Another interesting topic is space complexity, that is the amount of memory required by a piece of code will grow as the size grows. What do you think is the space complexity of the following code?

```
def product_matrix(L):
    """
    L is a list of numbers of length n"""
    M = []
    n = len(L)
    for i in range(n):
        M.append([])
        for j in range(n):
            M[i].append(L[i]*L[j])
    return M
```

~~Mycroft~~ At the end of the execution, M is a list of list that has length n. Each element of n is a list that has n elements.

Therefore, the space complexity of this function would be $n \cdot n = n^2$, $\mathcal{O}(n^2)$.

- Question 2

(c) Explain and distinguish between all the uses of * and ** operators in Python.

Operators * can be used as:

- product between two numbers, e.g. 3*8 = 24
- pack values in a tuple; e.g., a, *b = 1,2,3,4
- unpack values, e.g. my-set = {1,2,3}; {0, *my-set, 4}
- handle a non-determine number of arguments in a function, e.g.

```
def sum(*args):
    sum=0
    for n in args:
        sum = sum+n
    return n

sum(1,2), sum(1,4,5), ...
```

Operator ** can be used as:

- exponential operation, e.g. 2**3 = 8
- handle a non-determine number of arguments with ~~specific~~ keys:

```
def sum(**kwargs):
    ==

sum(a=1, b=2), sum(a=1, b=2, c=3)
```

(d) Explain and distinguish between ~~the uses of * and ** operators in Python~~ the @ symbol as used in (modern) Numpy and as used in plain Python

In modern Numpy, @ works as a multiplication of two matrices. This is equivalent to a@b ~ a.__matmult(b).

In plain Python, @ works as a decorator. Example:

```
@decorator
def decorator_function():
    ____
```

would be equivalent to decorator(decorator_function)

• Question 3:

(a) In Scikit-Learn, what happens if we run m.predict() for a model m which a has not yet been fit()ed? How does Scikit-Learn know whether it has been fit()ed?

Using predict on an unfitted model will trigger an error saying that that model has never been fitted. Scikit-Learn uses 'check_is_fitted' which verifies if the estimator/model is fitted by verifying the presence of fitted attributes.

(b) Suppose we have a Pandas DataFrame d with many columns. How can we extract from d a DataFrame with just the columns a and b? How can we extract just column as a 1D Numpy array?

d[['a','b']] will extract just columns a and b.

nparray(d['column']) will extract just that column as a 1D Np array.

(c) Give an example of a real-world application where we might use 5D

In a video application

(d) What data format could we use to describe Santa's trajectory on Christmas Eve? What techniques could we use to extract useful features from this data, e.g. features that would be useful to distinguish his trajectory during city visits from his trajectory in the countryside? Assume that after leaving any house, he usually visits the nearest unvisited house next.

We could used the coordinates of Santa's location in every house, such as latitude and longitude. If having the coordinates of the different cities that santa would be visiting, the distance from the current position to each of the cities would give us an idea of where he is, e.g., using the mayor of the city as a reference.

• Question 4: Tools and Applications

(a) Suppose we want to generate code. We could try choosing a random symbol, e.g. for, while, True, 0, <, one at a time, and then concatenating them. Will this work and why/why not? What alternatives are there.

It would not work as some sequence of these symbols are not syntaxily correct ~~in Python, eg~~ as a pseudo-code, e.g., 'for while <' would not make sense. In order to generate correct code, ~~we could use~~ a grammar could be used to make sure that the syntax is correct.

~~Example of grammar for this task:~~

~~<loop> ::= <variable> <for> <operator>~~

(b) In the context of formal languages, define language and sentence. What does it mean to say that a string is not a valid sentence.

A formal language L over an alphabet Σ is a subset of Σ^*, that is, a set of words over the alphabet. In a more general way, a formal language involves a context free grammar. This is a 4 tuple $G = (V, \Sigma, R, S)$ where:

1) V is a finite set where each element $v \in V$ is called a non-terminal character or variable.

2) Σ is a finite set of terminals, disjoint from V.

3) R is a relation in $V \times (V \cup \Sigma)^*$

4) S is the start variable, used to represent the whole sentence.

A string that is not a valid sentence would mean that, either this string has words that are not in Σ or that there is no combination of relationships in R that could represent these sentence as S.

Question 4

(c) Describe an approach to programming a bot to play tic-tac-toe. It should attempt to learn from the other player history. There is no need to provide complete code.

Each position can be represented by a, e.g., list with 9 elements. We can use always '0' to represent the move of the other player and change this in history if needed to be consistent.

Each game can be represented by a list of length n where n is the number of rounds in that game. ~~The last one has the pos~~

The list has positions as elements in such a way that the i-th element of the list represents the final position after round i.

The board can represent by -1 an empty cell, '0' player position and 'X' the agent.

The games can be divided by result and using a counter we can have a dictionary that has as keys a string with a position and as values the number of occurrence of that position in the category.

Let's see an example, we will have three dict d_player_won, d_draw, d_player_lost.

d_player_won = {"[0,X,-1,-1,-1,-1,-1,-1,-1]": 32, "[0,0,-1,X,-1,X,-1,-1,-1]": 21, ...}

↑ this position occurred 32 times in games where the player won.

Now, Ideally we want the agent to win, otherwise to draw and finally lose.

We can assign an score to each position: ~~score~~

score(position) = d_player_lost + d_player_draw · 0.5 − d_player_won

In this way, we know we want the agent to perform a positive move. The agent should first attempt the position with the highest score, otherwise the second one, etc. If no positive position can be achieved, the agent should then attend any position ~~that~~ (random) that doesn't have a negative score.