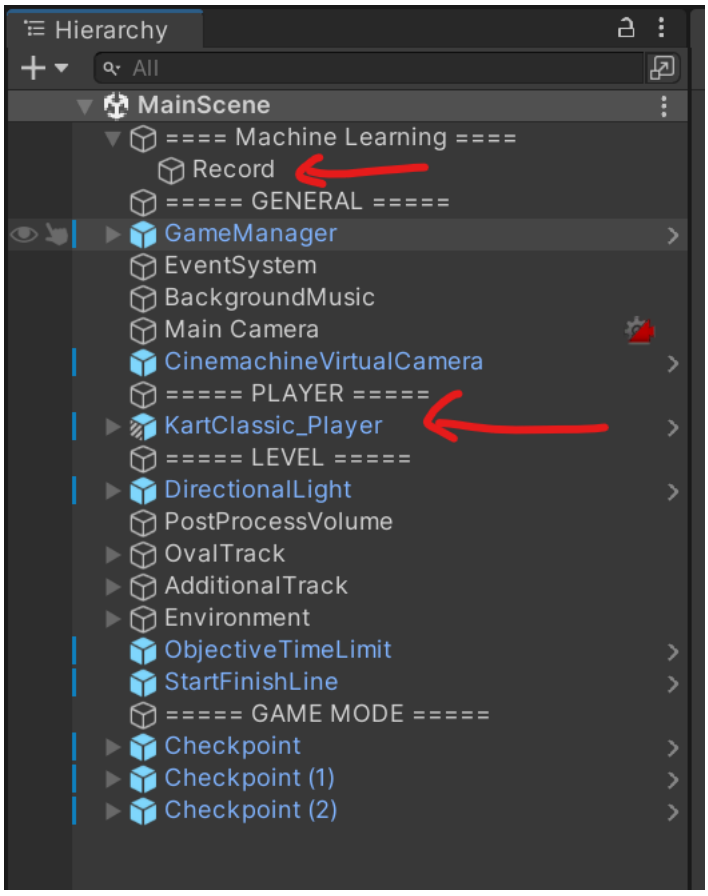
Practica sobre Machine Learning

Pertenecemos a un estudio de videojuegos que pretende desarrollar un juego de Karts similar a Super Mario Kart. Para ello se ha desarrollado un prototipo jugable que teneis a vuestro disposición en el campus virtual.

Vuestro cometido es desarrollar una IA para los coches controlados por el juego. Para ello, vamos a entrenar a un agente jugando al juego multiples veces. Una vez tengamos guardado los datos de ejemplo, nuestro cometido es realizar un estudio teórico sobre que modelo de Machine Learning es más eficiente para implementar en el agente.

Ejercicio 1 Juega al juego varias veces:

En el proyecto de Unity adjunto, cargando **MainScene en (Assets/Karting/Scenes)** podemos ver la siguiente jerarquia:



El GameObject KartClassic_Player tiene la implementación del control del

coche. Se ha añadido dos componentes, Perception y MLAgent:

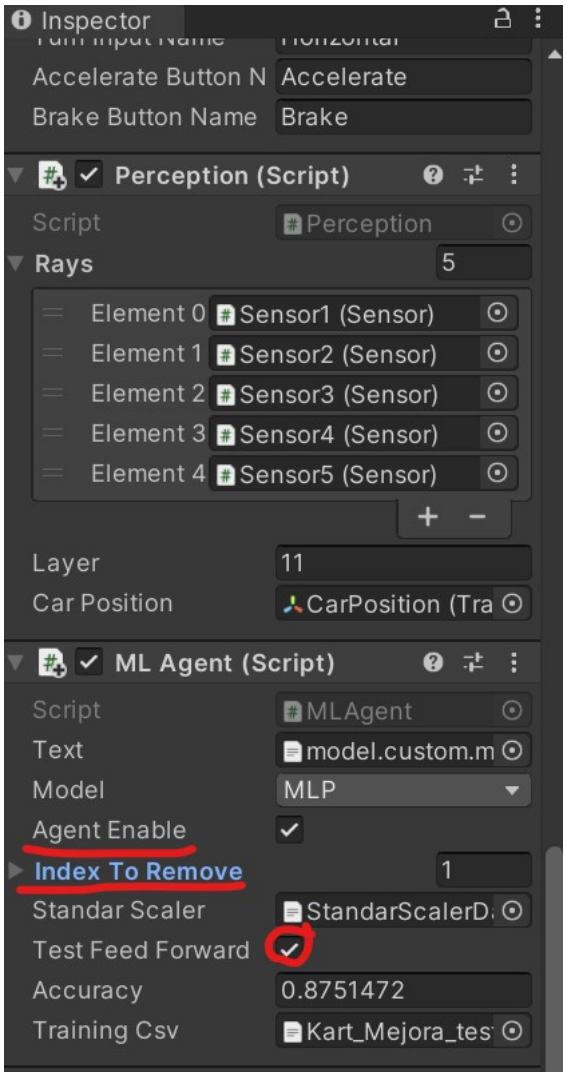
- Perception: es una clase que implementa la percepción del agente. Podéis modificarla todo lo que queráis para poder añadir más información al estado del juego. Pero por defecto puede generar un número de rayos fisicos indicados en el campo Rays. Rays admite el componente Sensor como entrada. Si desplegáis KartClassic_Player podéis ver que existe dos objetos, Sensors y CarPosition. Sensors es la lista de sensores disponibles y CarPosition la posición del coche.

La percepción genera un rayo que parte desde CarPosition a cada uno de los sensores. Dichos sensores los podéis colocar donde queráis, o incluso podéis añadir nuevos, pero os recomiendo que estén a la misma altura que CarPosition para evitar que se eleven y no impacten en el borde de la carretera.

Por defecto la codificación de la clase pone con el valor de -1 el rayo que impacta en la carretera y la distancia del impacto en caso de que se produzca. **Esta codificación puede ser modificada si lo considerais adecuado.**

- MLAgent: nos permite implementar un modelo de Machine Learning. Está preparado para poder implementar varios modelos, pero por defecto sólo implementa parcialmente el MLPClassifier de SkLearn. Cuando esta activo el check **AgentEnable**, el player leerá el imput del MLAgent y no del input normal (Teclado). En text tenemos el asset con el modelo serializado del MLPClassifier previamente generado desde Python.

Tendréis que implementar el feedforward, para probarlo activar el campo Test Feed Forward que usando como prueba los datos de entrenamiento, calcula el accuracy del modelo. Debería dar un resultado similar al accuray del modelo que implementa SKLearn usando todos los datos para calcular la predicción. En mi ejemplo conseguí un 87.5% de accuracy y en la implementación en C# obtuve un 88% esto puede ser normal ya que no sabemos cómo esta implementado MLPClassifier en SKLearn, pero el resultado debe ser similar. En **Index to remove** podemos quitar los índices del input que hayamos limpiado en Python porque hayamos considerado que no son relevantes para el juego después de haber hecho el estudio previo.



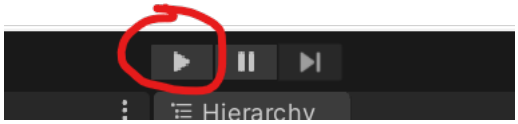
El GameObject Record nos permite grabar una partida. Si tenemos el check RecordMode activo al iniciar una partida se guarda la ejecución. El nombre del fichero se indica en Csv Output y se guarda automáticamente si tienes activo el check Save When Finish. También puedes iniciar la grabación y finalizarla pulsando la tecla R.

Perception Names es el nombre que le damos a los campos de los rayos. Además de estos guardamos: - kartx: posición x del Kart. - karty: posición y del Kart. - kartz: posición z del Kart. - time: tiempo en segundos donde se toma la captura de datos. - action: acción que ha realizado el jugador.

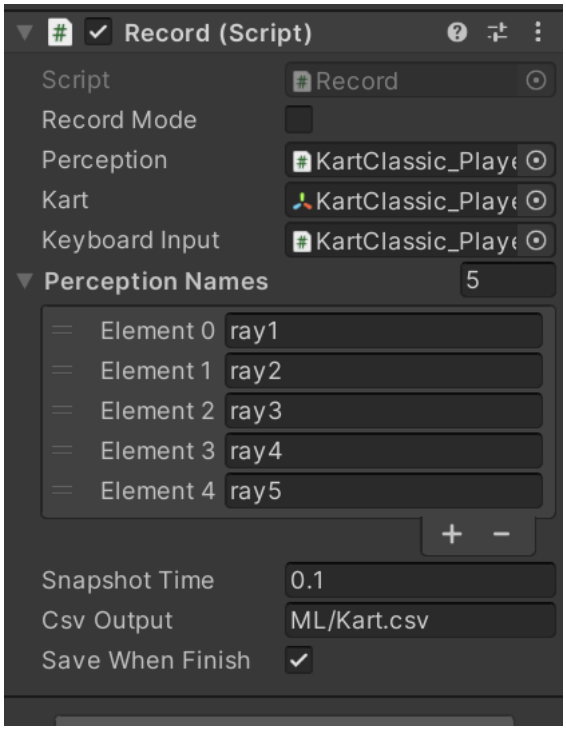
Las acciones posibles son:

```
public enum Labels { NONE=0, ACCELERATE=1, BRAKE=2, LEFT_ACCELERATE=3,
 RIGHT_ACCELERATE=4, LEFT_BRAKE=5, RIGHT_BRAKE=6 }
```

La idea es jugar varias veces con el record activado y con el **AgenEnable** desactivado y almacenar las trazas. Para ello basta con cargar MainScene y darle al play.



Con el RecordMode de Record activo (ten cuidado no machacar los ficheros de partidas previas, cambia el nombre en cada ejecución)



Snapshot Time es el tiempo en segundos que pasa entre cada captura de datos. Puedes jugar con el para generar más datos o menos en función de tus necesidades. **Por defecto, el fichero se sobrescribe, asi que recuerda o cambiar el nombre o guardar los datos entre partidas.** También puedes implementar un guardado incremental, pero esta funcionalidad no está añadida.

Ejercicio 2 visualiza los datos (1 punto):

Crea un CSV con todas las trazas del juego que hayas generado y llévatelo a Python / Jupiter Notebook.

Ahí dibuja la distribución de clases que hayáis grabado. Ten cuidado porque tendrá seguramente más de 3 atributos y no podréis visualizarlo directamente. Consulta como hacerlo en el Tema 6 de Data Mining.

Ejercicio 3 Limpia el dataset (1 punto):

Crea una versión del dataset limpia con la información que sea relevante tanto en atributos usados como en acciones. Comprueba que no haya valores erroneos o que no tengan sentido y corrígelos.

También en esta sección debes normalizar los datos usando StandarScaler.

Ejercicio 4 Prueba diferentes modelos de Machine Learning (hasta 6 puntos):

- Al menos uno de ellos debe ser vuestra implementación del perceptrón multicapa. El perceptrón multicapa debe permitir crear modelos de cualquier número de capas. Al menos en el jupiter notebook a entregar debe haber un ejemplo con más de 3 capas para poder contar con el punto asociado con esta característica, independientemente que el mejor modelo resulta ser con una sola capa oculta. **(1 punto)**
- Comprueba que los resultados de tu implementación son similares (que no idénticos) al MLPClassifier de SKLearn (poner los mismos valores de alpha, learning_rate_init y de **usar la función logística o sigmoidal para toda la red**) El resultado de validación debe ser superior al 70% par obtener el punto **(1 punto)**
- A parte de esta versión, podéis cacharrear con diferentes parámetros de la versión de SKLearn del MLPClassifier (con función de activación relu o con diferentes versiones del optimizador) hasta conseguir un buen resultad de validación. El resultado de validación debe ser superior al 70% para obtener el punto **(1 punto)**
- Crea un modelo KNN y comprueba el resultado que has obtenido. Intenta que el modelo tenga un resultado lo mas similar o superior al perceptrón que puedas. **(1 punto)**
- Crea un modelo de árboles de decisión y otro de Random Forest y comprueba el resultado que has obtenido. Intenta que los modelos tengan un resultado lo mas similar o superior al perceptrón que puedas. **(1 punto)**
- Muestra las matrices de confusión de todos los modelos usados, también calcula accuracy y otras métricas para todos los modelos y al final en

el propio Notebook usando Markdown explica que modelo crees que se adapta mejor al juego y cual elegirías. **(1 punto)**

- Itera entre los modelos, sus hiperparámetros, los datos exportados y la limpieza de los mismos, así como el número de ejemplos de entrenamiento hasta conseguir modelos con el mejor rendimiento teórico posible.

Ejercicio 5 Implementa el perceptrón multicapa que quieras en Unity (2 puntos).

Puedes elegir vuestro modelo o el modelo de MLPClassifier de SKLearn. Por defecto viene la fontanería necesaria para poder implementar el modelo de MLPClassifier en Unity, asi que te recomiendo que uses ese, pero si te animas puedes usar el tuyo. Para poder exportar dicho modelo a diferentes formatos podéis usar el método **Utils.ExportAllformatsMLPSKlearn** que os exporta el modelo a diferentes formatos: pickle, onnx, JSON y un formato custom que os facilita poder exportar vuestro modelo si así lo consideráis. Unity utiliza esta versión custom para cargar el modelo. **NOTA exportad el modelo con la función logistica como función de activación**

El formato custom se comporta de la siguiente forma (cada campo es una línea):

```
num_layers:4
parameter:0
dims:['9', '8']
name:coefficient
values:[4.485004762391437, 0.7937380313502955, ... -1.360755118312314]
parameter:0
dims:['1', '8']
name:intercepts
values:[-0.6982858709618917, 3.3853548406875134,... 0.1310577892518341]
--- repetir por cada capa / theta, cambiando el número de parameter y coefficient e intercep
```

- num_layer: nos indica le número de capas del perceptrón.
- dims: es la dimensión de la matriz
- name: es el nombre del coeficiente. Las matrices theta se llaman coefficient y los sesgos intercepts. Este es el formato por defecto, pero podéis cambiarlo si vuestro código introduce los sesgos dentro de las matrices, aunque tendréis que cambiar código en Unity.

La implementación actual lee correctamente los parámetros de MLPClassifier de SKLearn. Para conseguir los puntos de este ejercicio mínimo habría que usar esa implementación, pero se valorará usar vuestra propia implementación en el agente. Como podéis ver, SKLearn implementa los sesgos como un vector aparte y no dentro de la matriz de pesos.

Para implementar el agente hay que escribir las siguientes funciones en C# en el componente MLAgent

```
// TODO Implement FeedForward
public float[] FeedForward(float[] a_input)
{
    throw new NotImplementedException();
}
```

Y la función ConvertIndexToLabel ya que según como juegues puede que no uses todas las clases y por tanto el orden de las mismas en el MLP será diferente.

```
//TODO: implement the conversion from index to actions. You may need to implement sever
//transforming the data if you play in different ways. You must take into account how m
//you have used, and how One Hot Encoder has encoded them and this may vary if you chan
//data.
public Labels ConvertIndexToLabel(int index)
{

    throw new NotImplementedException();
}
```

También debéis implementar la función Transform de la clase StandarScaler. Esto lo necesitáis para transformar los datos de la misma forma que lo transformáis en Python. Para poder calcular las medias de los datos, desde python podéis usar WriteStandardScaler en la clase Utils. que genera un fichero con los valores de las medias y las varianzas que calcula StandarScaler de SKLearn. Ese fichero lo debéis asignar en el campo StadarScaler del MLAgent.

```
// TODO Implement the standar scaler.
public float[] Transform(float[] a_input)
{
    throw new NotImplementedException();
}
```

Puedes necesitar tocar algo de código adicional para implementar el MLP aunque hemos intentado que sea el menor posible. En cualquier caso, siéntete libre de cambiar lo que necesites.

Lo ideal es conseguir un modelo que permita al Kart llegar a la meta y que sea razonablemente similar a nuestra forma de jugar, pero **si no se consigue no pasa nada debido a las limitaciones de tiempo que disponemos.**

Ejercicio 6 Implementa otro ejecutar de un modelo de ML del os que hayas usado en Unity (opcional + 1 punto)

Realizar este apartado aporta **un punto extra** a la nota de la práctica. Si la práctica de por si tiene un 10 o un 9 y pico, el resto de la puntuación se sumará a la puntación del examen.

Puedes elegir cualquier otro ejecutor de modelo que quieras, Decision Tree, KNN

o modelos más complejos con deep learning. Si vas a utilizar modelos mas complejos te recomendamos usar ML-Agents o Sentis, pero si lo que quieres es implementar algo sencillo, puedes hacerlo con modelos como KNN o decisión tree donde crear un ejecutor es algo más o menos fácil de hacer.

Si realizas este apartado, explica en el Notebook si ha funcionado mejor en la práctica que el modelo de MLP que hayáis implementado en el ejercicio anterior. La implementación obviamente debe conseguir un modelo razonablemente bueno para que cuente el punto extra.

English Version

Practice on Machine Learning

We belong to a videogame studio that aims to develop a karting game similar to Super Mario Kart. We have developed a playable prototype that you have at your disposal in the virtual campus.

Your task is to develop an AI for the cars controlled by the game. To do this, we will train an agent by playing the game multiple times. Once we have saved the example data, our task is to carry out a theoretical study on which Machine Learning model is the most efficient to implement in the agent.

Exercise 1 Play the game several times:

In the attached Unity project, by loading **MainScene in (Assets/Karting/Scenes)** we can see the following hierarchy:

.

The GameObject KartClassic_Player has the car control implementation. Two components have been added, Perception and MLAgent:

- Perception: is a class that implements the perception of the agent. You can modify it as much as you want to be able to add more information to the game state. But by default it can generate a number of physical rays indicated in the Rays field. Rays supports the Sensor component as input. If you deploy KartClassic_Player you can see that there are two objects, Sensors and CarPosition. Sensors is the list of available sensors and CarPosition is the position of the car.

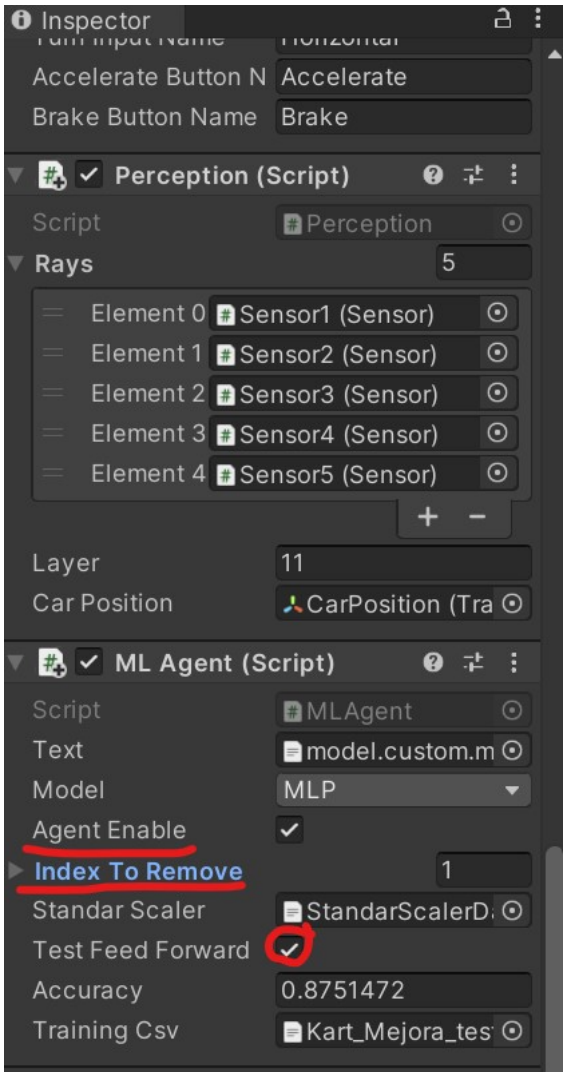
**The perception generates a beam that starts from CarPosition to each of the sensors. These sensors can be placed wherever you want, or you can even add new ones, but I recommend that they are at the same height as CarPosition to avoid that they rise and do not impact on the edge of the road.

By default the coding of the class puts with the value of -1 the ray that hits the road and the distance of the impact in case it occurs. **This coding can be

modified if you consider it appropriate.

- MLAgent: allows us to implement a Machine Learning model. It is prepared to implement several models, but by default it only partially implements the MLPClassifier of SkLearn. When the **AgentEnable** check is active, the player will read the imput from the MLAgent and not from the normal input (Keyboard). In text we have the asset with the serialized model of the MLPClassifier previously generated from Python.

You will have to implement the feedforward, to test it activate the Test Feed Forward field that using as test the training data, calculates the accuracy of the model. It should give a result similar to the accuray of the model implemented by SKLearn using all the data to calculate the prediction. In my example I got 87.5% accuracy and in the C# implementation I got 88%. This may be normal since we don't know how MLPClassifier is implemented in SKLearn, but the result should be similar. In **Index to remove** we can remove the indexes of the input that we have cleaned in Python because we have considered that they are not relevant for the game after having done the previous study.



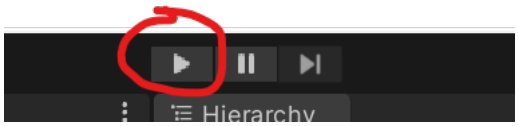
The GameObject Record allows us to record a game. If we have the Record-Mode check active when starting a game, the execution is saved. The file name is indicated in Csv Output and is automatically saved if you have the Save When Finish check active. You can also start the recording and end it by pressing the R key.

Perception Names is the name we give to the ray fields. In addition to these we save: - kartx: x position of the Kart. - karty: position y of the Kart. - kartz: position z of the Kart. - time: time in seconds where the data capture is taken. - action: action performed by the player.

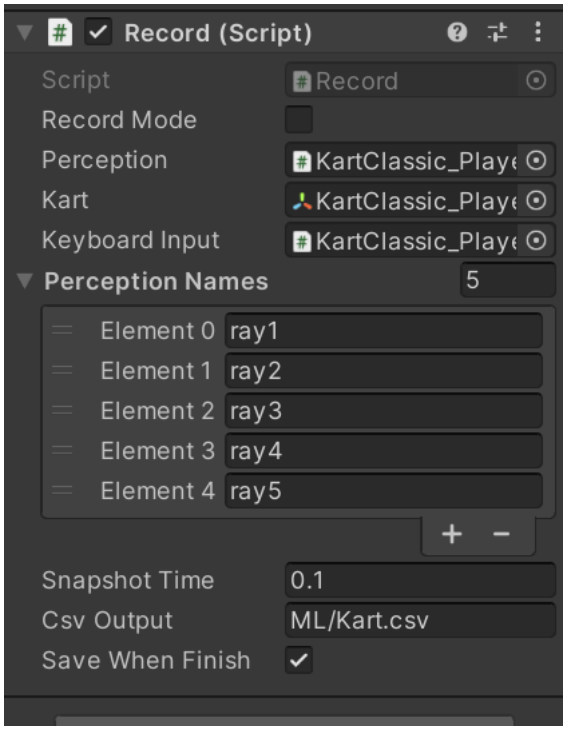
The possible actions are:

```
public enum Labels { NONE=0, ACCELERATE=1, BRAKE=2, LEFT_ACCELERATE=3,
 RIGHT_ACCELERATE=4, LEFT_BRAKE=5, RIGHT_BRAKE=6 }
```

The idea is to play several times with the record activated and with **AgenEnable** deactivated and store the traces. To do this, just load MainScene and press play.



With the RecordMode of Record active (be careful not to crush the files of previous games, change the name at each run)



Snapshot Time is the time in seconds that elapses between each data capture. You can play with it to generate more or less data depending on your needs. **By default, the file is overwritten, so remember to either rename or save the data between games.** You can also implement an incremental save, but this functionality is not added.

Exercise 2 visualize the data (1 point):

Create a CSV with all the game traces you have generated and take it to Python / Jupiter Notebook.

There draw the distribution of classes that you have recorded. Be careful because it will probably have more than 3 attributes and you will not be able to visualize it directly. See how to do it in Data Mining Topic 6.

Exercise 3 Clean up the dataset (1 point):

Create a clean version of the dataset with the information that is relevant in both attributes used and actions. Check that there are no erroneous or meaningless values and correct them.

Also in this section you should normalize the data using StandarScaler.

Exercise 4 Try different Machine Learning models (up to 6 points):

- At least one of them must be your implementation of the multilayer perceptron. The multilayer perceptron should allow you to create models of any number of layers. At least in the jupiter notebook to be delivered there must be an example with more than 3 layers to count for the point associated with this feature, regardless that the best model turns out to be with only one hidden layer. (1 point) **(1 point)**.
- Check that the results of your implementation are similar (but not identical) to the SKLearn MLPClassifier (set the same values of alpha, learning_rate_init and **use the logistic or sigmoidal function for the whole network**). The validation result must be higher than 70% to get the point **(1 point)**.
- Apart from this version, you can play with different parameters of the SKLearn version of the MLPClassifier (with relu activation function or with different versions of the optimizer) until you get a good validation result. The validation result must be higher than 75% to get the **(1 point)** point.
- Create a KNN model and check the result you have obtained. Try to make the model as similar or superior to the perceptron as possible. (1 point) **(1 point)**.
- Create a decision tree model and a Random Forest model and check the result you have obtained. Try to make the models have a result as similar or superior to the perceptron as you can. (1 point) **(1 point)**.
- It shows the confusion matrices of all the models used, also calculates accuracy and other metrics for all the models and at the end in the Notebook itself using Markdown explains which model you think fits the game best and which one you would choose. (1 point)**(1 point)**.
- Iterate between the models, their hyperparameters, the exported data and data cleaning, as well as the number of training examples until you get

models with the best possible theoretical performance.

Exercise 5 Implement the multilayer perceptron of your choice in Unity (2 points).

You can choose your model or the MLPClassifier model from SKLearn. By default comes the necessary plumbing to be able to implement the MLPClassifier model in Unity, so I recommend you to use that one, but if you feel like it you can use your own. To export the model to different formats you can use the **Utils.ExportAllformatsMLPSKlearn** method that exports the model to different formats: pickle, onnx, JSON and a custom format that makes it easier to export your model if you consider it. Unity uses this custom version to load the model.

The custom format behaves as follows (each field is a line):

“ num_layers:4 parameter:0 dims:[‘9’, ‘8’] name:coefficient values:[4.485004762391437, 0.7937380313502955, ... -1.360755118312314] parameter:0 dims:[‘1’, ‘8’] name:intercepts values:[-0.6982858709618917, 3.3853548406875134,... 0.1310577892518341] — repeat for each layer / theta, changing the number of parameter and coefficient and intercepts will have an associated number e.g. coefficient1, coefficient2.... intercepts1, intercepts2—

```
- num_layer: indicates the number of layers of the perceptron.
- dims: is the dimension of the matrix
- name: is the name of the coefficient. The theta matrices are called coefficient and the bi

The current implementation correctly reads the MLPClassifier parameters from SKLearn. To get

To implement the agent you have to write the following C# functions in the MLAgent component

```C#

    // TODO Implement FeedForward
    public float[] FeedForward(float[] a_input)
    {
        throw new NotImplementedException();
    }
```

And the ConvertIndexToLabel function because depending on how you play you may not use all the classes and therefore the order of the classes in the MLP will be different.

```
    //TODO: implement the conversion from index to actions. You may need to implement sever
    //transforming the data if you play in different ways. You must take into account how m
    //you have used, and how One Hot Encoder has encoded them and this may vary if you chan
    //data.
```
```

```
public Labels ConvertIndexToLabel(int index)
{

    throw new NotImplementedException();
}
```

You must also implement the Transform function of the StandarScaler class. You need this to transform the data in the same way you transform it in Python. In order to calculate the means of the data, from Python you can use WriteStandardScaler in the Utils class, which generates a file with the values of the means and variances that StandarScaler calculates from SKLearn. This file must be assigned in the StadarScaler field of the MLAgent.

```
// TODO Implement the standar scaler.
public float[] Transform(float[] a_input)
{
    throw new NotImplementedException();
}
```

You may need to touch some additional code to implement the MLP although we have tried to keep it as little as possible. In any case, feel free to change whatever you need.

Ideally we would like to get a model that allows the Kart to reach the goal and that is reasonably similar to the way we play, but **if this is not achieved it is ok due to the time constraints we have.

Exercise 6 Implement another run of a ML model of the ones you have used in Unity (optional + 1 point)

Performing this section adds **one extra point** to the practice grade. If the practice itself has a 10 or a 9 something, the rest of the score will be added to the exam score.

You can choose any other model runner you want, Decision Tree, KNN or more complex models with deep learning. If you are going to use more complex models we recommend you to use ML-Agents or Sentis, but if you want to implement something simple, you can do it with models like KNN or decision tree where creating an executor is more or less easy to do.

If you do this section, explain in the Notebook if it has worked better in practice than the MLP model you have implemented in the previous exercise. The implementation must obviously achieve a reasonably good model for the extra point to count.