



Bachelor's Degree in Telecommunications Engineering

Bachelor Thesis

**Text Games Synthetic Data Generation based on
Behavior Profiles**

Author

Vicente Balmaseda del Campo

Supervised by

Gregorio Ignacio López López

Jaime Pérez Sánchez

Madrid

July 2022

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Text Games Synthetic Data Generation based in Behavior Profiles

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el
curso académico **2021-2022** es de mi autoría, original e inédito y
no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha
sido tomada de otros documentos está debidamente referenciada.

Firmado por BALMASEDA DEL CAMPO
VICENTE - 05335491C el día
06/07/2022 con un certificado
emitido por AC ENMT Usuarios

Fdo.: Vicente Balmaseda del Campo

Fecha: 06/07/2022

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

LOPEZ LOPEZ
GREGORIO IGNACIO
- 75887558R

Digitally signed by
LOPEZ LOPEZ GREGORIO
IGNACIO - 75887558R
Date: 2022.07.06
20:16:24 +02'00'

Fdo.: Gregorio Ignacio López López

Fecha: 06/07/2022

PEREZ SANCHEZ
JAIME -
03138646C

Firmado digitalmente por
PEREZ SANCHEZ JAIME -
03138646C
Fecha: 2022.07.06 19:39:07
+02'00'

Fdo.: Jaime Pérez Sánchez

Fecha: 06/07/2022



Bachelor's Degree in Telecommunications Engineering

Bachelor Thesis

**Text Games Synthetic Data Generation based on
Behavior Profiles**

Author

Vicente Balmaseda del Campo

Supervised by

Gregorio Ignacio López López

Jaime Pérez Sánchez

Madrid

July 2022

GENERACIÓN DE DATOS SINTÉTICOS EN JUEGOS DE TEXTO BASADO EN PERFILES DE COMPORTAMIENTO

Autor: Balmaseda del Campo, Vicente.

Director: López López, Gregorio Ignacio.

Director: Pérez Sánchez, Jaime.

Entidad Colaboradora: ICAI — Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

La generación de datos es una técnica que se ha utilizado durante mucho tiempo para hacer frente a la falta de datos y mejorar los modelos entrenados con ellos. Sin embargo, la mayor parte de las técnicas de generación de datos para juegos o bien requieren datos preexistentes, lo cual suele provocar una baja variabilidad y un sesgo hacia dichos datos; o bien se centran en obtener la mayor puntuación posible, lo que no es adecuado para juegos centrados en la toma de decisiones o en los que las puntuaciones no son aplicables, como los juegos serios. Este trabajo presenta una arquitectura para generar datos sintéticos en juegos de texto abordando dichas limitaciones. Para ello se crea un conjunto de agentes que imitan el comportamiento de jugadores reales y se recolecta datos mientras juegan. Por lo tanto, no se requieren datos preexistentes. Se asigna a cada agente un perfil de comportamiento en relación con una serie de rasgos de carácter y los agentes seleccionan las acciones que mejor se ajustan a dichos rasgos, teniendo en cuenta también las acciones pasadas. Además, esta arquitectura incorpora dicha información de comportamiento en los datos generados y, por lo tanto, en los modelos entrenados con ellos. Ilustramos la aplicación de esta arquitectura implementándola en un juego serio de texto que pretende educar sobre los ciberdelitos, en particular el *Online Grooming*. Generamos un conjunto de datos sintéticos para el juego y entrenamos varios modelos de Machine Learning en ellos. Los datos generados logran enseñar a estos modelos a, dadas las decisiones tomadas en una ejecución del juego, predecir el perfil de comportamiento que la generó. A continuación, estos modelos pueden utilizarse para encontrar el grado de similitud del comportamiento de los jugadores reales con respecto a los perfiles dados.

Palabras clave: análisis del comportamiento, aprendizaje automático, generación de datos sintéticos, inteligencia artificial, juegos serios (basados en texto), procesamiento del lenguaje natural

1. Introducción

Los juegos de texto son juegos de un solo jugador por turnos en los que en cada turno, se presenta al jugador una descripción del contexto y una lista de acciones disponibles, ambas dadas por su representación textual en lenguaje natural. El jugador elige una acción que cambia el estado del juego, produciendo un nuevo contexto y un nuevo conjunto de acciones. El juego termina cuando el jugador alcanza un estado terminal. Por lo tanto, los jugadores tratan de encontrar una política óptima que les proporcione el mejor resultado final.

Los juegos de texto tienen otras aplicaciones además del entretenimiento. Algunas de estas aplicaciones tienen fines educativos, de análisis del comportamiento o incluso de investigación. El proyecto europeo H2020 [RAYUELA](#), en el que se enmarca este trabajo, está desarrollando un juego serio interactivo similar a una historia cuyo objetivo es doble: por un lado, permitirá a los menores aprender buenas prácticas en línea mientras juegan; y, por otro lado, los datos recogidos a través del juego se analizarán para tratar de identificar posibles factores de riesgo de modo que las Fuerzas y Cuerpos de Seguridad puedan desarrollar políticas basadas en pruebas para proteger a los menores en línea. Estas aplicaciones aprovechan la capacidad de entretenimiento del juego para recoger datos que luego se introducen en un modelo. Sin embargo, los datos pueden no estar disponibles en ese momento, ser insuficientes o incluso inobtenibles.

Esta falta de datos se puede paliar generando datos sintéticos para ampliar el conjunto de datos y utilizarlos como sustituto temporal o permanente de los datos reales. Las técnicas existentes generan datos aprovechando datos preexistentes (o información sobre ellos como su distribución) o mediante agentes. Sin embargo, ambas tienen inconvenientes. Las técnicas basadas en datos preexistentes sufren de sesgo hacia los datos de entrenamiento y tienden a generar datos sintéticos con baja variabilidad. Por otro lado, las técnicas basadas en agentes utilizan recompensas para obtener el mejor resultado final. Esto hace que estas técnicas sean inadecuadas para juegos centrados en la toma de decisiones en lugar de en ganar o en los que las recompensas no son aplicables, como los juegos serios. Dado que el foco está en la toma de decisiones, los agentes sintéticos utilizados para la generación de datos deben comportarse como lo harían los jugadores, lo que implica modelar estos agentes de acuerdo con el comportamiento de los jugadores reales. Por lo tanto, la generación de datos para juegos serios requiere un nuevo enfoque basado en los perfiles de comportamiento de los jugadores reales. Además, este enfoque debe abordar también la dificultad del Procesamiento del Lenguaje Natural (*Natural Language Processing*, NLP) que conlleva el texto en lenguaje natural.

Este trabajo propone un modelo para generar datos sintéticos en los juegos, especialmente en los juegos serios, en los que la atención se centra en la toma de decisiones y no en la obtención de la máxima puntuación o en los que no se aplican recompensas.

Dado que la mayoría de los juegos de toma de decisiones pueden convertirse en juegos de texto, nos centraremos en ellos. Sin embargo, nuestro modelo también puede tratar con otros tipos de información, como imágenes o audio.

El modelo presentado aborda las dificultades expuestas cumpliendo los siguientes requisitos:

- **Considera los perfiles de comportamiento** con respecto a un conjunto de rasgos de carácter¹.
- **Independiente de los datos existentes**
- **Reutilizabilidad**
- **Entrenamiento supervisado de los perfiles de comportamiento.** A través de este proceso, la información proporcionada por los perfiles de comportamiento puede incorporarse a los modelos y utilizarse para detectarlos o analizarlos en datos reales.
- **Segregación de tareas** en historia, NLP y política de los agentes (perfil de comportamiento)

El resto de este trabajo está estructurado de la siguiente manera. En la Sec. 2 se presenta la arquitectura con sus componentes e interacciones, y la funcionalidad de cada componente. A continuación, se ilustra esta arquitectura en la Sec. 3 implementándola en un juego serio de texto sobre *Online Grooming*. Generamos un conjunto de datos sintéticos para el juego y utilizamos estos datos para entrenar varios modelos de aprendizaje automático en la clasificación de cada ejecución del juego por el perfil de comportamiento que lo generó. También ilustramos cómo estos modelos pueden ser utilizados en datos reales para encontrar el grado de similitud entre el comportamiento de los jugadores y los perfiles de comportamiento dados. Las conclusiones y el trabajo futuro se pueden encontrar al final en la Sec. 4.

2. Descripción de la Arquitectura

La idea principal del modelo propuesto es utilizar un conjunto de perfiles de comportamiento para crear agentes sintéticos orientados por políticas que jugarán el juego. Mientras los agentes juegan, se recogerán datos sintéticos como si fueran jugadores reales. De este modo, generamos datos con agentes que implementan los perfiles de comportamiento y sin utilizar datos preexistentes.

El modelo presentado se divide en dos componentes principales, el simulador o entorno, y el agente o jugador. Cada uno de estos componentes es independiente e inter-

¹Por rasgos de carácter nos referimos al conjunto de rasgos de comportamiento o cualidades que definen el tipo de persona. Activo, tranquilo, cuidadoso y atrevido son algunos ejemplos de rasgos de carácter.

actúa con el otro a través de una interfaz. Este enfoque modular facilita la colaboración interdisciplinar al separar las subtarefas de la historia, NLP y el modelado del perfil de comportamiento en componentes independientes. El entorno se encarga de la historia del juego, mientras que el agente representa a los jugadores, encargándose así del NLP y de la política. Fig. 1 presenta un diagrama de los componentes y cómo interactúan entre sí.

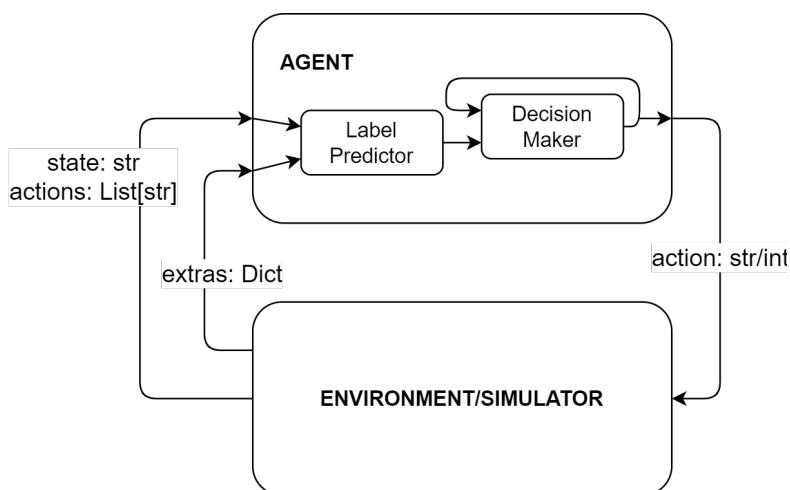


Figura 1: Diagrama completo de la arquitectura

Entorno → agente

- **Contexto** (*str*). El contexto actual dado por su representación textual.
- **Acciones** (*lista de str*). La lista de acciones disponibles que el jugador puede elegir para tomar, dada con su representación de texto.
- **Extras** (*diccionario*). Un diccionario de pares clave-valor con información adicional para el agente, como imágenes en el juego de texto, o información no destinada a ser utilizada por el agente mientras juega, como información para el entrenamiento del agente o el análisis de datos.

Agente → entorno

- **Acción** (*str/int*). La acción realizada por el jugador. Puede darse como la representación textual de la acción o como su posición en la lista de acciones disponibles.

2.1. Entorno

El entorno o simulador es la representación de la historia. El entorno se encarga de seguir el hilo de la historia y de cómo afectan las acciones del agente. Este componente puede generalizarse utilizando un grafo de decisiones. Un grafo de decisiones es un grafo que contiene todos los estados del juego y las acciones disponibles en cada estado.

2.2. Agente

El agente es la representación de los jugadores. Los jugadores del juego procesan la información sobre el contexto y seleccionan la acción que creen que consigue un mejor resultado en la historia. En los juegos basados en recompensas, el mejor resultado se obtiene cuando se llega al estado terminal con una elevada suma de recompensas. Sin embargo, este enfoque supone que los jugadores tienen información sobre la puntuación que reciben (o el impacto que tienen sus acciones en el final) y que las acciones y los finales pueden etiquetarse como “buenos” o “malos”. Estas suposiciones no se sostienen bien en juegos centrados en la toma de decisiones y no en la obtención de una puntuación alta, por ejemplo en los juegos serios. Además, en estos juegos nos interesa recopilar datos sobre cómo los jugadores toman decisiones. Estos datos se obtienen sobre todo en los primeros intentos, ya que, tras un par de ellos, los jugadores empiezan a memorizar el impacto de sus decisiones.

En cambio, supondremos que los agentes no tienen información sobre el impacto de sus acciones en los finales y que tratan de maximizar sus puntuaciones subjetivas para cada acción, tomando las acciones que consideran mejores. Cada agente será asignado un perfil de comportamiento. Los agentes seleccionan las acciones que mejor se alinean con los rasgos de carácter de su perfil. Mientras que las decisiones pasadas pueden influir en las acciones actuales del agente, las decisiones futuras no pueden, ya que asumimos que los agentes no saben qué estados y acciones futuras estarán disponibles.

Los agentes se dividen en el predictor de etiquetas y el decisor.

2.2.1. Predictor de Etiquetas basado en NLP

Se encarga de procesar la información dada por el juego y obtener el grado de alineación entre las acciones y los rasgos de carácter, o etiquetas, que definen los perfiles de comportamiento. Esta información se utilizará posteriormente en Sec. 2.2.2 para seleccionar las acciones a realizar.

La entrada de este subcomponente es la misma que la del agente, el contexto actual y una lista $[a_1, \dots, a_I]$ de I acciones. La extracción de un conjunto $\{l_1, \dots, l_J\}$ de J etiquetas predefinidas (rasgos de carácter) puede verse como una tarea de clasificación multietiqueta. Por lo tanto, el resultado del subcomponente de predicción de etiquetas será un tensor H bidimensional ($I \times J$) en el que el elemento $h_{i,j}$ es la probabilidad de que la acción a_i se alinee con el rasgo de carácter l_j .

Una solución simple podría ser proporcionar manualmente estas etiquetas. Sin embargo, hacerlo así sesgaría el resultado hacia lo que creemos que implican las acciones. En cambio, estas etiquetas deberían ser lo que los jugadores creen que implica la acción cuando la realizan. Una forma de evitar este sesgo es utilizar un modelo de NLP preentrenado para la extracción de etiquetas. Sin embargo, encontrar un modelo preentrenado

en un vocabulario similar al del juego en cuestión que prediga los rasgos de carácter de una acción puede ser complicado. En su lugar, podemos construir y entrenar un modelo para esta tarea. Dado que el entrenamiento de un modelo de NLP suele ser costoso y requiere una gran cantidad de datos, haremos *fine-tune* a un modelo preentrenado como BERT [2]. Si es posible, utilizaremos varios juegos o historias con vocabularios similares para entrenar nuestro modelo y reducir el sesgo hacia el juego.

A continuación, presentamos una arquitectura que puede utilizarse para extraer las etiquetas. Esta arquitectura permite utilizar un modelo de NLP preentrenado y ajustarlo al conjunto de acciones, estados e información adicional de la historia. En Fig. 2 se presenta un diagrama de la arquitectura que utiliza sólo la representación textual del contexto y la acción. Sin embargo, si se quisiera incorporar información adicional, se podría hacer incluyendo un bloque individual adicional. Esta arquitectura recibe como entrada pares estado-acción y devuelve el tensor de probabilidad que será la salida del predictor de etiquetas.

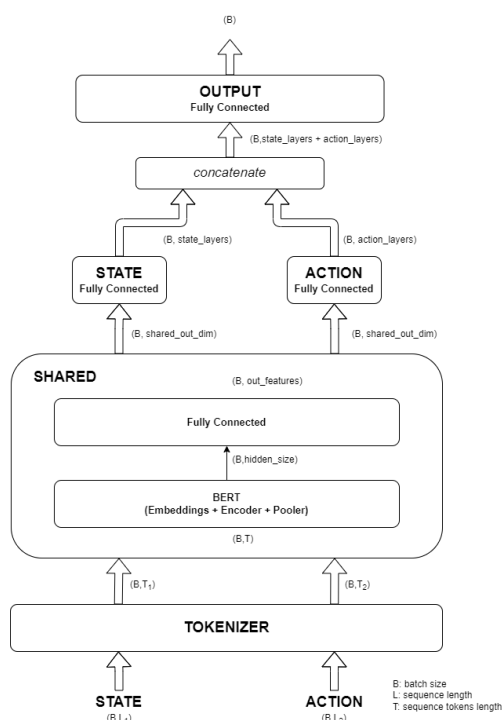


Figura 2: Modelo para el predictor de etiquetas basado en NLP

2.2.2. Decisor

El decisor se encargará del segundo paso del proceso de toma de decisiones del agente. Este segundo y último paso consiste en utilizar los rasgos de carácter, o etiquetas, implicados en cada una de las acciones disponibles para elegir la acción que mejor se

alinee con el perfil de comportamiento del agente. Por lo tanto, este subcomponente es el núcleo del proceso de toma de decisiones y se encarga de tomar decisiones como lo haría un jugador real.

El decisor recibe su entrada del predictor de etiquetas (Sec. 2.2.1) y emite la acción elegida por el agente (teniendo así su misma salida). Dado un perfil de comportamiento, el decisor asigna una probabilidad a cada acción disponible en función de cómo se alineen las etiquetas de esa acción (rasgos de carácter) con las etiquetas del perfil de comportamiento. Estas probabilidades se obtienen calculando una suma ponderada de todas las etiquetas, cuyas ponderaciones vienen dadas por el perfil de comportamiento.

Formalmente, el decisor recibe un perfil de comportamiento caracterizado por un conjunto $\mathbf{f} = [f_1, \dots, f_J]$ de funciones y un número N de acciones pasadas a considerar. Cada función $f_j : \mathcal{M}_{(N+1) \times J}(\mathbb{R}) \times \mathcal{M}_{N+1,1}(\mathbb{R}) \rightarrow \mathbb{R}$ devuelve la importancia del rasgo de carácter $l_j \in \{l_1, \dots, l_J\}$ para la toma de decisiones. Esta importancia se da en forma de puntuación. Cuanto mayor sea la puntuación, mayor será la importancia de l_j . Si la importancia es positiva, el perfil de comportamiento se alinea positivamente con l_j , favoreciendo las acciones con dicho rasgo. Si es negativa, desfavorece las acciones con dicho rasgo l_j . Estas funciones f_j toman como entrada la matriz de memoria M de N interacciones y el vector de iteraciones S .

La matriz de memoria M es una matriz $(N+1) \times J$ donde cada fila representa la iteración $t-n \in \{t, t-1, \dots, t-N\}$ (siendo t la iteración actual) y cada columna representa el rasgo de carácter l_j . Así, M se define como

$$M = \begin{bmatrix} 1 & 1 & \dots & 1 \\ l_1^{t-1} & l_2^{t-2} & \dots & l_J^{t-1} \\ \vdots & \vdots & \ddots & \vdots \\ l_1^{t-N} & l_2^{t-N} & \dots & l_J^{t-N} \end{bmatrix} \quad (1)$$

donde el elemento $m_{n+1,j} = l_j^{t-n}$ es el porcentaje en el que la acción realizada por el agente en la iteración $t-n$ se alinea con el rasgo de carácter l_j . Los valores dados para la iteración t (elementos $m_{1,j}$) son 1 ya que la decisión actual aún no se ha tomado. Estos valores son la percepción del agente sobre sus decisiones pasadas, ya que el agente no tiene acceso a las verdaderas etiquetas. Dado que el predictor de etiquetas proporciona la percepción del agente sobre las acciones, el decisor guardará los rasgos de carácter predichos de las decisiones que ha tomado para N iteraciones anteriores. Es importante señalar que la importancia de un determinado rasgo de carácter l_j puede verse afectada por otras etiquetas l_k de decisiones pasadas aparte de l_j . Por lo tanto, la memoria consiste en el recuerdo del propio agente y no en un valor verdadero para todos los agentes.

El vector de iteraciones S es un vector columna de tamaño $N+1$ tal que $s_t = 1$ si se ha tomado una decisión para el paso $t-1$, y $s_t = 0$ en caso contrario. Este vector da

información sobre la validez de la matriz de memoria. Si s_t es igual a 1, significa que la fila M_{t*} de la matriz de memoria es válida. s_1 siempre es igual a 1 ya que la fila M_{1*} es siempre válida.

Entonces, f_j devuelve la importancia w_j del rasgo de carácter l_j para la decisión actual.

$$w_j = f_j(M, S) \quad (2)$$

En cada paso t , el decisor recibe como entrada una matriz H tal que el elemento $h_{i,j}$ es la probabilidad de que la acción $a_i \in \{a_1, \dots, a_I\}$ se alinee con el rasgo de carácter l_j . Consideramos que $h_{i,j}$ es el grado de alineamiento entre la acción a_i y el rasgo de carácter l_j . Por lo tanto, el vector de probabilidades de tomar una determinada acción disponible viene dado por

$$\mathbf{p}_a(H, \mathbf{w}) = \text{softmax}(H\mathbf{w}^T) \quad (3)$$

donde $\mathbf{w} = [w_1, \dots, w_J]$.

Combinando Ecuación (2) y (3), llegamos a la expresión de decisión general Ecuación (4). Esta ecuación modela cómo las probabilidades de tomar las acciones en $\mathbf{a} = \{a_1, \dots, a_I\}$ se obtienen del perfil de comportamiento del agente, caracterizado por $\mathbf{f} = [f_1, \dots, f_J]$ (with $f_j : \mathcal{M}_{(N+1) \times J}(\mathbb{R}) \times \mathcal{M}_{N+1,1}(\mathbb{R}) \rightarrow \mathbb{R}$), y las entradas H (matriz de etiquetas $I \times J$), M (matriz de memoria $(N+1) \times J$), y S (vector columnas de iteraciones $N+1$).

$$\mathbf{p}_a(H, \mathbf{f}, M, S) = \text{softmax}(H\mathbf{f}(M, S)^T) \quad (4)$$

donde $\mathbf{f}(M, S) = [f_1(M, S), \dots, f_J(M, S)]$.

El agente selecciona qué acción tomar a través de una elección aleatoria ponderada, con los pesos siendo $\mathbf{p}_a$.

3. Aplicación y Resultados

En esta sección, ilustramos la aplicación de esta arquitectura implementándola en un juego serio de texto [5] destinado a educar sobre los ciberdelitos, en particular *Online Grooming*. Por lo tanto, estamos interesados principalmente en cómo los jugadores toman las decisiones en lugar de ganar el juego.

Utilizando un conjunto de perfiles de comportamiento, se han generado datos sintéticos para este juego y para entrenar una variedad de modelos de aprendizaje automático. Los modelos entrenados utilizan las decisiones tomadas a lo largo de cada ejecución

para clasificar cada ejecución según el perfil de comportamiento que la generó. De este modo, estos modelos pueden utilizarse posteriormente en datos reales para encontrar el grado de similitud del comportamiento de los jugadores con respecto a los perfiles dados.

3.1. Simulador

El primer paso es poder jugar al juego. Este juego está disponible en formato [Yarn Spinner](#), por lo que se necesita un parseador para los archivos Yarn Spinner. Se ha utilizado como base un parseador desarrollado por Pfeiffer [4], el cual ha sido ampliado con algunas funcionalidades adicionales necesarias para nuestra historia². A continuación, utilizamos el parseador para obtener el grafo de decisiones para el juego, el cual se usará para el simulador. Esto acelera la simulación, ya que no requiere volver a parsear los archivos para cada ejecución. El grafo de decisión se obtiene mediante *Depth First Search*, que es menos computacionalmente costoso que *Breadth First Search*, debido a la incapacidad de muchos parseadores para retroceder en el juego.

3.2. Predictor de etiquetas basado en NLP

Ahora continuaremos implementando el agente. En primer lugar, implementaremos el predictor de etiquetas, que aborda la tarea de NLP. La opción preferible para reducir el sesgo es utilizar un modelo preentrenado en un diccionario o juego similar. Cuando no existe un modelo preentrenado adecuado para nuestra tarea, podemos utilizar herramientas como TextWorld [1] para entrenar el modelo en un conjunto controlado de juegos usando un diccionario específico. Sin embargo, por simplicidad, se ha entrenado un modelo propio en el juego.

3.2.1. Datos

El primer paso es etiquetar el juego si no está ya etiquetado. El objetivo del predictor de etiquetas es, dado el contexto actual y una acción, analizar la alineación entre la acción y los rasgos de carácter utilizados para definir los perfiles de comportamiento. Por lo tanto, se deben etiquetar las acciones del juego según si están alineadas o no con dichos rasgos.

Debido a la simplicidad del juego, hemos utilizado un único rasgo (etiqueta). En el juego, al jugador se le presenta un conjunto de situaciones en las que podría ser potencialmente víctima de *Online Grooming*. Por lo tanto, etiquetaremos las acciones por su seguridad, nivel de riesgo. Etiquetaremos con 1 las acciones consideradas seguras y con 0 las acciones consideradas de riesgo.

Los datos utilizados para entrenar el modelo se obtienen del grafo de decisión. Estos

²El código está disponible en <https://github.com/vibalcam/yarn-pygame>

datos consisten en una lista de tuplas (*contexto*, *accin*). Cada elemento representa el contexto actual y una acción disponible. El juego en cuestión contiene un total de 1052 pares de tuplas etiquetados.

3.2.2. Modelo

Para el modelo, hemos seguido la arquitectura presentada en Sec. 2.2.1. Dada la gran cantidad de datos necesarios para entrenar un modelo desde cero, aprovechamos el aprendizaje por transferencia utilizando un modelo preentrenado. Lo mejor sería utilizar un modelo preentrenado en un campo similar. Sin embargo, abordaremos el caso general utilizando un modelo preentrenado de uso general como BERT. BERT es un modelo estado del arte ampliamente utilizado para hacer *fine-tuning* en muchas tareas diferentes en NLP, lo que lo convierte en una excelente opción. Además, se puede obtener un modelo BERT preentrenado para el idioma que elijamos en Hugging Face con un esfuerzo mínimo. Hemos optado por utilizar el modelo *BERT multilingual cased* ya que permite que el modelo funcione sin problemas en diferentes idiomas.

Se ha entrenado y probado varios modelos con diferentes dimensiones y aumentos del conjunto de datos. Además, también hemos entrenado un modelo similar que utiliza una LSTM en lugar del modelo preentrenado BERT. En contraste con el modelo basado en BERT, el modelo basado en LSTM no es capaz de aprender la tarea y se comporta como un modelo casi aleatorio. Este resultado era de esperar debido a la escasa cantidad de datos disponibles, que no son suficientes para entrenar el modelo desde cero, y confirma la necesidad de utilizar un modelo preentrenado para aprovechar el aprendizaje por transferencia, el cual ayuda sustancialmente al aprendizaje.

3.3. Decisor

El último paso del proceso es seleccionar el conjunto de perfiles de comportamiento para los agentes. Dado que sólo hay un rasgo de carácter, la seguridad, sólo tenemos que proporcionar una única función $f(M, S)$ que caracterice el perfil de comportamiento para ese rasgo de carácter³ y el número N de acciones pasadas a considerar.

A cada perfil se le ha asignado un id que se utilizará posteriormente para identificarlo en el conjunto de datos. Los perfiles implementados son:

- 0 **Aleatorio.** Perfil sin memoria ($N = 0$) que no da importancia a la seguridad, tomando las decisiones al azar. Caracterizado por $f(M, S) = 0$.
- 1 **Arriesgado.** Perfil sin memoria ($N = 0$) que desfavorece la seguridad, favoreciendo el riesgo. Caracterizado por $f(M, S) = -2$.
- 2 **Seguro (aversión al riesgo).** Perfil sin memoria ($N = 0$) que favorece la seguridad. Caracterizado por $f(M, S) = 2$.

³ M es la matriz de memoria $((N + 1) \times 1)$ y S es el vector columnas de iteraciones $((N + 1) \times 1)$

- 3 **Arriesgado Menos (pero no demasiado).** Perfil con $N = 2$ que favorece el riesgo por naturaleza pero que se vuelve más seguro cuanto más riesgo ha asumido en las últimas 2 acciones. Caracterizado por $f(M, S) = b_{1,1} + S^T((1 - M) \odot B)$, donde $\odot$ es la multiplicación elemento a elemento, 1 es la matriz $(N + 1) \times J$ de unos, y
- $$B = \begin{bmatrix} -8 \\ 8 \\ 6 \end{bmatrix}.$$
- 4 **Seguro Menos (pero no demasiado).** Perfil con $N = 2$ que favorece la seguridad por naturaleza pero se vuelve más arriesgado cuanto más seguro ha sido en las últimas 2 acciones. Caracterizado por $f(M, S) = S^T(M \odot B)$ donde $\odot$ es la multiplicación elemento a elemento y $B = \begin{bmatrix} 8 \\ -8 \\ -6 \end{bmatrix}.$

3.4. Generación de Datos y Entrenamiento de Modelos

Dejamos que cada agente juegue 100.000 veces (ejecuciones) el juego sin fijar la semilla aleatoria, por lo que las ejecuciones no son deterministas. Registramos las acciones tomadas para cada decisión, así como el id del perfil de comportamiento del agente, y utilizamos esa información para construir el conjunto de datos.

A continuación se entrenan multiples modelos de aprendizaje automático usando los datos generados. Los modelos reciben como entrada el conjunto de acciones realizadas para cada pregunta en una ejecución (o -1 si la pregunta no se aplica a dicha ejecución) y se entrenan para predecir qué perfil de comportamiento la generó. Por lo tanto, esta tarea será supervisada ya que tenemos información sobre qué perfil de comportamiento generó cada ejecución. Esto permite entrenar los modelos usando datos sintéticos y luego utilizarlos en datos reales para clasificar a los jugadores reales por perfil de comportamiento o para encontrar el grado de similitud entre su comportamiento y los perfiles de comportamiento dados. Este es un punto fuerte a favor del uso de esta técnica de generación en comparación con otras técnicas, que implicarían que esta tarea no fuera supervisada.

En el siguiente análisis, es importante recordar que el conjunto de datos generado está equilibrado, teniendo el mismo número de muestras para cada perfil, lo que implica que el *accuracy* puede ser utilizada como una métrica fiable. Tabla 1 resume las métricas de *test* obtenidas por estos modelos. Todos los modelos obtienen resultados sustancialmente mejores que los de un modelo aleatorio (0,2 de *accuracy* global al haber cinco clases). El modelo AdaBoost obtiene los peores resultados con un *accuracy* de 0,647 y un *f1-score* de 0,63, mientras que los mejores resultados los obtienen los modelos de árbol de decisión y MLP con un *accuracy* de 0,8 y un *f1-score* de 0,8.

También se observa que los resultados varían en función de la clase. Son especial-

Cuadro 1: Métricas de *test* obtenidas por los modelos de clasificación de perfiles de comportamiento

Perfil	Métrica	DT ^a	LR ^b	Ada ^c	RF ^d	KNN ^e	MLP ^f
Aleatorio	F1	0.583	0.506	0.488	0.581	0.548	0.577
	Accuracy	0.527	0.444	0.426	0.522	0.517	0.51
Arriesgado	F1	0.819	0.805	0.466	0.819	0.784	0.82
	Accuracy	0.81	0.815	0.333	0.81	0.789	0.817
Seguro	F1	0.842	0.83	0.765	0.842	0.82	0.843
	Accuracy	0.825	0.811	0.681	0.826	0.819	0.833
Arriesgado Menos	F1	0.862	0.809	0.671	0.861	0.841	0.861
	Accuracy	0.928	0.858	0.912	0.929	0.86	0.921
Seguro Menos	F1	0.881	0.821	0.762	0.88	0.861	0.881
	Accuracy	0.926	0.88	0.886	0.928	0.888	0.935
Global	F1	0.797	0.753	0.63	0.796	0.77	0.796
	Accuracy	0.802	0.761	0.647	0.802	0.774	0.803

^aÁrbol de decisión ^bRegresión logística ^cClasificador AdaBoost ^dRandom Forest ^eK-nearest neighbor ^fPerceptrón multicapa con optimizador adam

mente interesantes los resultados obtenidos para la clase aleatoria, en torno a 0,5 de *accuracy* y *f1-score*. Una explicación para esto es que, dado que este perfil actúa completamente al azar, no tiene ningún patrón de decisión. Por lo tanto, cada una de las ejecuciones generadas por este perfil puede ser similar a las generadas por otros perfiles, lo que lleva a los modelos a clasificar esas ejecuciones incorrectamente. Esto no es un problema de los datos, sino un problema inherente al comportamiento aleatorio. Esto se confirma al comparar los resultados de cada modelo en el perfil aleatorio con el resto de los perfiles. Es necesario utilizar otros modelos para diferenciar el comportamiento aleatorio del resto.

A excepción del perfil aleatorio, todas las clases obtienen puntuaciones similares, con un *accuracy* y un *f1-score* en torno a 0,8–0,9. Con estos resultados, podemos concluir que la generación de datos sintéticos ha logrado entrenar modelos capaces de utilizar las decisiones tomadas para diferenciar los perfiles de comportamiento.

4. Conclusiones

Algunas contribuciones clave de este trabajo han sido incorporadas a un artículo científico titulado «A Child’s Play: An Agent-based Simulator to Protect Minors Online» [3], que ha sido aceptado en una importante conferencia internacional en el campo de las ciencias sociales computacionales. Este artículo se presentará en la 8ª Conferencia

Internacional de Ciencias Sociales Computacionales (*IC²S²*) celebrada en la Universidad de Chicago (19–22 de julio de 2022).

La generación de datos es una técnica que se ha utilizado durante mucho tiempo para hacer frente a la falta de datos reales y para mejorar los modelos entrenados con esos datos. Hemos proporcionado un nuevo enfoque adecuado para juegos de decisión y esperamos que este trabajo abra la exploración de la generación de datos en este tipo de juegos.

Hemos presentado un modelo para generar datos sintéticos en juegos centrados en la toma de decisiones y no en la obtención de la máxima puntuación, o en los que no existen las recompensas (no hay decisiones “buenas” o “malas”). Nuestro modelo genera datos sintéticos creando agentes que imitan el comportamiento de los jugadores reales y recolectando datos mientras juegan. A cada uno de los agentes se le asigna un perfil de comportamiento relativo a varios rasgos de carácter que utilizan para la toma de decisiones. Los agentes seleccionan las acciones que mejor se ajustan a los rasgos de su perfil, teniendo en cuenta también las acciones pasadas. Los perfiles de comportamiento utilizados, los rasgos de carácter que los caracterizan y el número de acciones pasadas a considerar son hiperparámetros que permiten al modelo ajustarse a diversas aplicaciones y juegos. Este enfoque no requiere de datos preexistentes. Por lo tanto, no sufre el sesgo intrínseco y la falta de variabilidad que surgen al utilizar estos datos. Además, los agentes no siguen un enfoque basado en la recompensa, ya que esto sería inadecuado para los juegos centrados en la toma de decisiones o en los que las recompensas no son aplicables. Además, nuestro modelo incorpora información sobre el comportamiento en los datos generados y, por tanto, en los modelos entrenados con ellos. Estos modelos pueden entonces detectar o analizar estos perfiles de comportamiento en los datos reales. En otras palabras, nuestro modelo transforma la tarea de analizar el comportamiento de los jugadores reales de una tarea no supervisada a una supervisada.

Hemos ilustrado la aplicación y las ventajas del modelo implementándolo en un juego de texto serio existente. El conjunto de datos sintéticos generados se utiliza para entrenar varios modelos de Machine Learning que reciben como entrada las decisiones tomadas en cada ejecución y que son entrenados para predecir el perfil de comportamiento que la generó. Los resultados obtenidos por los modelos son sustancialmente mejores que los de un modelo aleatorio (ver Tabla 1), alcanzando *accuracy* y *f1-score* globales de 0,8.

Existen múltiples líneas de trabajo futuro que consisten en mejorar el predictor de etiquetas para que se ajuste mejor a la percepción del jugador, mejorar la memoria del decisor para que se ajuste mejor a la percepción del pasado del jugador, disminuir el tiempo de implementación, probar el modelo en otros juegos de decisión aparte de los juegos de texto, y probar el modelo con otros juegos de texto y en un escenario real. Cada una de estas líneas pretende aumentar las capacidades del modelo o probar su

aplicación en una gama más amplia de escenarios, incluidos los reales.

Referencias

- [1] Marc-Alexandre Côté y col. «TextWorld: A Learning Environment for Text-based Games». Ver. 2. En: (2019). DOI: [10.48550/ARXIV.1806.11532](https://doi.org/10.48550/ARXIV.1806.11532). URL: <https://arxiv.org/abs/1806.11532> (visitado 31-05-2022) (vid. pág. 9).
- [2] Jacob Devlin y col. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». Ver. 2. En: (2018). DOI: [10.48550/ARXIV.1810.04805](https://doi.org/10.48550/ARXIV.1810.04805). URL: <https://arxiv.org/abs/1810.04805> (visitado 05-06-2022) (vid. pág. 6).
- [3] Jaime Pérez y col. «A Child's Play: An Agent-based Simulator to Protect Minors Online». En: IC2S2 (8th International Conference on Computational Social Science). Chicago, 2022 (vid. pág. 12).
- [4] Robert Pfeiffer. *Yarn.Py, a Python Runtime for Yarn Dialogue*. 29 de oct. de 2020. URL: <https://github.com/robertpfeiffer/yarn-pygame> (visitado 17-06-2022) (vid. pág. 9).
- [5] Ana Leticia Urbistondo Murua. «Creación de Un Videojuego Interactivo Para Educar En El Uso Responsable de Las Redes Sociales». Tesis de lic. Universidad Pontificia Comillas, Escuela Técnica Superior de Ingeniería (ICAI), 2020. URL: <https://repositorio.comillas.edu/xmlui/handle/11531/41473> (visitado 13-02-2022) (vid. pág. 8).

TEXT GAMES SYNTHETIC DATA GENERATION BASED ON BEHAVIOR PROFILES

Author: Balmaseda del Campo, Vicente.

Supervisor: López López, Gregorio Ignacio.

Supervisor: Pérez Sánchez, Jaime.

Collaborating Entity: ICAI — Universidad Pontificia Comillas.

ABSTRACT

Data generation is a technique used to cope with the lack of real data and to improve the models trained on that data. However, most of the techniques for data generation on games either require pre-existing data, thus usually suffering from bias and low variability, or are focused on obtaining high scores, which is not suitable for games where the focus is on decision-making or where rewards are not applicable, such as serious games. We present an architecture to generate synthetic data suitable for decision-making games. Our model tackles the above-stated limitations by creating agents that imitate real players' behavior and gathering the data as they play the game. Therefore, our approach does not require pre-existing data. The agents are each given a behavior profile regarding several character traits they use for decision-making. The agents select the actions that best align with their profile's character traits while also considering past actions. In addition, this architecture incorporates behavior information into the generated data and, therefore, into the models trained with it. We illustrate this architecture's application by implementing it on a serious text game aimed at educating about cybercrimes, particularly *Online Grooming*. We generate a synthetic dataset for the game and train various Machine Learning models on it. The generated data successfully teaches these models to predict the behavior profile that generated each game run given as input the decisions taken. These models can then be used to find the degree of similarity of the real players' behavior with respect to the given profiles.

Keywords: artificial intelligence, behavioral analysis, machine learning, natural language processing, serious (text-based) games, synthetic data generation

1 Introduction

Text-based games are textual turn-based single-player games. Each turn, the player is presented with a description of the context or current state and a list of available actions, both given by their natural language textual representation. The player then chooses an action that changes the game state, producing a new state description and a

new set of actions. The game ends when the player reaches a terminal state. Therefore, players in the game try to find an optimal policy that will provide them with the best ending result.

Text games have other applications besides entertainment. Some of these applications are educational purposes, behavioral analysis, or even research. The European H2020 [RAYUELA](#) project, in which this work is framed, is developing an interactive story-like serious games whose objective is twofold: on the one side, it will allow minors to learn good practices online while playing; and, on the other side, the data gathered through the game will be analyzed to try to identify potential risk factors so that Law Enforcement Agencies can develop evidence-based policies to protect minors online. These applications take advantage of the game's entertaining ability to gather data which is then fed into a model. However, data may be unavailable at the time, insufficient, or even unobtainable.

This lack of data can be palliated by generating synthetic data to expand the dataset and using it as a temporal or permanent substitute for real data. Existing techniques generate data by taking advantage of available real data (or data information such as its distribution) or simulating policy-based agents playing. However, both have flaws. Techniques based on available data suffer from bias towards their training data and tend to generate synthetic data with low variance. On the other hand, most techniques based on agents use rewards to obtain the best final result. This makes these techniques unsuitable for games where the focus is on decision-making instead of winning or where rewards are not applicable, such as serious games. Since the focus is on decision-making, synthetic agents used for data generation must behave as players would, which entails modeling these agents according to the behavior of real players. Therefore, generating data for serious games requires a new approach based on the behavior profiles of the real players. Moreover, this approach must also tackle the Natural Language Processing (NLP) difficulty that comes with natural language text.

This work proposes a model to generate synthetic data in games, especially serious games, where the focus is on decision-making rather than obtaining the highest score or where rewards are not applicable. Since most decision-making games can be converted into text games, we will focus on these. However, our model can also deal with other kinds of information such as images or audio.

The model presented tackles the above-stated existing difficulties by meeting the following requirements:

- **Considers behavior profiles** with regard to a set of character traits¹.
- **Independent from existing data**

¹By character traits we are referring to the a set of behavioral traits or qualities that define one's type of person. Active, calm, careful and daring are some examples of character traits.

-
- **Reusability**
 - **Supervised training of behavior profiles.** Through this process, the information given by the behavior profiles can be incorporated into the models and used to detect or analyze them on real data.
 - Story, NLP, and policy (behavior profile) modeling **task segregation**

The remainder of this work is structured as follows. Sec. 2 presents the architecture with its components and interactions, providing an explanation of each component’s functionality. This architecture is then illustrated in Sec. 3 by implementing it on a serious text game about *Online Grooming*. We generate a synthetic dataset for the game and use this data to train a variety of Machine Learning models on classifying each game run by the behavior profile that generated it. We also illustrate how this models can be used on real data to find the degree of similarity of the players’ behavior with respect to the given behavior profiles. Conclusions and future work can be found at the end in Sec. 4.

2 Architecture Description

The main idea behind the proposed model is to use a set of behavior profiles to create policy-driven synthetic agents that will play the game. While the agents play the game, synthetic data will be collected as if they were real players. By doing so, we are generating data with the agents’ policy implementing the behavior profiles and without using pre-existing data.

The model presented is divided into two main components, the simulator or environment, and the agent or player. Each of these components are independent and interact with each other through an interface. This modular approach eases interdisciplinary collaboration by separating the story, NLP, and policy (behavior profile) modeling sub-tasks into independent components. The environment takes care of the game’s storyline, while the agent represents the players, thus taking care of the NLP and policy modeling. Fig. 1 presents a diagram of the components and how they interact with each other.

Environment → agent

- **State** (*str*). The current context or game state, given by its text representation.
- **Actions** (*list of str*). The list of available actions the player can choose to take, given with its text representation.
- **Extras** (*dictionary*). A dictionary of key-value pairs with additional information for the agent, such as images in the text game, or information not intended to be used by the agent while playing, such as information for training the agent or data analytics.

Agent → environment

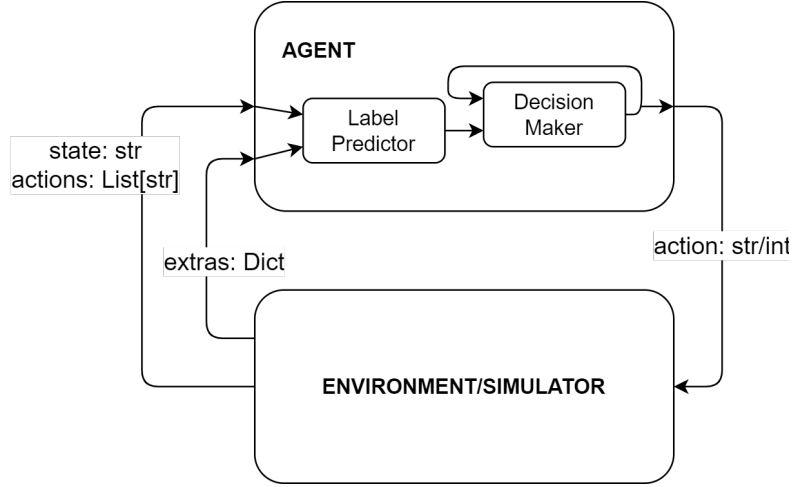


Figure 1: Complete architecture diagram

- **Action** (*str/int*). The action taken by the player. It can be given as the text representation of the action or as its position in the list of available actions.

2.1 Environment

The environment or simulator component is the representation of the story. The environment is responsible for following the thread of the story and how the agent's actions affect it. This component can be generalized by using a decision graph, which holds all the information. A decision graph is a graph that holds all the game states and available actions at each state.

2.2 Agent

The agent is the representation of the players. The players in the game process thoughtfully the information received from the environment and select the action that they believe produces the best impact on the story. In reward-driven games, the best result is obtained when we arrive at the terminal state with a high sum of rewards. However, this approach assumes players have information about the score they receive (or the impact their actions have on the ending) and that actions and endings can be labeled as "good" or "bad". These assumptions do not hold well when dealing with games where the focus is on decision-making instead of obtaining a high score, such as serious games. Moreover, in these games we are interested in gathering data about how players make decisions. This data is mostly obtained on the first few attempts since, after a couple of tries, the players start memorizing the impact of their decisions.

Instead, we assume the agents do not have information about the impact of their actions on the endings and try to maximize their subjective scores for every action,

taking the actions they consider best. We give each agent a specific behavior profile. The agents select the actions that best align with the character traits of their behavior profile. While past decisions can influence the agent’s current actions, future decisions can not since we assume the agents do not know what future states and actions will be available.

The agents are divided into the label predictor and the decision maker.

2.2.1 NLP-Based Label Predictor

The NLP-based label predictor subcomponent will be responsible for the first step of the agent’s decision-making process. This first step involves processing the information given by the game and obtaining the degree of alignment between the actions and the character traits, or labels, that define the behavior profiles. This information is later used in Sec. 2.2.2 to select which actions to take.

The input of this subcomponent is the same as the agent’s input, the current context and a list $[a_1, \dots, a_I]$ of I actions. Extracting a set $\{l_1, \dots, l_J\}$ of J predefined labels (character traits) can be seen as a multi-label classification task. Therefore, the output of the label predictor subcomponent will be a 2-dimensional $I \times J$ tensor H such that the element $h_{i,j}$ is the probability that action a_i aligns with the character trait l_j .

A naïve approach could be manually providing these labels. However, doing so would bias the result towards what we believe the actions entail. Instead, these labels should be what the players will believe the action entails when they reach it. A way to avoid this bias is by using a pre-trained NLP model as a proxy for the label extraction. However, finding a model trained on a vocabulary similar to the one used by the game at hand that predicts the character traits of an action can be tricky. Instead, we can build and train a model for this task. Since training an NLP model is usually expensive and requires a high quantity of data, we will fine-tune a pre-trained model such as BERT [2]. If possible, we will use multiple games or stories with similar vocabularies to fine-tune our model to reduce bias towards the game.

Next, we present an architecture that can be used to extract the required labels. This architecture allows using a pre-trained NLP model and adjusting to the story’s set of actions, states, and additional information. A diagram of the architecture using only states and action text information is presented in Fig. 2. However, if additional information wanted to be incorporated, it could be done by including an additional individual block. This architecture receives as input state-action pairs and returns the probability tensor that will be the output of the label predictor.

2.2.2 Decision Maker

The decision maker subcomponent will be responsible for the second step of the agent’s decision-making process. This second and last step involves using the extracted

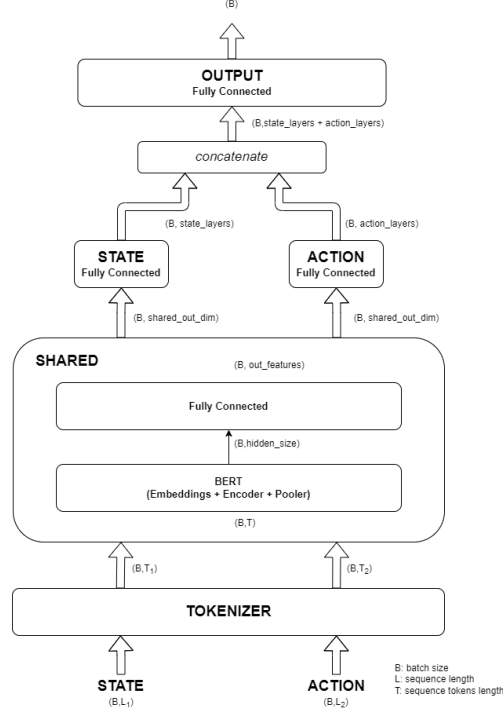


Figure 2: Model for the NLP-based label predictor

character traits, or labels, involved in each of the available actions to choose the action that best aligns with the agent’s given behavior profile. Therefore, this subcomponent is the core of the decision-making process and is responsible for using the provided behavior profile to make decisions as the real player would.

The decision maker receives its input from the label predictor (Sec. 2.2.1) and outputs the action chosen by the agent (thus having the same output as the agent). Given a behavior profile, the decision maker will assign a probability to each available action depending on how well that action’s labels (character traits) align with the behavior profile’s labels. These probabilities are obtained by calculating a weighted sum of all the labels, with the weights being given by the behavior profile.

Formally, the decision maker is given a behavior profile characterized by a set $\mathbf{f} = [f_1, \dots, f_J]$ of functions and a number N of past steps to consider. Each function $f_j : \mathcal{M}_{(N+1) \times J}(\mathbb{R}) \times \mathcal{M}_{N+1,1}(\mathbb{R}) \rightarrow \mathbb{R}$ returns the importance of the character trait $l_j \in \{l_1, \dots, l_J\}$ for decision making. This importance is given as a score. The higher the returned value, the higher the importance of l_j . If the importance is positive, the behavior profile aligns positively with l_j , favoring actions with that character trait. If it is negative, it disfavors actions with character trait l_j . These f_j functions take as input the N -steps memory matrix M and step vector S .

The memory matrix M is a $(N + 1) \times J$ matrix where each row represents the step $t - n \in \{t, t - 1, \dots, t - N\}$ (with t being the current step) and each column represents the character trait l_j . Thus, M is defined as

$$M = \begin{bmatrix} 1 & 1 & \dots & 1 \\ l_1^{t-1} & l_2^{t-2} & \dots & l_J^{t-1} \\ \vdots & \vdots & \ddots & \vdots \\ l_1^{t-N} & l_2^{t-N} & \dots & l_J^{t-N} \end{bmatrix} \quad (1)$$

where the element $m_{n+1,j} = l_j^{t-n}$ is the percentage by which the action taken by the agent in step $t - n$ aligns with the character trait l_j . The values given for step t (elements $m_{1,j}$) are 1 since the current decision has not yet been taken. These values are the agent's perception about his past decisions since the agent does not have access to the true labels. Since the label predictor provides the agent's perception of the actions, the decision maker will save the predicted character traits of the decisions it has taken for N past steps. It is important to note that the importance of a given character trait l_j can be impacted by other labels l_k of past decisions apart from l_j . Therefore, the memory consists of the agent's own recollection rather than a true value for all agents.

The step vector S is a column vector of size $N + 1$ such that $s_t = 1$ if a decision has been taken for step $t - 1$, and $s_t = 0$ otherwise. This vector gives information about the validity of the memory matrix. If s_t equals 1, this means that row M_{t*} of the memory matrix is valid. s_1 always equals 1 since row M_{1*} is always valid.

Then, f_j returns the importance w_j of the character trait l_j for the current decision.

$$w_j = f_j(M, S) \quad (2)$$

On each step t , the decision maker receives as input a matrix H such that the element $h_{i,j}$ is the probability that action $a_i \in \{a_1, \dots, a_I\}$ aligns with the character trait l_j . We consider $h_{i,j}$ to be the percentage by which action a_i aligns with the character trait l_j . Therefore, the vector of probabilities of taking a certain available action is given by

$$\mathbf{p}_a(H, \mathbf{w}) = \text{softmax}(H\mathbf{w}^T) \quad (3)$$

where $\mathbf{w} = [w_1, \dots, w_J]$.

By combining Equation (2) and (3), we arrive at the general decision-making expression Equation (4). This equation models how the probabilities of taking the actions in $\mathbf{a} = \{a_1, \dots, a_I\}$ are obtained from the agent's behavior profile, characterized by $\mathbf{f} = [f_1, \dots, f_J]$ (with $f_j : \mathcal{M}_{(N+1) \times J}(\mathbb{R}) \times \mathcal{M}_{N+1,1}(\mathbb{R}) \rightarrow \mathbb{R}$), and the inputs H ($I \times J$ labels matrix), M ($(N + 1) \times J$ memory matrix), and S ($N + 1$ step column vector).

$$\mathbf{p}_a(H, \mathbf{f}, M, S) = \text{softmax}(H\mathbf{f}(M, S)^T) \quad (4)$$

where $\mathbf{f}(M, S) = [f_1(M, S), \dots, f_J(M, S)]$.

The agent selects which action to take through a weighted random choice, with the weights being $\mathbf{p}_a$.

3 Implementation and Results

In this section, we illustrate this architecture’s application by implementing it on a serious text game [5] aimed at educating about cybercrimes, particularly *Online Grooming*. Thus, we are primarily interested in how players make the decisions rather than beating the game.

Using a set of behavior profiles, we generate a synthetic dataset for this game and use it to train a variety of Machine Learning models. The trained models use the decisions taken throughout each run to classify each run by the behavior profile that generated it. By doing so, these models can later be used on real data to find the degree of similarity of the players’ behavior with respect to the given behavior profiles.

3.1 Simulator

The first step is to be able to play the game. This game is available in [Yarn Spinner](#) format, thus we need a parser for the Yarn Spinner files. We used as base a parser developed by Pfeiffer [4] and expanded it with some additional functionality needed to work with our story². We then use the parser to obtain the decision graph for the game and use it for the simulator component. This speeds up the simulation since it does not require reparsing the files for each run. The decision graph is obtained through Depth-First Search, which is computationally less costly than Breadth First Search since many parsers are not able to go backward in the story.

3.2 NLP-Based Label Predictor

We will now continue by implementing the agent. First, we will implement the label predictor, which tackles the NLP task. The preferable choice to reduce bias is to use a pre-trained model on a similar dictionary or game. When there is no suitable pre-trained model for our task, we can use tools such as TextWorld [1] to train our model on a controlled set of games with our dictionary. However, for the sake of simplicity, we will train the model in our game.

²The code is available on <https://github.com/vibalcam/yarn-pygame>

3.2.1 Data

The first step is labeling the game if it is not already labeled. The objective of the label predictor is, given the current state and available action, to analyze the alignment between the action and the character traits used to define the behavior profiles. Thus, we must label the game's actions according to whether they are aligned or not with the given traits.

Due to the game's simplicity, we have used a single label. In the game, the player is presented with a set of situations in which he could be potentially be victim of *Online Grooming*. Therefore, we will label the actions by their safety, risk level. Therefore, label 1 for the actions considered safe, and 0 for the actions considered risky.

The data used to train the label predictor model is obtained from the decision graph. This data consists of a list of $(state, action)$ tuples. Each element represents the current state and an available action that can be taken. The game in hand contains a total of 1052 labeled unique state-action pairs.

3.2.2 Model

For the model, we have followed the architecture presented in Sec. 2.2.1. Given the large amount of data required to train a model from scratch, we take advantage of transfer learning using a pre-trained model. It would be best to use a model pre-trained in a similar field. However, we will tackle the general case using a general pre-trained model such as BERT. BERT is a state-of-the-art model widely used for fine-tuning many different tasks in NLP, making it an excellent choice for our task. Moreover, a pre-trained BERT model for the language of our choice can be obtained from Hugging Face with minimal effort. We have chosen to use the *BERT multilingual cased* model since it allows the model to work seamlessly in different languages.

Several models have been trained and tested with different dimensions and dataset augmentations. In addition, we have also trained a similar model that uses an LSTM instead of the BERT pre-trained model. In contrast with the BERT-based model, the LSTM-based model is not capable of learning the task and behaves as a nearly random model. This result was expected due to the small amount of data available, which is not enough to train the model from scratch, and confirms the need to use a pre-trained model to take advantage of transfer learning, which helps substantially the model's learning.

3.3 Decision Maker

The last step of the process is to select the set of behavior profiles for the agents. Since there is only one character trait, safety, we only need to provide a single function $f(M, S)$ that characterizes the behavior profile for that character trait³ and the number

³ M is the memory matrix $((N + 1) \times 1)$ and S is the steps column vector $((N + 1) \times 1)$

N of past steps to consider.

Each profile has been assigned an id that will be later used to identify it in the dataset. The profiles implemented are:

- 0 **Random.** No-memory profile ($N = 0$) that does not give any importance to safety, taking decisions randomly. Characterized by $f(M, S) = 0$.
- 1 **Risky.** No-memory profile ($N = 0$) that disfavors safety, favoring risk. Characterized by $f(M, S) = -2$.
- 2 **Safe (risk aversion).** No-memory profile ($N = 0$) that favors safety. Characterized by $f(M, S) = 2$.
- 3 **Risky NTM (but not too much).** Profile with $N = 2$ that favors risk by nature but becomes safer the more risk it has taken in the past 2 actions. Characterized by $f(M, S) = b_{1,1} + S^T((1 - M) \odot B)$, where $\odot$ is the element-wise multiplication, 1 is the $(N + 1) \times J$ matrix of ones, and $B = \begin{bmatrix} -8 \\ 8 \\ 6 \end{bmatrix}$.
- 4 **Safe NTM (but not too much).** Profile with $N = 2$ that favors safety by nature but becomes riskier the safer it has been in the past 2 actions. Characterized by $f(M, S) = S^T(M \odot B)$ where $\odot$ is the element-wise multiplication and $B = \begin{bmatrix} 8 \\ -8 \\ -6 \end{bmatrix}$.

3.4 Data Generation and Model Training

We let each agent play 100,000 times (runs) the game without locking the random seed so the runs are not deterministic. We record the actions taken by the agents for each decision, as well as the agent's behavior profile id, and use that information to build the dataset.

We will now test the generated synthetic dataset by training a few Machine Learning models on it. The models are given as input the action taken for each question in a run (or -1 if the question does not apply for that run) and are trained to predict which behavior profile generated that run. Therefore, this task will be supervised since we have information about which behavior profile generated each run. This allows training models on the synthetic data and then using them on real data to classify the real players by behavior profile or to find the degree of similarity of the players' behavior with respect to the given behavior profiles. This is a strong point in favor of using this generation technique compared to other techniques, which would entail this task being unsupervised.

In the following analysis, it is important to remember that the dataset generated is balanced, having the same number of samples for each profile, which entails that the

Table 1: Behavior profile classification models test metrics

Profile	Metric	DT ^a	LR ^b	Ada ^c	RF ^d	KNN ^e	MLP ^f
Random	F1	0.583	0.506	0.488	0.581	0.548	0.577
	Accuracy	0.527	0.444	0.426	0.522	0.517	0.51
Risky	F1	0.819	0.805	0.466	0.819	0.784	0.82
	Accuracy	0.81	0.815	0.333	0.81	0.789	0.817
Safe	F1	0.842	0.83	0.765	0.842	0.82	0.843
	Accuracy	0.825	0.811	0.681	0.826	0.819	0.833
Risky NTM	F1	0.862	0.809	0.671	0.861	0.841	0.861
	Accuracy	0.928	0.858	0.912	0.929	0.86	0.921
Safe NTM	F1	0.881	0.821	0.762	0.88	0.861	0.881
	Accuracy	0.926	0.88	0.886	0.928	0.888	0.935
Global	F1	0.797	0.753	0.63	0.796	0.77	0.796
	Accuracy	0.802	0.761	0.647	0.802	0.774	0.803

^aDecision tree ^bLogistic regression ^cAdaBoost ^dRandom forest

^eK-nearest neighbors ^fMulti-layer Perceptron using adam optimizer

accuracy is reliable as a test metric. Table 1 summarizes the test metrics obtained by these models. All the tested models obtain substantially better results than random (0.2 global accuracy since there are five classes). The AdaBoost model obtains the worst results with 0.647 accuracy and 0.63 f1-score, while the best results are obtained by the decision tree and MLP models with 0.8 accuracy and 0.8 f1-score.

It can also be observed that the results vary depending on the class. Of particular interest are the results obtained for the random class, around 0.5 accuracy and f1-score. An explanation for this is that, since this profile acts completely random, it has no decision-making pattern. Therefore, each of the runs generated by this profile may be similar to those generated by other profiles, leading the models to classify those runs incorrectly. This is not a problem in the data but an inherent problem of random behavior. This is confirmed when comparing each model’s results on the random profile with the rest of the profiles. Other models to differentiate random behavior from the rest must be trained to cope with this problem.

Except for the random profile, all classes obtain similar scores with around 0.8–0.9 accuracy and f1-scores. With these results, we can conclude that the generation of synthetic data has successfully trained models capable of using the decisions taken to differentiate the behavior profiles.

4 Conclusions

Some key contributions of this work have been incorporated into a scientific article titled “A Child’s Play: An Agent-based Simulator to Protect Minors Online” [3], which has been accepted at a leading international conference within the field of computational social science. This article will be presented at the 8th International Conference on Computational Social Science (*IC²S²*) held at the University of Chicago (July 19–22, 2022).

Data generation is a technique that has been used for a long time to cope with the lack of real data and to improve the models trained on that data. We have provided a new approach suitable for decision-making games and expect that this work will open up the exploration of data generation for decision-making games.

We have presented a model to generate synthetic data in games where the focus is on decision-making rather than obtaining the highest score or where rewards are not applicable (no “good” or “bad” decisions). Our model generates synthetic data by creating agents that imitate real players’ behavior and gathering the data as they play the game. The agents are each given a behavior profile regarding several character traits they use for decision-making. The agents select the actions that best align with their profile’s character traits while also considering past actions. The behavior profiles used, the character traits that characterize them, and the number of past steps to consider are hyperparameters that allow the model to adjust to various applications and games. This approach does not require existing data. Therefore, it does not suffer from the intrinsic bias and lack of variability that arises from using this data. Furthermore, the agents do not follow a reward-driven approach since this would be unsuitable for games where the focus is on decision-making or rewards are not applicable. In addition, our model incorporates behavior information into the generated data and, therefore, into the models trained with the synthetic data. These models can then detect or analyze these behavior profiles on real data. In other words, our model transforms the task of analyzing the real players’ behavior from an unsupervised into a supervised task.

We have illustrated the model’s application and advantages by implemented it on an existing serious text game. The synthetic dataset generated is then used to train various Machine Learning models which are given as input the decisions taken on each run and are trained to predict the behavior profile that generated it. The results obtained by the models are substantially better than random (see Table 1), reaching global scores of 0.8 accuracy and f1-score.

There are various lines of future work which are improving the label predictor to adjust better to the player’s perception, improving the decision maker memory to adjust better to the player’s perception of the past, decreasing implementation time, testing the model in other decision-making games apart from text games, and testing the model

with other text games and on a real scenario. Each of these lines are intended to either increase the model's capabilities or test its application in a wider range of scenarios including real scenarios.

References

- [1] Marc-Alexandre Côté et al. "TextWorld: A Learning Environment for Text-based Games". Version 2. In: (2019). DOI: [10.48550/ARXIV.1806.11532](https://arxiv.org/abs/1806.11532). URL: <https://arxiv.org/abs/1806.11532> (visited on 05/31/2022) (cit. on p. 8).
- [2] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". Version 2. In: (2018). DOI: [10.48550/ARXIV.1810.04805](https://arxiv.org/abs/1810.04805). URL: <https://arxiv.org/abs/1810.04805> (visited on 06/05/2022) (cit. on p. 5).
- [3] Jaime Pérez et al. "A Child's Play: An Agent-based Simulator to Protect Minors Online". In: IC2S2 (8th International Conference on Computational Social Science). Chicago, 2022 (cit. on p. 12).
- [4] Robert Pfeiffer. *Yarn.Py, a Python Runtime for Yarn Dialogue*. Oct. 29, 2020. URL: <https://github.com/robertpfeiffer/yarn-pygame> (visited on 06/17/2022) (cit. on p. 8).
- [5] Ana Leticia Urbistondo Murua. "Creación de Un Videojuego Interactivo Para Educar En El Uso Responsable de Las Redes Sociales". BA thesis. Universidad Pontificia Comillas, Escuela Técnica Superior de Ingeniería (ICAI), 2020. URL: <https://repositorio.comillas.edu/xmlui/handle/11531/41473> (visited on 02/13/2022) (cit. on p. 8).



Bachelor's Degree in Telecommunications Engineering

Bachelor Thesis

**Text Games Synthetic Data Generation based on
Behavior Profiles**

Author

Vicente Balmaseda del Campo

Supervised by

Gregorio Ignacio López López

Jaime Pérez Sánchez

Madrid

July 2022

Abstract

Data generation is a technique used to cope with the lack of real data and to improve the models trained on that data. However, most of the techniques for data generation on games either require pre-existing data, thus usually suffering from bias and low variability, or are focused on obtaining high scores, which is not suitable for games where the focus is on decision-making or where rewards are not applicable, such as serious games. We present an architecture to generate synthetic data suitable for decision-making games. Our model tackles the above-stated limitations by creating agents that imitate real players' behavior and gathering the data as they play the game. Therefore, our approach does not require pre-existing data. The agents are each given a behavior profile regarding several character traits they use for decision-making. The agents select the actions that best align with their profile's character traits while also considering past actions. In addition, this architecture incorporates behavior information into the generated data and, therefore, into the models trained with it. We illustrate this architecture's application by implementing it on a serious text game aimed at educating about cybercrimes, particularly *Online Grooming*. We generate a synthetic dataset for the game and train various Machine Learning models on it. The generated data successfully teaches these models to predict the behavior profile that generated each game run given as input the decisions taken. These models can then be used to find the degree of similarity of the real players' behavior with respect to the given profiles.

Keywords: artificial intelligence, behavioral analysis, machine learning, natural language processing, serious (text-based) games, synthetic data generation

*The code is available on <https://github.com/vibalcam/text-games>.

Contents

1	Introduction	1
1.1	Project Definition	3
1.2	Contribution	4
1.3	Organization	4
2	State of the Art	6
3	Notions and Preliminaries	9
3.1	Graph Theory Basics	9
3.1.1	Directed Acyclic Graph (DAG)	9
3.1.2	Directed Tree	10
3.1.3	Decision Graph	10
3.2	Natural Language Processing Models	11
3.2.1	Long Short-Term Memory (LSTM)	12
3.2.2	BERT	14
4	Text-Based Games Modeling	16
4.1	Text Games	16

4.2	Architecture Overview	17
4.2.1	Components	17
4.2.2	Interactions	18
5	Architecture	20
5.1	Environment	21
5.1.1	Main Methods	21
5.1.2	Implementing the Environment	21
5.2	Agent	22
5.2.1	Main Methods	23
5.2.2	NLP-Based Label Predictor	24
5.2.3	Decision Maker	28
5.3	Modules	46
5.3.1	Graph Simulator	46
5.3.2	HTML Webpage Generation	46
5.3.3	Compound Agents	47
6	Implementation	48
6.1	Simulator	48
6.1.1	Yarn Spinner Format	48
6.1.2	Yarn Parser	49
6.1.3	Decision Graph	50
6.1.4	HTML Webpage Generation	52
6.2	NLP-Based Label Predictor	54

6.2.1	Labeling	55
6.2.2	Data	56
6.2.3	Model	56
6.2.4	Training	57
6.2.5	Results	59
6.3	Decision Maker	61
6.4	Synthetic Dataset	64
7	Results	65
7.1	Model Training and Analysis	65
8	Conclusions and Future Work	69
8.1	Conclusions	69
8.2	Future Work	71
A	Alignment with the Sustainable Development Goals (SDGs)	77
B	Decision Graph of the Online Grooming Game	

List of Figures

2.1	RNN (Left) and its unfolded representation (Right) Source [14] . . .	7
3.1	Directed vs undirected graph	10
3.2	An example of a directed acyclical graph (DAG)	10
3.3	An example of a directed tree	11
3.4	LSTM module Source [14]	12
3.5	LSTM cell state Source [14]	13
3.6	LSTM forget gate Source [14]	13
3.7	LSTM input gate Source [14]	14
3.8	LSTM output gate Source [14]	14
4.1	Environment and agent interactions diagram	18
5.1	Complete architecture diagram	20
5.2	Agent and its subcomponents diagram	23
5.3	Model for the NLP-based label predictor	26
5.4	Example: Safe behavior per iteration	37
5.5	Example: Risky behavior per iteration	38

5.6	Example: Random behavior per iteration	39
5.7	Example: Safe NTM behavior per iteration	42
5.8	Example: Risky NTM behavior per iteration	45
5.9	Example: Wrong Risky NTM behavior per iteration	45
6.1	Graph narrative obtained from the game	53
6.2	Original narrative of the game [22]	54
6.3	Example generated HTML document	55
6.4	Label predictor models comparison between pretrained BERT (right) and LSTM (left)	58

List of Tables

6.1	Size of training, validation and test sets	59
6.2	Training hyper-parameters	59
6.3	Label predictor LSTM vs BERT accuracy	60
6.4	Label predictor BERT vs BERT with negative augmentation	60
6.5	Selected label predictor model hyper-parameters	61
6.6	Label predictor certainty statistics estimated over the actions	61
6.7	Example synthetic data collected	64
7.1	Training dataset: choices taken per run	66
7.2	Behavior profile classification models test metrics	67
7.3	Example prediction: Probabilities of run being generated by a policy	68

List of Algorithms

1	Decision making process	35
2	Decision Graph using Depth First Search	51

1 | Introduction

Text-based games are textual turn-based single-player games. Each turn, the player is presented with a description of the context or current state and a list of available actions, both given by their natural language textual representation. The player then chooses an action that changes the game state, producing a new state description and a new set of actions. The game ends when the player reaches a terminal state. Therefore, players in the game try to find an optimal policy that will provide them with the best ending result.

Text games have other applications besides entertainment. Some of these applications are educational purposes such as improving reading behavior [8], behavioral analysis [5], or even research in the Social Science, Natural Language Processing, or Reinforcement Learning fields. The European H2020 [RAYUELA](#) project, in which this work is framed, is developing an interactive story-like serious games whose objective is twofold: on the one side, it will allow minors to learn good practices online while playing; and, on the other side, the data gathered through the game will be analyzed to try to identify potential risk factors so that Law Enforcement Agencies can develop evidence-based policies to protect minors online.

Text games' can take advantage of their entertaining ability to gather data. This data is typically about the players' decisions and behavior in-game and can be used for many tasks such as analyzing the players' behavior. Whichever the case, the data gathered is finally consumed by a model. These models must be calibrated, pre-tuned, and tested for quick and successful deployment. However, these tasks (and many others) cannot be carried out without data, which may be unavailable at the time, insufficient, or even simply unobtainable or prohibitively expensive. This lack of data leads to time delays. It can even lead to the loss of modeling performance or the impossibility of using the intended architecture due to a lack of data. The consequences could even expand beyond modeling, affecting the project in its totality. However, this lack of data can be palliated by generating synthetic data to expand the dataset and using it as a temporal or

permanent substitute for real data.

Existing techniques generate data by taking advantage of available real data (or data information such as its distribution) or simulating policy-based agents playing. Techniques based on available data suffer from bias towards their training data, generating synthetic data with low variance. Moreover, in many situations, real data is not previously available or, even if available, existing data from a game can not always be transferred to other games. A further drawback arises when using serious games to analyze the players' behavior or certain character aspects. Since these techniques do not consider any additional information, training models on this task can only be done unsupervised, with the difficulties that this entails. The same applies to models trained with the generated synthetic data and used on real data. The key to alleviate these difficulties is to make this task supervised by using policy-based agents. By doing so, the information about which policy generated what data can be used to train the models in a supervised manner.

The stated difficulties can be tackled by generating synthetic data by simulating agents playing the game. Existing techniques, such as those based on policies or Reinforcement Learning (RL), generally focus on finding agents that beat the game and obtain the highest possible scores. However, the synthetic data generated by this approach is utterly biased by the rewards given to the technique and does not have much variance. Moreover, this approach does not provide adequate results in games where the focus is not on winning but on decision making, such as the serious games used for educational purposes or behavioral analysis. In these games, decisions are not "good" or "bad", making rewards meaningless and synthetic agents based on rewards unsuitable. Since the focus is on decision-making, synthetic agents used for data generation must behave as players would, which entails modeling these agents according to the behavior of real players. Therefore, generating data for serious games requires a new approach based on the behavior profiles of the real players, making this task an interdisciplinary problem. Moreover, this approach must also tackle the considerable added difficulties that come with text games.

Players in games use the given information to make decisions according to what they believe best, choosing the actions that best align with their character traits. The same applies to synthetic agents based on behavior profiles. These agents analyze the information given in the game to obtain a set of characteristics or labels that they will then use to choose the actions that best align with their given character traits. However, in text games, this information is given in the form of natural language text, which entails dealing with Natural Language Processing (NLP). NLP is complicated due to data complexity, having to deal with context,

the many ambiguities, and many other complexities of text and human language. This adds a new layer that hinders the task of generating synthetic data.

Given the importance and benefits of generating synthetic data in text games when facing data shortage, we have developed a model that tackles the stated challenges and can be used in various scenarios and games without the need for existing real data. Moreover, it considers the task’s interdisciplinary nature and eases collaboration.

1.1 Project Definition

This work proposes a model to generate synthetic data in games, especially serious games, where the focus is on decision-making rather than obtaining the highest score or where rewards are not applicable. Since most decision-making games can be converted into text games, we will focus on these. However, our model can also deal with other kinds of information such as images or audio.

The model presented tackles the above-stated existing difficulties by meeting the following requirements:

- **Considers behavior profiles.** The model generates synthetic data in text games given several behavior profiles with regard to a set of character traits¹.
- **Independent from existing data.** The model does not require pre-existing real data.
- **Reusability.** The model allows reusing components, minimizing the effort needed to adapt it to a new text game or change the behavior profiles.
- **Supervised training of behavior profiles.** Generated synthetic data can be used to train models to analyze the behavior profiles of the players in a supervised manner. Through this process, the information given by the behavior profiles can be incorporated into the models and used to detect or analyze them on real data.
- **Task segregation.** The model eases interdisciplinary collaboration by separating the story, NLP, and policy (behavior profile) modeling subtasks into independent components.

¹By character traits we are referring to the a set of behavioral traits or qualities that define one’s type of person. Active, calm, careful and daring are some examples of character traits.

Apart from generating synthetic data, following the proposed model brings other benefits. A decision graph and HTML format can be readily obtained for the story. The NLP model’s learning can give valuable insights into any biases the game might have. It can help enhance the game by ensuring no unwanted relations are made between the state or actions and their consequences.

The following intermediate objectives are accomplished throughout this work to meet the requirements mentioned above and to illustrate the model’s application:

- **Development and formal definition** of the model following the requirements stated above.
- **Adapt an existing text game** to illustrate the model’s features and the process followed for its application. This entails labeling how well the actions align with a set of character traits if the game is not already labeled.
- **Build** and train an **NLP model** that the synthetic agents will use to extract the required information for decision making.
- **Generate** a number of **synthetic agents** to illustrate the results obtained.
- **Use the generated synthetic data** to build and train a simple Machine Learning model to illustrate the results obtained.

1.2 Contribution

This work contributes to the literature by presenting an architecture to generate synthetic data in games where the focus is on decision-making rather than obtaining the highest score or where rewards are not applicable. In contrast with previous techniques, it does not require existing data and is not focussed on obtaining the best ending, approach which is not suitable for decision-making games.

1.3 Organization

The remainder of this work is structured as follows. Ch. 2 reviews prior research done on the treatment of serious text games and the generation of synthetic agents and data on them. Ch. 3 explains the main notions and preliminaries about the

techniques used throughout this work. The next chapters present our proposed model. Ch. 4 presents an overview of the architecture by presenting its main components and their interactions. It also provides a mapping between each component and the aspect of the text games which it models. Ch. 5 dives deeper into the architecture, providing a detailed explanation of each component. This architecture is then illustrated in Ch. 6 by implementing it on an existing serious text game aimed at educating about cybercrimes, particularly *Online Grooming*. In Ch. 7 we generate a synthetic dataset for the game and use this data to train a variety of Machine Learning models on classifying each game run by the behavior profile that generated it. We also illustrate how this models can be used on real data to find the degree of similarity of the players' behavior with respect to the given behavior profiles. Conclusions and future work can be found at the end in Ch. 8.

2 | State of the Art

In this section, we will conduct a very synthetic review of state of the art on the treatment of serious text games and the generation of synthetic agents and data on them to show the research gap in the literature that our work is filling up, as stated previously.

Markov Chain Agents on adventure text games can be modeled as Markov Decision Processes. Agents make decisions that transition the game to a new state. Thus, we can represent these transitions using Markov chains, a stochastic process that models a finite set of states with fixed conditional probabilities of jumping from one state to another. These probabilities can be expressed using a transition matrix which can then be used to make decisions. Markov chains have also been used in Natural Language Processing (NLP) to generate text [20, 21]. Some modifications of the Markov chains, such as Hidden Markov chains [9], improve the basic model. However, Markov Chains rely on hard assumptions that are not always true, which leads to significant limitations. The state transitions solely depend on the current state, without considering past decisions or context. This is a significant flaw when trying to model a decision-making agent or dealing with an NLP task, where the context and past can have a significant impact. Moreover, when dealing with NLP, having a pre-defined finite and fixed number of steps leads to rigid and complex structures. This also makes the model highly dependent on the training data and can lead to huge transition matrices if the number of states is too big (something not difficult to occur when referring to language).

Reinforcement Learning When dealing with games, Reinforcement Learning (RL) is used extensively. Different models can be used to train agents to play the games. One of these models is the Deep Q-Network (DQN) [11, 15], which is based on Q-learning [24]. For this model to work, we must have return values (rewards) that express whether the player is on the right path. The model tries to approximate a Q-function which gives the expected return of taking each action

given the current input. This approach is valid when training agents to obtain the best score possible.

Adventure text games must also tackle the NLP task since the information is given in its natural language text representations. NLP tasks have proven to be more evasive to RL techniques [13]. This is due to the large number of words and phrases that make up the action space in natural language domains. Additionally, understanding the current state of a conversation in the natural language domain is more complicated than looking at a chess board or reading a robot’s sensors. Therefore, agents in text games need to incorporate architectures capable of dealing with NLP.

Deep Learning for NLP Deep Learning has been gaining popularity in the last years. Its flexibility allows tackling a wide variety of tasks, and NLP is no exception. Moreover, Deep Learning has achieved state-of-the-art results in NLP tasks with the appearance of the Transformer architecture [23]. Deep Learning architectures are especially good at treating NLP tasks when there is a significant amount of data available and a large state space. One of the most well-known architectures that solves the flaws of Markov Chains are Recurrent Neural Networks (RNN) [19]. In RNNs, there is no limit on the number of states, and past states are considered through a hidden state passed from one layer to the next, as presented in Fig. 2.1.

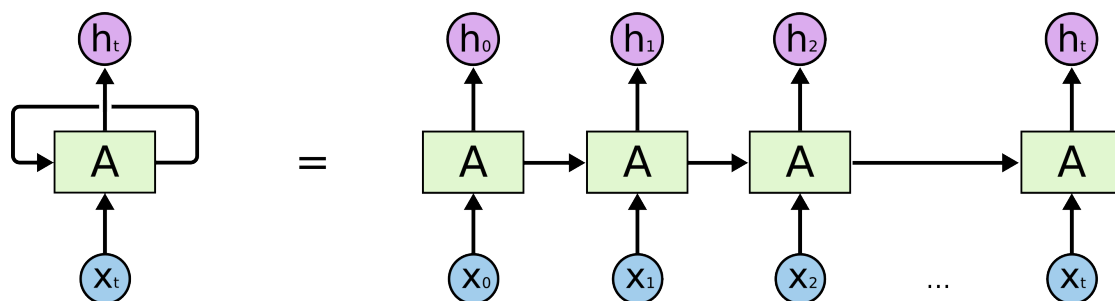


Figure 2.1: RNN (Left) and its unfolded representation (Right)
Source [14]

However, RNNs also have their shortcomings. They need a significant quantity of data to perform well, and they cannot remember states too far in the past. Other Neural Networks architectures, such as Causal Convolutional Neural Networks (Causal CNN), can alleviate this last flaw, thus increasing the model’s capabilities to remember information.

These Deep Learning models can be combined with RL agents to work on text

games. Research has been done in this field, using bag-of-words, LSTMs (a kind of RNN), and Bidirectional LSTMs to train RL agents [12]. These RL agents extract information from the text in the game and learn a set of control policies capable of generalizing. While the idea of using agents that maximize their rewards provides a successful approach to model players, RL agents maximize the final rewards (the sum of rewards) instead of maximizing each individual reward. This focus on the final result instead of decision-making is not as practical for our purpose in serious text games.

Other Research. Other research on text games has developed frameworks to ease handcrafting or automating the generation of games with the difficulty, scope, and language that the researcher requires [1]. This is especially useful to train models on multiple games that share the same dictionary (or the same field) or to gradually train a model to improve the model’s generalization capabilities.

Synthetic Data Generation. Research has also been conducted on generating synthetic data. When the accessible data is insufficient due to unavailability or our inability to produce more (for example, due to privacy or being prohibitively expensive), these models allow us to artificially create data that we can use as a substitute for real data. There are some naïve techniques, such as generating or fitting data according to a known distribution. Deep learning has provided better and more complex techniques such as Generative Adversarial Networks (GAN) [4] and Variational Autoencoders (VAE) [3]. However, these techniques require access to a certain amount of real data, and the quantities required are sometimes too large to be practical. Moreover, the data generated has limited variability and is biased by the training data. This is even more true when using distribution information. These limitations hinder their applicability for our purpose since we do not want only to expand a current dataset with similar data but to generate new synthetic data similar to that generated by certain behavioral profiles without requiring pre-existing data.

The model that this work presents tackles these difficulties, allowing for the generation of synthetic data even when no data is available and considering the behavior profiles of the players. This makes our model suitable to generate synthetic data for text games where no existing data is available or where the focus is on decision-making rather than obtaining the highest score possible.

3 | Notions and Preliminaries

This section provides a superficial overview of some of the techniques we have used throughout this work.

3.1 Graph Theory Basics

A graph is a data structure consisting of a set of objects in which some pairs are somehow related. These objects are called nodes (vertices), and the relations between them are represented by edges. Thus, a graph $G = (V, E)$ is a set of nodes V connected by edges, with E being the set of all edges.

Graphs represent a network. We say that the graph is directed if the nodes are related through directional dependencies. If not, the graph is undirected (Fig. 3.1). In undirected graphs, the degree of a node is the number of edges that have that node as a vertex.

3.1.1 Directed Acyclic Graph (DAG)

A directed acyclic graph (DAG) is a directed graph with no directed cycles. In other words, it is a graph with directed edges from one vertex to another, such that following those directions will never form a closed loop. An example of a DAG is presented in Fig. 3.2.

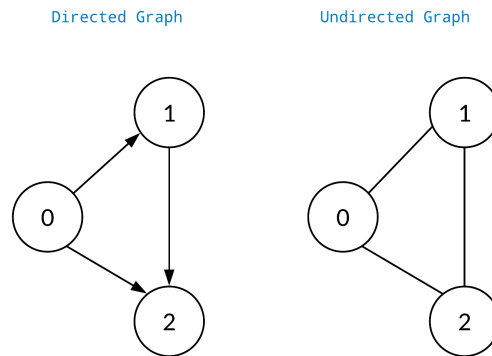


Figure 3.1: Directed vs undirected graph

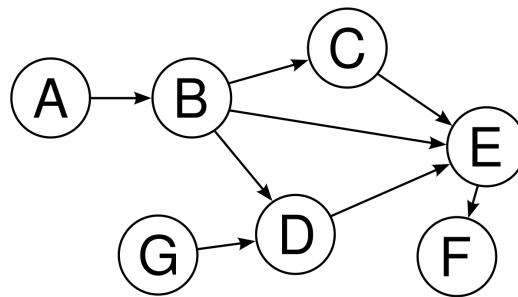


Figure 3.2: An example of a directed acyclical graph (DAG)

3.1.2 Directed Tree

A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, a tree is a connected acyclic undirected graph. A directed tree is a DAG whose underlying undirected graph is a tree. An example of a directed tree is presented in Fig. 3.3.

3.1.3 Decision Graph

The decision graph of a text-based game is a graph that represents the states and actions available in the game. The decision graph of a game can be obtained by:

- For each game state, a node in the graph is created and the state's informa-

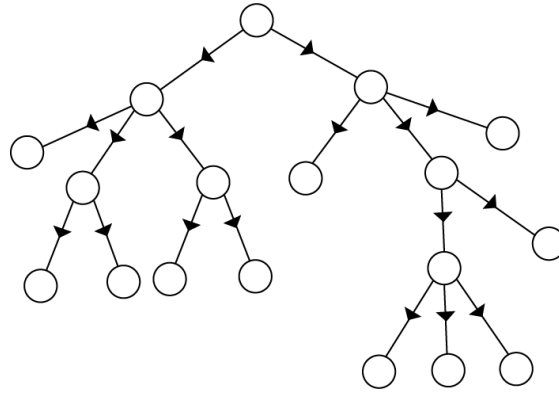


Figure 3.3: An example of a directed tree

tion added as a node feature.

- For each action that changes the game state, a directed edge between the source state and the destination state is created. The information about the action is added as an edge feature.

A decision graph is a data structure that can be used to represent a text-based game and that includes all the available information. In some situations, if the game does not have any loops, its decision graph is a decision tree.

3.2 Natural Language Processing Models

Natural Language Processing (NLP) is the ability of a computer program to understand and process human language. This includes analyzing, interpreting, and generating both spoken and written language. Speech recognition, text to speech (and visa-versa), and language translation are examples of tasks in the natural language domain.

In the last few years, this field has received a lot of attention and improved greatly thanks to the use of Deep Learning. Two of the most well-known Deep Learning algorithms used for NLP are LSTMs and Transformers.

3.2.1 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) [7] networks are a type of Recurrent Neural Networks (RNN). They were introduced to alleviate RNNs difficulties learning long-term dependencies. They share the same chain structure as RNNs, but the repeating module has been modified to ensure information is retained for long periods of time. The LSTM module can be divided into four layers that interact with each other, as presented in Fig. 3.4.

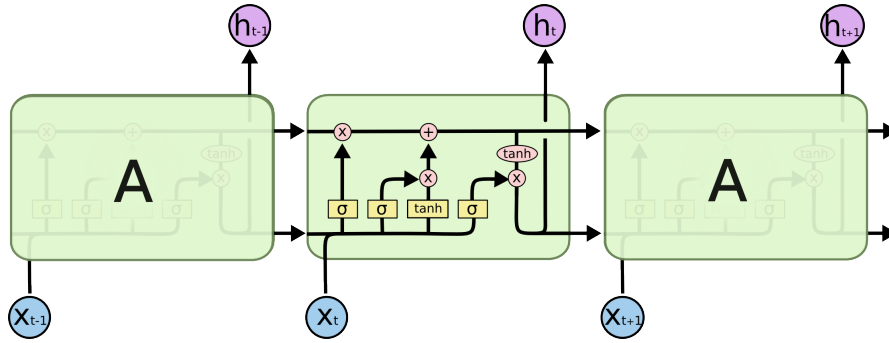


Figure 3.4: LSTM module
Source [14]

Cell State

The cell state contains the context information which passes from an iteration of the LSTM to the next with only minor modifications introduced by two gates. Gates are composed of a sigmoid layer and a pointwise multiplication, which allows the model to regulate how much information can go through the gate. Gates allow adding and removing information from the cell state.

Forget Gate

This gate f_t looks at the last hidden state h_{t-1} and the input x , and decides what and how much information from the cell state is kept. This gate is multiplied by the previous cell state C_{t-1} to remove unwanted information.

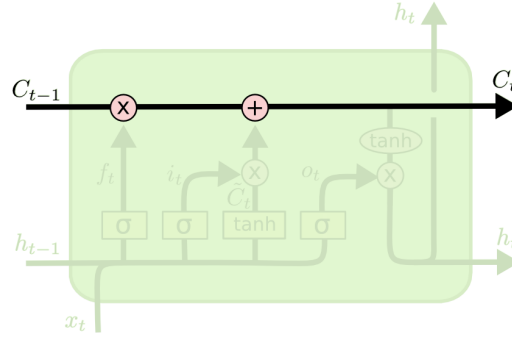
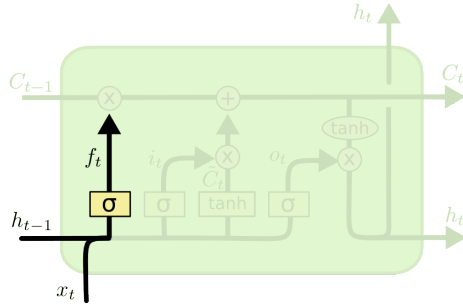


Figure 3.5: LSTM cell state
Source [14]



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Figure 3.6: LSTM forget gate
Source [14]

Input Gate

This gate i_t looks at the last hidden state h_{t-1} and the input x , and decides what new information is going to be stored in the cell gate. The potential new information $\tilde{C}_t$ is obtained by passing the last hidden state h_{t-1} and the input x through a $\tanh$ layer. The output of the $\tanh$ is multiplied by the input gate i_t and then added to the previous cell state (after the forget gate), thus obtaining the new cell state C_t .

Output Gate

This gate o_t looks at the last hidden state h_{t-1} and the input x , and filters the cell state to obtain the output of this iteration. The cell state C_t is passed through a $\tanh$ layer before being multiplied by the output gate o_t . The result of this

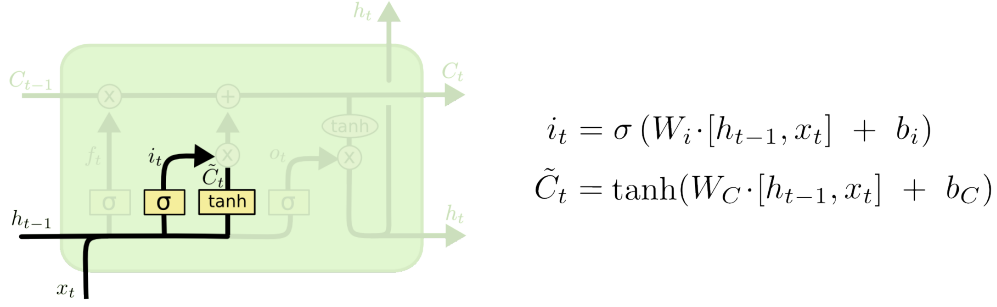


Figure 3.7: LSTM input gate
Source [14]

operation is this iteration's hidden state h_t .

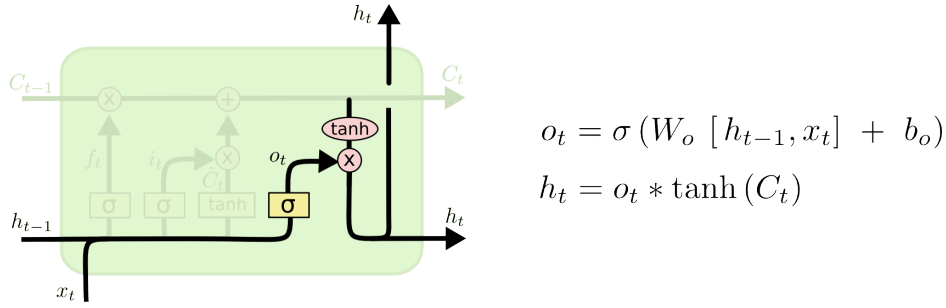


Figure 3.8: LSTM output gate
Source [14]

3.2.2 BERT

BERT (Bidirectional Encoder Representations from Transformers) [2] is a language model that obtained state-of-the-art results in a wide variety of NLP tasks and that has been widely used as a pre-trained model that can be fine-tuned to easily tackle a wide range of NLP tasks.

BERT is based on the Transformer architecture [23] and it is pre-trained on a large corpus of unlabelled text including the Wikipedia and the Brown Corpus. BERT has been pre-trained on two particular NLP tasks, Masked Language Modeling and Next Sentence Prediction. This pre-training is what gives BERT a deeper understanding on how the language works and makes it useful in such a wide variety of NLP tasks.

Some websites such as [Hugging Face](#) allow us to obtain a BERT model with its pre-trained weights in order to fine-tune it to our task. Sharing pre-trained language models is essential in reducing the overall computation cost and reduces the efforts needed to obtain state-of-the-art results on NLP tasks. Moreover, many BERT fine-tuned models are also available, which can even remove the necessity of training a model ourselves.

4 | Text-Based Games Modeling

This chapter first gives a brief introduction on text games (Sec. 4.1). It then continues by giving an overview of the architecture (Sec. 4.2) used to model them to generate synthetic data. It explains the different components of the architecture, why they are needed, their responsibilities, and how they interact with each other.

4.1 Text Games

Text-based games, also known as "Interactive Fiction," are textual turn-based single-player games. Each turn, the player is presented with the context or current state and a list of available actions, both given by their natural language textual description. The player then chooses an action that changes the game state, producing a new state description and a new set of actions. The game ends when the player reaches a terminal state. More formally, text games are sequential decision-making tasks with both state and action spaces given in natural language. Therefore, the task of learning to play text games is closely related to Natural Language Processing.

Generally, there are several terminal states with different endings, which can be sorted in order of preference. This order can be absolute, the same for all players, or it can depend on each player's personal preference. Each of the actions/states in the game can be assigned labels or scores (rewards) that provide more information about them and how they impact the possible endings.

The game players try to find an optimal policy that will provide the best ending for them. A policy is a strategy that a player, or agent, uses in pursuit of his goals. Formally, a policy $\pi : s \rightarrow a$ is a function that returns a possible action given the game context or state. Therefore, a player's policy dictates the actions it takes.

The players modify their policy as they learn to play the game.

4.2 Architecture Overview

As discussed in Ch. 1, this work generates synthetic data based on the hypothesis that players use the given information to make decisions according to what they believe to be best. Moreover, we discussed that the approaches based on existing data have significant downsides, which lead us to use policy-driven synthetic agents based on the behavior profiles of real players.

Therefore, the main idea behind the proposed model is to use the given behavior profiles to create synthetic agents that will play the game. While the agents play the game, synthetic data will be collected as if they were real players. By doing so, we are generating data with the agents' policy implementing the behavior profiles and without using pre-existing data (as per the requirements in Sec. 1.1).

Even though the model specifies how the agent extracts information from the text game and how its behavior profile impacts the agent's decisions, it is designed with a set of flexible components that can be modified, or even substituted by a self-developed component, to adapt to future changes. Our model can use these components as long as they adopt an interface of interactions between them. This approach also increases our model's flexibility and modularity, allowing components to be reused by only substituting the parts that have changed.

4.2.1 Components

The model presented is divided into two main components, the simulator or environment, and the agent or player. Each of these components are independent and interact with each other through an interface. This modular approach eases interdisciplinary collaboration by separating the story, NLP, and policy (behavior profile) modeling subtasks into independent components.

- **Environment.** The environment takes care of the game's storyline and is the player's counterpart. It provides the agent with information about the current game state and available actions, both given by their natural language text representation. It then receives the action taken by the agent, processes it, saves the game state, and advances in the story. This component

is responsible for following the thread of the story and how the agent's actions affect it. This component has been modeled with the simulator, which is presented in Sec. 5.1.

- **Agent.** The agent represents the player of the game. It receives the current state and the available actions from the environment, processes the information, and chooses an action following a specific policy. It then reports it to the environment to advance in the story. Thus, this component is responsible for the NLP and policy subtasks. This component has been modeled with the agent, presented in Sec. 5.2.

4.2.2 Interactions

The components presented above interact with each other in various ways. These interactions are the only hard requirement of our model that even self-developed components must meet. Fig. 4.1 presents a diagram of the components and how they interact with each other.

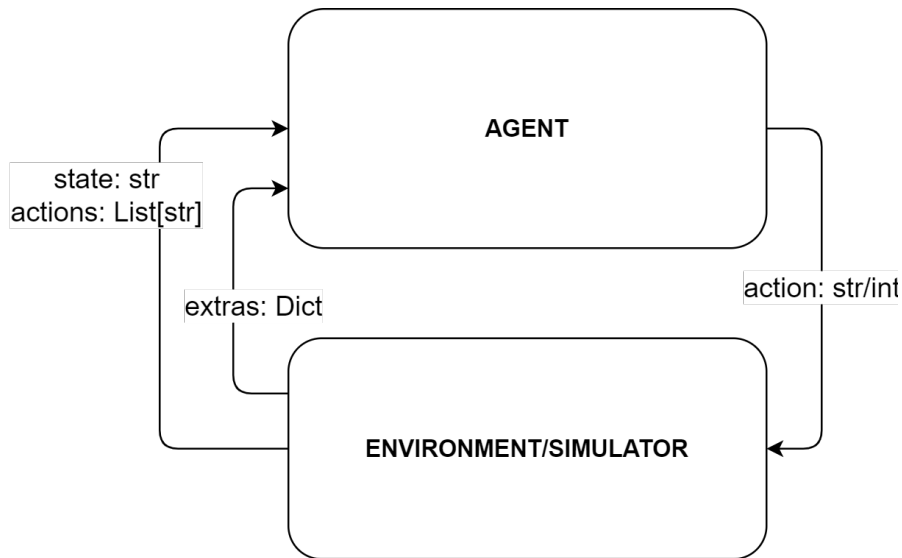


Figure 4.1: Environment and agent interactions diagram

Environment $\rightarrow$ agent

- **State** (*str*). The current context or game state, given by its text representation.

- **Actions** (*list of str*). The list of available actions the player can choose to take, given with its text representation.
- **Extras** (*dictionary*). A dictionary of key-value pairs with additional information for the agent, such as images in the text game, or information not intended to be used by the agent while playing, such as information for training the agent or data analytics.

Agent $\rightarrow$ environment

- **Action** (*str/int*). The action taken by the player. It can be given as the text representation of the action or as its position in the list of available actions.

5 | Architecture

This chapter dives deeper into the architecture used to model the text games for synthetic data generation. We first present the environment component (Sec. 5.1) and state how to use the game’s decision graph to implement this component readily. We then move into the agent component (Sec. 5.2). We present the two subcomponents in which the agent is divided, the label predictor and the decision maker, and explain why this separation is needed. We show how the label predictor (Sec. 5.2.2) tackles the extraction of information from the text game. Finally, we present the decision maker (Sec. 5.2.3), which models the agent’s decision-making, thus providing the agent’s policy.

Fig. 5.1 gives an overview of the complete architecture, showing the three main blocks discussed in this section.

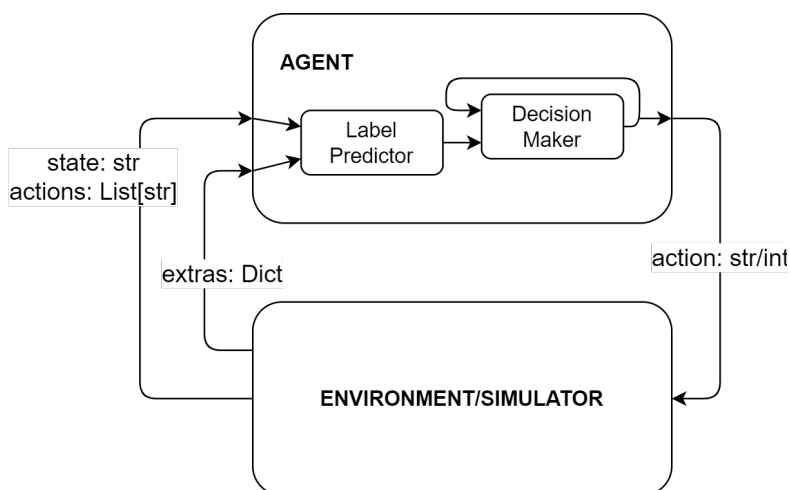


Figure 5.1: Complete architecture diagram

5.1 Environment

The environment or simulator component is the representation of the story. This component can be implemented in multiple ways, from a graphical interface for real players to a programming interface for easy scripting. The environment is responsible for following the thread of the story and how the agent's actions affect it. This involves storing the game state, actions taken, rewards obtained (if any), and anything related to the story.

The environment is modeled with the simulator. In this section, we will present its characteristics.

5.1.1 Main Methods

- **Read** ($void \rightarrow str, list, dictionary$). This method returns the current output of the simulator as stated in Sec. 4.2.2. In other words, it allows accessing the current state, list of actions and extras.
- **Transition** ($str/int \rightarrow str, list, dictionary$). Method used to report the action chosen by the agent to the simulator. It returns the same values as the read method.

5.1.2 Implementing the Environment

The environment is implemented by adapting the story to comply with the component's requirements. In many cases, a parser is needed to adapt the story's original format to this component. Whichever the case, this component can be generalized by using a decision graph. A decision graph is a graph that holds all the required information for the story. It is created as follows:

- **Nodes.** The nodes are the story states. They might have node attributes to hold information related to the states, such as rewards or their text representation.
- **Edges.** The edges are the actions that can be taken from a given state to the next. They might have edge attributes to hold information related to the

actions, such as its text representation or the character traits aligned with the action.

5.2 Agent

The agent is the representation of the players. Thus, it simulates the behavior of a person playing the game. The players in the game process thoughtfully the information received from the environment and select the action that they believe produces the best impact on the story. Our model of the players, the agents, will follow the same approach, making thoughtful decisions to get the best results. However, this leaves us to define what the best result is. In reward-driven games, the best result is obtained when we arrive at the terminal state with a high sum of rewards. However, this approach assumes players have information about the score they receive (or the impact their actions have on the ending) and play the game multiple times to learn what combinations achieve the best ending. These assumptions do not hold well when trying to model real players to generate synthetic data in decision-making games, such as serious games, because of the following:

- Players in serious games often do not have reward information throughout the game or even at the terminal state, but rather have a subjective feeling on how good the results they are obtaining are for them.
- Serious games often do not have "good" or "bad" actions. Therefore, there are no rewards per se. Instead, actions can be labeled to certain character traits, which relates to the fact that the same action can be good for a player while being bad for another. An example could be whether to take a risk or not.
- In serious games, we are normally interested in gathering data about how players make decisions. This information is mostly obtained on the first few attempts playing the game since, after a few attempts, the players start memorizing the impact of their decisions and trying other options to obtain different results.

Therefore, the reward-driven approach is not suitable. Instead, we assume the agents do not have information about the impact of their actions on the endings and try to maximize their subjective scores for every action, taking the actions they consider best.

Following these assumptions, we give each agent a specific behavior profile. The agents select the actions that best align with the character traits of their behavior profile. In other words, in each step, the agents select the action that they consider best according to their behavior profile. While past decisions can influence the agent's current actions, future decisions can not since we assume the agents do not know what future states and actions will be available.

With this in mind, the agents are divided into two subcomponents. Each of these components takes care of a different task in the decision-making process. First, the *label predictor* (Sec. 5.2.2) will process the information received from the simulator to give information about whether the character traits (also known as labels from now on) align with a particular action. Then the *decision maker* (Sec. 5.2.3) will use the extracted labels to choose the action that the agent considers to be best according to its behavior profile. Therefore, the label predictor will be responsible for the NLP task, and the decision maker will implement the agent's policy according to a given behavior profile.

The remainder of this section will first give an overview of the agent's input, output, and main methods. We will then dive deeper into how the agent's subcomponents fulfill their roles and interact with each other to model a real player. Fig. 5.2 presents a diagram of the agent component and its subcomponents. This diagram summarizes the agent's input, output, and what information is passed from one subcomponent to the next.

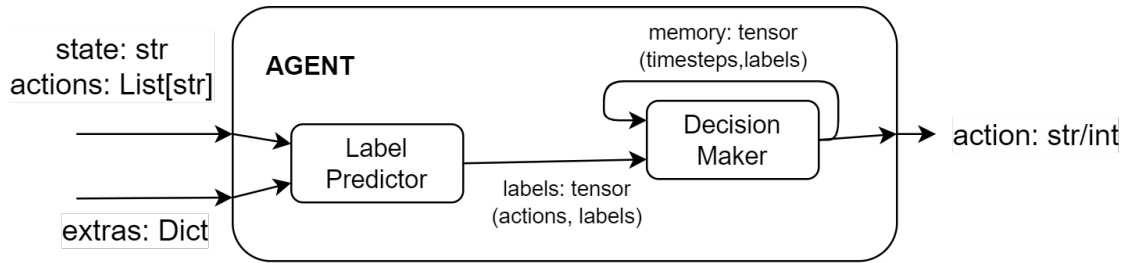


Figure 5.2: Agent and its subcomponents diagram

5.2.1 Main Methods

- **Act** ($state, \dots \rightarrow str/int$). Method used to provide the agent with the current state and other information, such as the list of actions, for decision making. It returns the action chosen by the agent as stated in Sec. 4.2.2.

5.2.2 NLP-Based Label Predictor

The NLP-based label predictor subcomponent will be responsible for the first step of the agent’s decision-making process. This first step involves processing the information given by the game and obtaining the degree of alignment between the actions and the character traits, or labels, that define the behavior profiles. This information is later used in Sec. 5.2.3 to select which actions to take.

Extracting a set $\{l_1, \dots, l_J\}$ of J predefined labels (character traits) can be seen as a multi-label classification task. Therefore, the input and output of the label predictor subcomponent will be:

- **Input.** The same as the agent’s input.
 - **State** (*str*)
 - **Actions** (*list of str*). List $[a_1, \dots, a_I]$ of actions. The length of the list I is not fixed and might change from one game state to the next.
 - **Extras** (*dictionary*). Optional, can be used in some text games with additional information such as images.
- **Output.** 2-dimensional $I \times J$ tensor H such that the element $h_{i,j}$ is the probability that action a_i aligns with the character trait l_j .
 - **Labels** (*tensor (actions, labels)*).

A naïve approach could be manually providing these labels. However, doing so would bias the result towards what we believe the actions entail. These labels do not have to be exact nor what we believe each action entails, but rather what the players will believe it entails when they reach it. Therefore, these labels will depend not only on the current context and action being presented but also on the whole context of the game. A way to avoid this bias is by using a pre-trained NLP model as a proxy for the label extraction. However, finding a model trained on a vocabulary similar to the one used by the game at hand (such as cyberbullying) that predicts the character traits of an action can be tricky. Instead, we can build and train a model for this task. Therefore, we will pre-label each action with a set of character traits (the same set we used to define the behavior profiles) and, instead of using them directly, train a multi-label classification model. Since training an NLP model is usually expensive and requires a high quantity of data, we will fine-tune a pre-trained model such as BERT. If possible, we will use multiple games

or stories with similar vocabularies to fine-tune our model to reduce bias towards the game.

Next, we present an architecture that can be used to extract the required labels. This architecture allows using a pre-trained NLP model, such as BERT, and adjusting to the story's set of actions, states, and additional information. Moreover, this architecture can incorporate any information the text game provides. A diagram of the architecture using only states and action text information is presented in Fig. 5.3. This diagram also shows the inputs and outputs of each layer. This architecture receives as input state-action pairs and returns the probability tensor that will be the output of the label predictor. It consists of four main layers: a tokenizer, shared layer, individual layer, and output layer. The state and list of actions given as input to the label predictor subcomponent $(s, [a_1, \dots, a_I])$ will pass through each layer.

Tokenizer

The tokenizer is the input layer for text information. This layer will receive state-action pairs. Therefore, given the following state and list of actions input to the label predictor subcomponent $(s, [a_1, \dots, a_I])$, the input of the tokenizer will be $[(s, a_1), \dots, (s, a_I)]$.

This layer will run separately over the states and the actions. It will first tokenize the text information and adjust the length to comply with the maximum admitted length by the shared layer. This implies two operations:

- Add padding if needed for batches.
- If the length of a sentence exceeds the maximum, truncate the sentence keeping the tokens starting from the right making sure we do not split a sentence in two.

The tokenizer used depends on the pre-trained model that is being used. For example, when using a BERT pre-trained model, we need to use a tokenizer for BERT.

The tokenizer returns a list of tokens for the state and a list of tokens for the action.

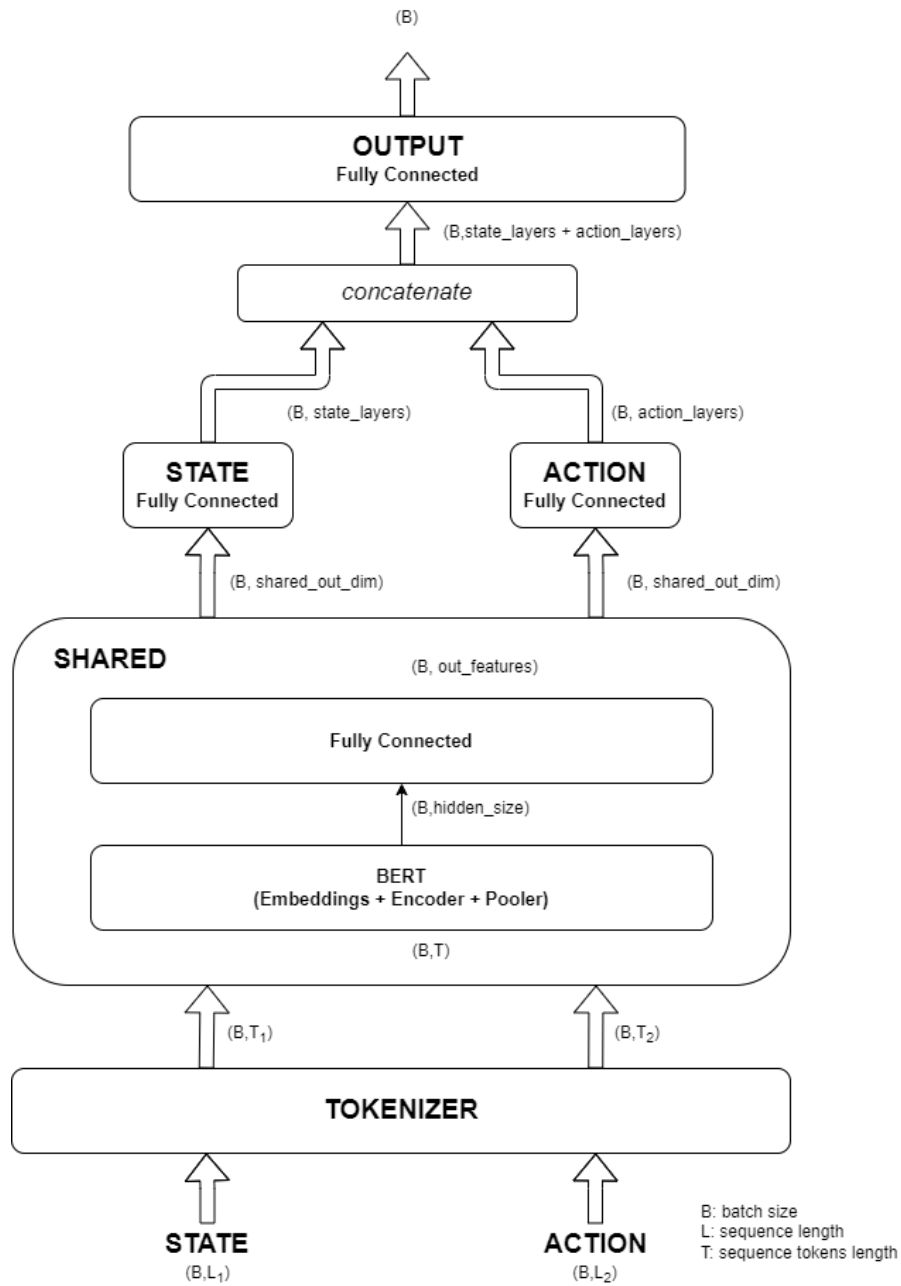


Figure 5.3: Model for the NLP-based label predictor

Shared Layer

The shared layer is shared by the state s and action a inputs and will process each of these inputs separately. This layer receives a list of tokens and returns a representation of the information in them. Therefore, it is responsible for understanding the meaning of the sentences, for which it will take advantage of transfer learning by incorporating a pre-trained model. Fig. 5.3 presents a BERT pre-trained model with the embeddings, encoder, and pooler layers, though other pre-trained models can be used.

When possible, this layer will be fine-tuned as a whole. However, pre-trained models are usually too big to be trained on a personal computer. In this situation, we can freeze the pre-trained model and incorporate a fully connected neural network (FCNN) to fine-tune it, as demonstrated in Fig. 5.3.

The shared layer outputs a vector of a predefined dimension (*shared_out_dim*) in which the information is embedded.

Individual Layer

The individual layer is comprised of multiple blocks. This layer has a block for each kind of information, such as states, actions, or images given as extra. The composition of each of these blocks can vary depending on the type of information being processed. For example, in Fig. 5.3, both the state and action individual blocks are fully connected layers. However, if the information being processed are images given as extra, this block might be a Convolutional Neural Network (CNN).

Each individual block provides a deeper analysis and processing by focusing only on a single kind of information. Therefore, this layer extracts the essential information provided by a single input, which will later be combined in the output layer with the rest of the inputs to extract the labels.

The output of each of the individual layers will be a vector of a predefined dimension, which can be different for each individual block.

Output Layer

The output layer receives its input from the various individual layers, combines it, and returns the probabilities for the predefined set of labels $\{l_1, \dots, l_J\}$. It does so by concatenating all the given information (the outputs from the different individual layers) and passing it through an FCNN. It then passes the output through a sigmoid function to obtain probabilities.

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Therefore, the output of this layer will be the output required by the label predictor, a 2-dimensional tensor of (I, J) dimension (actions, labels).

Extending with Extras

In Fig. 5.3 we have only taken into account state and action information. We have done so because this is the minimum information given on text games, while additional information may or may not be given depending on the game. However, to consider additional information, an additional individual block would be added to process it. As mentioned above, the individual block used depends on the information. If the additional information is an image, a CNN could be used. If the additional information is text, it would first go through the tokenizer and shared layer and then proceed with a similar neural network as the one used for the state and action inputs.

The FCNN used, such as the one used for the output or individual layers, can be replaced by other architectures such as transformers or LSTMs.

5.2.3 Decision Maker

The decision maker subcomponent will be responsible for the second step of the agent's decision-making process. This second and last step involves using the extracted character traits, or labels, involved in each of the available actions to choose the action that best aligns with the agent's given behavior profile. Therefore, this subcomponent is the core of the decision-making process and is responsible for using the provided behavior profile to make decisions as the real player would.

The decision maker receives its input from the label predictor (Sec. 5.2.2) and outputs the action chosen by the agent (thus having the same output as the agent). Therefore, given a set $\{l_1, \dots, l_J\}$ of J predefined labels and a list $[a_1, \dots, a_I]$ of I actions, this subcomponent's input and output are the following:

- **Input.** The output of the label predictor.
 - **Labels.** (*tensor (actions, labels)*). 2-dimensional $I \times J$ tensor H of values such that the element $h_{i,j}$ is the probability that action a_i aligns with the character trait (or label) l_j .
- **Output.** The agent's output.
 - **Action** (*str/int*). The action taken by the agent.

The agents used to model the players are policy-based synthetic agents. The decision maker is responsible for the policy the agent follows, obtained from the given behavior profile. Thus, the decision maker must be able to characterize the behavior profile it has been given and implement the agent's policy in accordance with it. Moreover, the policy must consider the character traits used to define the behavior profile and past decisions. In this section, we will present a model that accomplishes these requirements, give some insights into the model's functioning, and provide a few illustrative examples.

Decision Maker Model

Given a behavior profile, the decision maker will assign a probability to each available action depending on how well that action's labels (character traits) align with the behavior profile's labels. These probabilities are obtained by calculating a weighted sum of all the labels, with the weights being given by the behavior profile. The behavior profile is characterized by a set of predefined functions (hyperparameter of the model) that quantify the alignment between that label and the agent's profile. The number of functions equals the number of character traits being considered. Therefore, these functions represent the relative importance the behavior profile gives to each of the character traits and is affected by the agent's subjective sentiment of the past (since, as mentioned in Sec. 5.2, the agent is not given reward information nor the true character traits throughout the game).

Formally, the decision maker is given a behavior profile characterized by a set $\mathbf{f} = [f_1, \dots, f_J]$ of functions and a number N of past steps to consider. Each

function $f_j : \mathcal{M}_{(N+1) \times J}(\mathbb{R}) \times \mathcal{M}_{N+1,1}(\mathbb{R}) \rightarrow \mathbb{R}$ returns the importance of the character trait $l_j \in \{l_1, \dots, l_J\}$ for decision making. This importance is given as a score. The higher the returned value, the higher the importance of l_j . If the importance is positive, the behavior profile aligns positively with l_j , favoring actions with that character trait. If it is negative, it disfavors actions with character trait l_j . These f_j functions take as input the N -steps memory matrix M and step vector S .

The memory matrix M is a $(N+1) \times J$ matrix where each row represents the step $t-n \in \{t, t-1, \dots, t-N\}$ (with t being the current step) and each column represents the character trait l_j . Thus, M is defined as

$$M = \begin{bmatrix} 1 & 1 & \dots & 1 \\ l_1^{t-1} & l_2^{t-2} & \dots & l_J^{t-1} \\ \vdots & \vdots & \ddots & \vdots \\ l_1^{t-N} & l_2^{t-N} & \dots & l_J^{t-N} \end{bmatrix} \quad (5.1)$$

where the element $m_{n+1,j} = l_j^{t-n}$ is the percentage by which the action taken by the agent in step $t-n$ aligns with the character trait l_j . The values given for step t (elements $m_{1,j}$) are 1 since the current decision has not yet been taken. Since the agent does not have access to the true labels, these values are the agent's perception about his past decisions. They are obtained from past predictions of the label predictor. Therefore, the memory consists of the agent's own recollection rather than a true value for all agents.

The step vector S is a column vector of size $N+1$ where each element equals 1 or 0. This vector gives information about the validity of the memory matrix. If s_t equals 1, this means that row M_{t*} of the memory matrix is valid. Therefore, $s_t = 1$ if a decision has been taken for step $t-1$, $s_t = 0$ otherwise. s_1 always equals 1 since row M_{1*} is always valid.

Then, f_j returns the importance w_j of the character trait l_j for the current decision.

$$w_j = f_j(M, S) \quad (5.2)$$

On each step t , the decision maker receives as input a matrix H such that the element $h_{i,j}$ is the probability that action $a_i \in \{a_1, \dots, a_I\}$ aligns with the character trait l_j . We consider $h_{i,j}$ to be the percentage by which action a_i aligns with the character trait l_j . Therefore, the vector of probabilities of taking a certain available action is given by

$$\mathbf{p}_a(H, \mathbf{w}) = \text{softmax}(H\mathbf{w}^T) \quad (5.3)$$

where $\mathbf{w} = [w_1, \dots, w_J]$.

Thus, the probability of taking action a_i is

$$\begin{aligned} \mathbf{p}_a(H, \mathbf{w})_i &= \frac{e^{\sum_{j=1}^J h_{i,j} w_j}}{\sum_{i=1}^I e^{\sum_{j=1}^J h_{i,j} w_j}} \\ &= \frac{e^{s_i}}{\sum_{i=1}^I e^{s_i}} \end{aligned}$$

where s_i is the score $\sum_{j=1}^J h_{i,j} w_j$ (weighted sum) given to action a_i .

By combining Equation (5.2) and (5.3), we arrive at the general decision-making expression Equation (5.4). This equation models how the probabilities of taking the actions in $\mathbf{a} = \{a_1, \dots, a_I\}$ are obtained from the agent's behavior profile, characterized by $\mathbf{f} = [f_1, \dots, f_J]$ (with $f_j : \mathcal{M}_{(N+1) \times J}(\mathbb{R}) \times \mathcal{M}_{N+1,1}(\mathbb{R}) \rightarrow \mathbb{R}$), and the inputs H ($I \times J$ labels matrix), M ($(N+1) \times J$ memory matrix), and S ($(N+1)$ step column vector).

$$\mathbf{p}_a(H, \mathbf{f}, M, S) = \text{softmax}(H\mathbf{f}(M, S)^T) \quad (5.4)$$

where $\mathbf{f}(M, S) = [f_1(M, S), \dots, f_J(M, S)]$.

The agent selects which action to take through a weighted random choice, with the weights being $\mathbf{p}_a$.

Explanation and Intuition

The first step in the process is to characterize the behavior profile for decision-making. The agent will choose the actions that best align with the character traits given by his behavior profile. This model proposes to score that alignment by giving weights to each character trait. The higher the weight, the more importance the behavior profile gives to a specific character trait being present in an action. These weights are what characterize the behavior profile. However, the behavior

profile should also be able to consider the past to set these weights, which requires making them dynamic. Instead of fixed weights, our model characterizing the behavior profile with a set of functions that depend on past decisions and return the dynamic weights used for decision making in a certain step t , as in Equation (5.2).

We have now successfully characterized the behavior profile. However, we have to specify what information from past decisions will be used. Following the discussion from Sec. 5.2, we assume that the agent does not receive the true reward information nor true character traits throughout the game and that it does not know in advance the consequences of each action in the game. Therefore, all the information available to the agent is its perception from past decisions. The label predictor provides the agent’s perception of the actions. Therefore, the model uses as a memory for decision-making its perception of the decisions it has taken¹. In other words, it will save the predicted character traits of the decisions it has taken for N past steps. The number of past steps N is a hyperparameter of the model. It is important to note that the importance of a given character trait l_j can be impacted by other labels l_k of past decisions apart from l_j . Algorithm 1 presents the iterative process for decision making and saving the memory matrix.

Let us now dive deeper into the meaning of the weights obtained in Equation (5.2). Since $h_{i,j} \in [0, 1]^{\mathbb{R}}$, the sign of the impact of the character traits on the action’s score solely depends on the weights as follows

- if $w_j > 0$, the agent’s behavior profile aligns with l_j , thus giving it positive importance. Increases the action’s score.
- if $w_j = 0$, the agent’s behavior profile is not influenced by l_j , thus ignoring it. No impact on the action’s score.
- if $w_j < 0$, the agent’s behavior profile aligns negatively with l_j , thus giving it negative importance. Decreases the action’s score.

Therefore, the sign of the weights $\mathbf{w}$ gives information on the model’s alignment with each character trait for the current decision. On the other hand, the

¹Another factor that can impact the agent’s perception is the context, or change in context, obtained after an action has been taken. By analyzing the context after an action, the agent perceives the consequences of the action they have taken. This information can be incorporated into the model as an additional column in the memory matrix (Equation (5.1)).

However, analyzing this information requires another NLP model and can only be applied to games where the context gives this information. Thus, it has been ignored in the presented model.

magnitude gives information about the degree of impact a certain character trait will have on the final action probabilities. This is particularly true given that the $\text{softmax}(\cdot)$ function is not invariant under scaling², which entails obtaining a greater disparity in the final action probabilities. Therefore, the higher the weights $\mathbf{w}$, the higher the scores, and the greater the disparity. This entails that the weights can be scaled to control the disparity in the final action probabilities to reduce the randomness of the decision.

We can also analyze the impact of a certain past action, character trait, or the current labels' importance on the weight w_j of the character trait l_j . This can be accomplished by evaluating f_j on a matrix with all null elements except the ones we want to analyze. This analysis is much simpler when f_j can be written as follows

$$\begin{aligned} f_j(M, S) &= S^T (M \odot B) \mathbf{1}_J \\ &= \sum_{i=1}^{N+1} \sum_{k=1}^J s_i m_{i,k} b_{i,k} \end{aligned} \quad (5.5)$$

where $\odot$ is the element-wise multiplication and $\mathbf{1}_x$ is the column vector of ones of size $x \times 1$.

This function can be summarized as giving a certain weight $b_{i,k}$ to each element $m_{i,k}$ of the memory matrix. The step vector S assures that no invalid values of M are taken into account. Therefore, the matrix B is a $(N + 1) \times J$ matrix of weights.

Given f_j in the shape of Equation (5.5), the element $b_{i,k}$ expresses the importance that $m_{i,k}$ has on the weight w_j . In other words, $b_{i,k}$ is the importance that the label l_k of the action taken in step $t - i + 1$ has on the weight w_j of the character trait l_j . Moreover, since each element $b_{i,k}$ has a linear impact on the weights $\mathbf{w}$, the insights on the meaning of the weight's sign and magnitude also apply to these elements. If $b_{i,k} > 0$, having taken a past action with high alignment to character trait l_k will favor actions with high alignment to character trait l_j . If $b_{i,k} < 0$, it will disfavor those actions. On the other hand, by scaling the matrix B we can control the disparities of the final probabilities $\mathbf{p}_a$.

The intercept of f_j ($w_{\text{present}} = \sum_{k=1}^J b_{1,k}$) represents the importance of the labels obtained for the current actions (step t). Thus, by comparing the elements

² $\text{softmax}(k \cdot x) \neq k \cdot \text{softmax}(x)$.

$\{b_{i,j} \mid i > 1\}$ with the intercept $w_{present}$, we can find the relative importance of the past with respect to the present. More specifically, if $\sum_{i=2}^I \sum_{k=1}^J b_{i,k} > w_{present} = \sum_{k=1}^J b_{1,k}$, the past can potentially³ have a higher impact on the weight than the present.

Another detail about the functions $\mathbf{f}$ is that they can be any function. This provides considerable flexibility for implementing a wide range of behavior profiles. There are two options when implementing behavior profiles that change their behavior drastically when certain conditions are met or have sharp changes, such as changing from risky behavior to a risk aversion behavior from one step to the next.

- **Piece-wise function.** Since $\mathbf{f}$ can be any function that receives a matrix as input and returns a real number, piece-wise functions can be used to adjust to these situations.
- **Markov chain agent or state machine agent.** In some situations, separating a complex behavior profile into simpler profiles and combining them through a Markov chain or a state machine that changes from one agent to another might be easier.

Once we have obtained the final action probabilities $\mathbf{p}_a$, the agent selects which action to take through a weighted random choice, with the weights being $\mathbf{p}_a$. Therefore, this procedure is non-deterministic, even though the agents will select their highest probability choice in the mean. This is necessary since players do not make their choices deterministically but also explore the game and have some uncertainty. As already mentioned, the functions $\mathbf{f}$ can be scaled to increase the disparities of the final action probabilities, thus decreasing the importance of the synthetic players' exploratory/uncertain behavior.

Examples

We will now present a practical example. Some of the main ideas mentioned throughout this section must be kept in mind.

- The level of alignment of the actions with the character traits is given by matrix H , the output of the label predictor.

³The actual impact of the elements $\{b_{i,j} \mid i > 1\}$ depends on the values of the input L .

Algorithm 1 Decision making process

- 1: **Hyperparameters.** $\mathbf{f} \leftarrow$ row vector of behavior profile functions
 $N \leftarrow$ number of past steps to save in memory
 - 2: **Initialization.**
 $J \leftarrow \text{length}(\mathbf{f})$ $\triangleright$ J is the number of character traits
 $M \leftarrow \text{zeros}(N + 1, J)$ $\triangleright$ memory matrix, $(N + 1) \times J$ zero matrix
 $M[1, :] \leftarrow 1$ $\triangleright$ set the first row of M to all 1 as in Equation (5.1)
 $S \leftarrow \text{zeros}(N + 1, 1)$ $\triangleright$ step vector, $(N + 1) \times 1$ zero column vector
 $S[1, 1] \leftarrow 1$ $\triangleright s_1 = 1$
 - 3: **while** game has not ended **do**
 - 4: **input** from the simulator
 $s \leftarrow$ current context/state
 $\mathbf{a} = \{a_1, \dots, a_I\} \leftarrow$ list of available actions
 - 5: $H \leftarrow \text{label_predictor}(s, \mathbf{a})$ $\triangleright I \times J$ matrix of $[0, 1]^{\mathbb{R}}$ label values
 H gives the percentages of character trait alignment for each action
 - 6: $\mathbf{p}_a \leftarrow \text{softmax}(H\mathbf{f}(M, S)^T)$ $\triangleright \mathbf{f}(M, S) = [f_1(M, S), \dots, f_J(M, S)]$
 - 7: $\text{action_idx} \leftarrow$ choose randomly from $\{1, \dots, I\}$ with weights given by $\mathbf{p}_a$
 - 8: **update** the memory matrix M
 $M[3 :, :] \leftarrow M[2 : -1, :]$ $\triangleright$ shift all rows down except M_{1*} (row 1)
 $M[2, :] \leftarrow H[\text{action}, :]$ $\triangleright$ update row 2 with the values of H
corresponding to the chosen action
 - 9: **update** the step vector S
 $S[2 :, 1] \leftarrow M[1 : -1, 1]$ $\triangleright$ shift all rows down
 $S[1, 1] \leftarrow 1$ $\triangleright$ set element $s_1 = 1$
 - 10: **output** to the simulator
 action_idx
 - 11: **end while**
-

- The memory matrix M , Equation (5.1), holds past labels used for decision making.
- The functions $\mathbf{f}$ characterize the behavior profile given to the agent and are the hyperparameter of the model.
- The functions $\mathbf{f}$ are used to obtain the weights $\mathbf{w}$. These weights give the agent's preference for the current decision of an action being aligned with a certain character trait. Positive weights show preference, while negative values show evasion.
- Decisions are a random choice with weights $\mathbf{p}_a$.
- $\mathbf{p}_a$ is obtained as a weighted sum (with weights $\mathbf{w}$) of the character traits (labels).
- Exploratory/uncertain behavior can be controlled by scaling the functions $\mathbf{f}$. Higher values entail higher disparities in the final action probabilities, thus increasing the deterministic behavior.

For the following examples, we will assume only one character trait is used to characterize the behavior profiles, which is safety (l_s). Moreover, the matrix of character traits alignment H and the available actions $\mathbf{a} = \{a_1, a_2\}$ are constant throughout the iterations.

$$H = \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix}$$

Therefore, a_2 is considered a safe action since its safety value is $h_2 = 0.7 > 0.5$, while a_1 is a risky action since its safety value is $h_1 = 0.3 < 0.5$.

1. Safe (Risk Aversion) Behavior

For a simple safe agent without memory ($N = 0$), the behavior profile is characterized by

$$f(M, S) = 2 \tag{5.6}$$

This profile favors safety by giving it a positive importance.

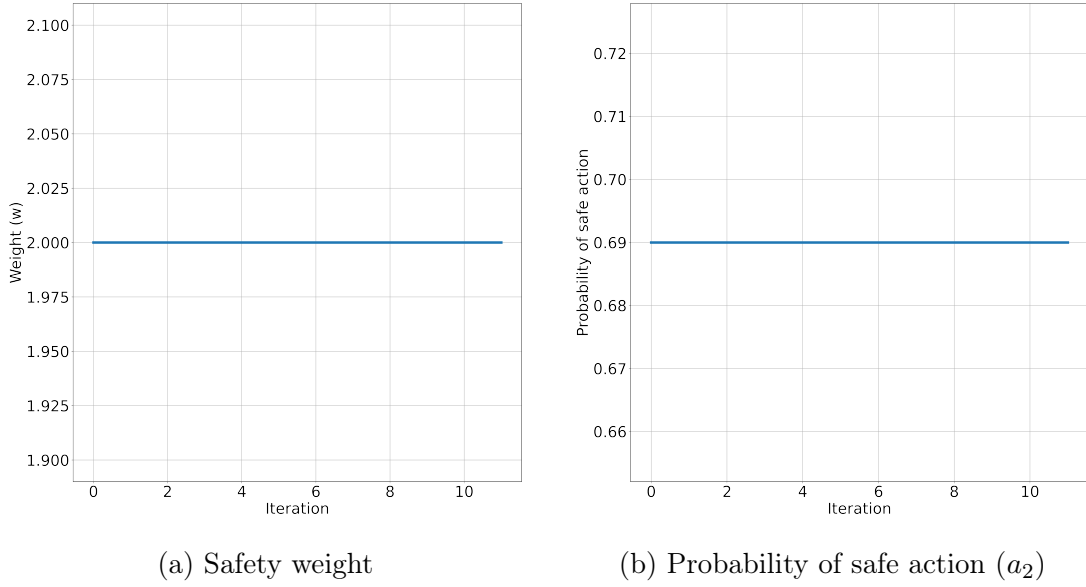


Figure 5.4: Example: Safe behavior per iteration

Since $N = 0$, $M = [1]$. For any iteration, $\mathbf{w} = \mathbf{f}(M, S) = [2]$. The vector of action probabilities is $\mathbf{p}_a = \text{softmax}(L[2]) = \text{softmax}\left(\begin{bmatrix} 0.6 \\ 1.4 \end{bmatrix}\right) = \begin{bmatrix} 0.31 \\ 0.69 \end{bmatrix}$. This safe agent will choose a_2 with probability 69% and a_1 with probability 31%. Therefore, it will always choose the safe action.

Fig. 5.4 presents how the safety weight $\mathbf{w}_1$ and the probability of choosing the safe action p_{a_2} vary with iteration. For the safe behavior, the weight and the probability are both constant since it has no memory and the inputs do not change.

If we choose a value higher than 2, the disparity between these probabilities will increase. For example, with $f(M, S) = 3 \rightarrow \mathbf{p}_a = \begin{bmatrix} 0.23 \\ 0.77 \end{bmatrix}$.

2. Risky Behavior

For a simple risky agent without memory ($N = 0$), the behavior profile is characterized by

$$f(M, S) = -2 \tag{5.7}$$

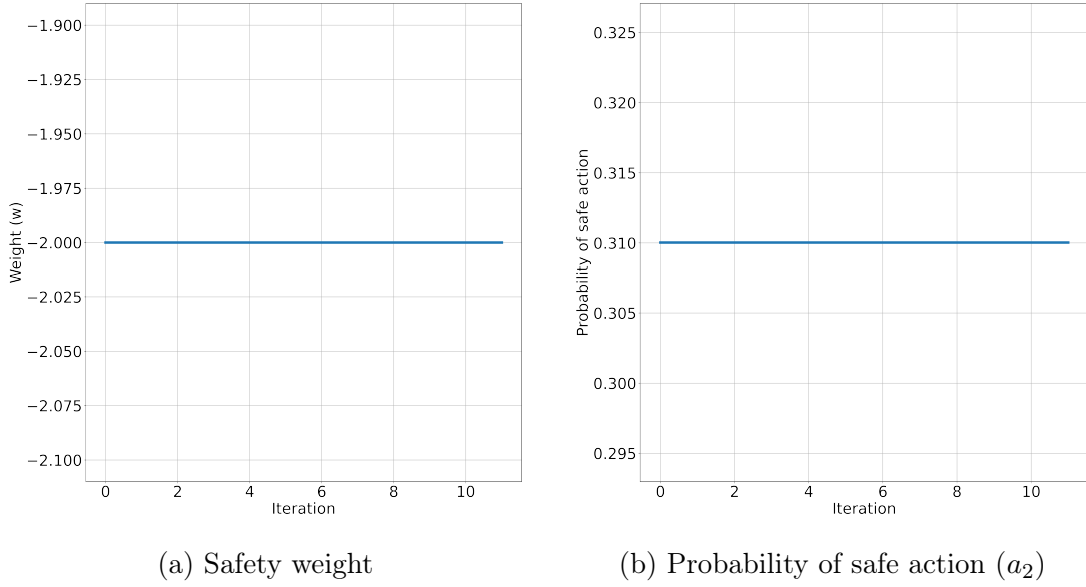


Figure 5.5: Example: Risky behavior per iteration

This profile disfavors safety (favors risk) by giving it a negative importance.

Since $N = 0$, $M = [1]$. For any iteration, $\mathbf{w} = \mathbf{f}(M) = [-2]$. The vector of action probabilities is $\mathbf{p}_a = \text{softmax}(L[-2]) = \text{softmax}\left(\begin{bmatrix} -0.6 \\ -1.4 \end{bmatrix}\right) = \begin{bmatrix} 0.69 \\ 0.31 \end{bmatrix}$. This risky agent will choose a_1 with probability 69% and a_2 with probability 31%. Therefore, it will always choose the risky action, the opposite to the safe behavior.

Fig. 5.5 presents how the safety weight $\mathbf{w}_1$ and the probability of choosing the safe action p_{a_2} vary with iteration. For the risky behavior, the weight and the probability are both constant since it has no memory and the inputs do not change.

3. Random Behavior

For a random agent without memory ($N = 0$), the behavior profile is characterized by

$$f(M, S) = 0 \quad (5.8)$$

This profile does not give any importance to safety. Since there is no other character trait, it will take decisions randomly.

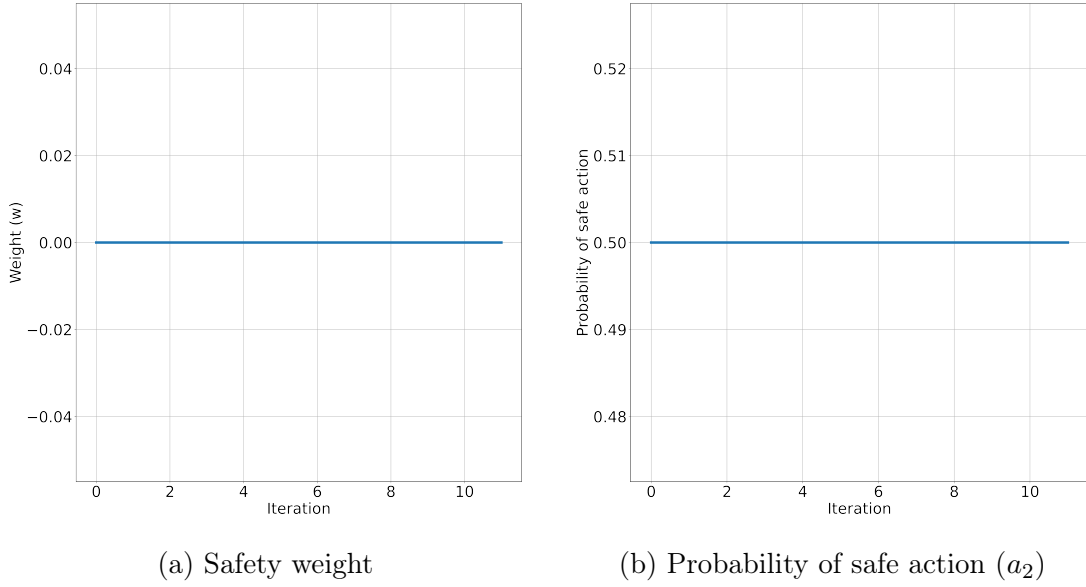


Figure 5.6: Example: Random behavior per iteration

Since $N = 0$, $M = [1]$. For any iteration, $\mathbf{w} = \mathbf{f}(M) = [0]$. The vector of action probabilities is $\mathbf{p}_a = \text{softmax}(L[0]) = \text{softmax}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 0.50 \\ 0.50 \end{bmatrix}$. This random agent will choose a_1 with probability 50% and a_2 with probability 50%.

Fig. 5.6 presents how the safety weight $\mathbf{w}_1$ and the probability of choosing the safe action p_{a_2} vary with iteration. For the random behavior, the weight and the probability are both constant since it has no memory and the inputs do not change.

4. Safe NTM (but not too much) Behavior

We will now implement a more complex agent which is safe by nature but becomes riskier the safer it has been in the past 2 decisions. Therefore, it has a 2-steps memory ($N = 2$) and is characterized by

$$\begin{aligned}
 f(M, S) &= S^T(M \odot B) \\
 &= \sum_{i=1}^{N+1} s_i m_{i,1} b_{i,1} \\
 &= b_{1,1} + s_2 m_{2,1} b_{2,1} + s_3 m_{3,1} b_{3,1}
 \end{aligned} \tag{5.9}$$

where $\odot$ is the element-wise multiplication and B is

$$B = \begin{bmatrix} 8 \\ -8 \\ -6 \end{bmatrix}$$

This profile takes into account the last 2 actions. $b_{1,1} > 0$, so it will favor safety on current actions (step t). However, $b_{2,1} < 0$ and $b_{3,1} < 0$, thus giving a negative weight to safety in past actions (steps $t-1$ and $t-2$ respectively). Therefore, the safer the actions it has taken in the past ($m_{i,1} > 0.5$), the larger the absolute value of the past importance ($|s_i m_{i,1} b_{i,1}|$ with $i \neq 1$), thus leading towards risky actions. A risky action is taken when $|s_2 m_{2,1} b_{2,1} + s_3 m_{3,1} b_{3,1}| > b_{1,1}$ since $f(M, S) < 0$. In contrast, when a risky action has been taken ($m_{i,1} < 0.5$), the absolute value of the past becomes smaller, increasing the relative importance of the present and leading to safe actions ($f(M, S) > 0$) once again.

We will now illustrate how this behavior takes decisions and changes from safe to risky through a couple of iterations. For simplicity, we will assume the action with the highest probability is always chosen (in reality this is not always the case).

Since $N = 2$, the initial memory matrix will be $M = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ and the step vector $S = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$.

Iteration 1

$\mathbf{w} = \mathbf{f}(M, S) = [S^T(M \odot B)] = [8]$. Thus, $\mathbf{p}_a = \text{softmax}(L[8]) = \begin{bmatrix} 0.04 \\ 0.96 \end{bmatrix}$, and a_2 is chosen. The memory matrix is then updated with the labels from a_2

$$M = \begin{bmatrix} 1 \\ 0.7 \\ 0 \end{bmatrix}$$

and the step vector is also updated $S = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$.

Iteration 2

$\mathbf{w} = [2.4]$. Thus, $\mathbf{p}_a = \text{softmax}(L[2.4]) = \begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix}$, and a_2 is chosen once more. The memory matrix is then updated with the labels from a_2

$$M = \begin{bmatrix} 1 \\ 0.7 \\ 0.7 \end{bmatrix}$$

and the step vector is also updated one last time $S = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

Iteration 3

$\mathbf{w} = [-1.8]$. Thus, $\mathbf{p}_a = \text{softmax}(L[-1.8]) = \begin{bmatrix} 0.67 \\ 0.33 \end{bmatrix}$, and a_1 is chosen. The memory matrix is then updated with the labels from a_1

$$M = \begin{bmatrix} 1 \\ 0.3 \\ 0.7 \end{bmatrix}$$

and the step vector S remains unchanged.

As illustrated, this behavior profile becomes riskier as it takes on safe actions, progressing from a very safe behavior to a risky behavior. If we continue iterating, it will progress back to a safe behavior since it has taken up risk in the last action. Since the input label matrix H is constant, this process will repeat itself indefinitely. Fig. 5.7 presents how the safety weight $\mathbf{w}_1$ and the probability of choosing the safe action p_{a_2} vary with iteration. In this figure we can observe the periodic pattern mentioned.

5. Risky NTM (but not too much) Behavior

We will now implement the opposite to the *Safe NTM* agent. This agent is risky by nature and becomes safer the riskier the actions it has taken in the past 2 steps. Therefore, it a 2-steps memory ($N = 2$) and its characterized by

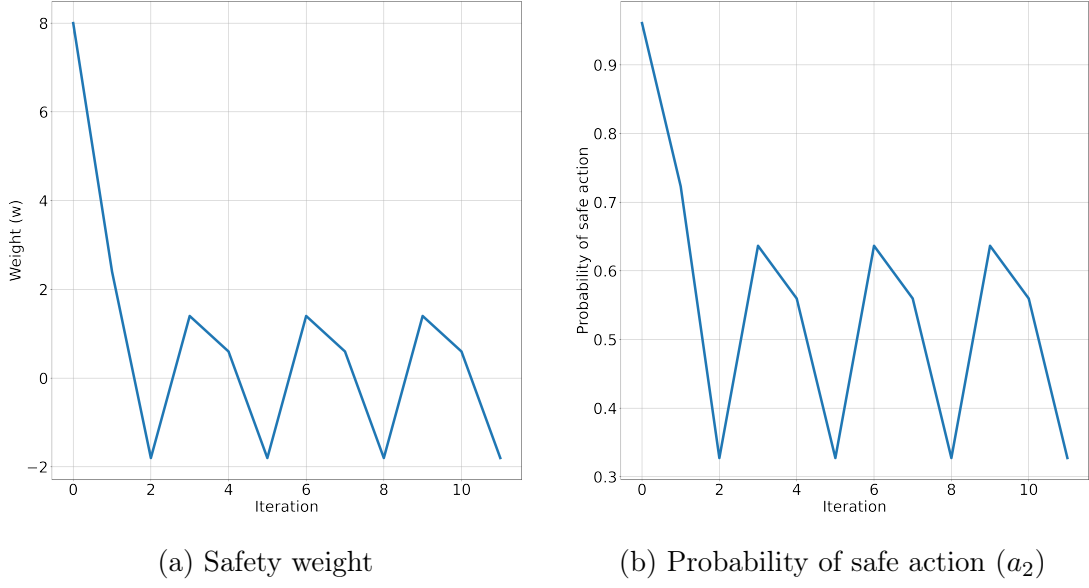


Figure 5.7: Example: Safe NTM behavior per iteration

$$\begin{aligned}
f(M, S) &= b_{1,1} + S^T((1 - M) \odot B) \\
&= b_{1,1} + \sum_{i=2}^{N+1} s_i(1 - m_{i,1})b_{i,1} \\
&= b_{1,1} + s_2(1 - m_{2,1})b_{2,1} + s_3(1 - m_{3,1})b_{3,1}
\end{aligned} \tag{5.10}$$

where $\odot$ is the element-wise multiplication, 1 is the $(N + 1) \times J$ matrix of ones, and B is

$$B = \begin{bmatrix} -8 \\ 8 \\ 6 \end{bmatrix}$$

This profile considers the last 2 actions. $b_{1,1} < 0$, so it will favor safety on current actions (step t). However, $b_{2,1} > 0$ and $b_{3,1} > 0$, thus giving a positive weight to safety in past actions (steps $t - 1$ and $t - 2$ respectively). Therefore, the riskier the actions it has taken in the past ($m_{i,1} < 0.5$), the larger the absolute value of the past importance ($s_i(1 - m_{i,1})b_{i,1}$ with $i \neq 1$), thus leading towards safer actions. A safe action is taken when $s_2(1 - m_{2,1})b_{2,1} + s_3(1 - m_{3,1})b_{3,1} > |b_{1,1}|$ since

$f(M, S) > 0$. In contrast, when a safe action has been taken ($m_{i,1} > 0.5$), the absolute value of the past becomes smaller, increasing the relative importance of the present and leading to risky actions ($f(M, S) < 0$) once again.

We will now illustrate how this behavior takes decisions and changes from risky to safe through a couple of iterations. For simplicity, we will assume the action with the highest probability is always chosen (in reality this is not always the case).

Since $N = 2$, the initial memory matrix will be $M = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ and the step vector

$$S = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

Iteration 1

$\mathbf{w} = \mathbf{f}(M, S) = [b_{1,1} + S^T((1-M) \odot B)] = [-8]$. Thus, $\mathbf{p}_a = \text{softmax}(L[-8]) = \begin{bmatrix} 0.96 \\ 0.04 \end{bmatrix}$, and a_1 is chosen. The memory matrix is then updated with the labels from a_1

$$M = \begin{bmatrix} 1 \\ 0.3 \\ 0 \end{bmatrix}$$

and the step vector is also updated $S = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$.

Iteration 2

$\mathbf{w} = [-2.4]$. Thus, $\mathbf{p}_a = \text{softmax}(L[-2.4]) = \begin{bmatrix} 0.72 \\ 0.28 \end{bmatrix}$, and a_1 is chosen once more. The memory matrix is then updated with the labels from a_1

$$M = \begin{bmatrix} 1 \\ 0.3 \\ 0.3 \end{bmatrix}$$

and the step vector is also updated one last time $S = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

Iteration 3

$\mathbf{w} = [1.8]$. Thus, $\mathbf{p}_a = \text{softmax}(L[1.8]) = \begin{bmatrix} 0.33 \\ 0.67 \end{bmatrix}$, and a_2 is chosen. The memory matrix is then updated with the labels from a_2

$$M = \begin{bmatrix} 1 \\ 0.7 \\ 0.3 \end{bmatrix}$$

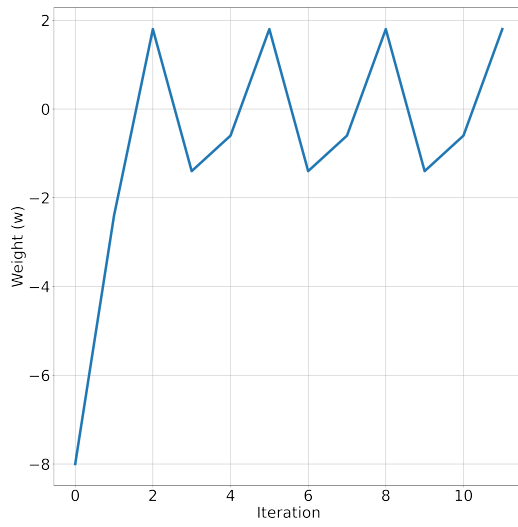
and the step vector S remains unchanged.

As illustrated, this behavior profile acts as the complete opposite to the *Safe NTM*, becoming safer as it takes on risky actions and progressing from a very risky behavior to a safe behavior. If we continue iterating, it will progress back to a risky behavior since it has taken a safe action in the last step. Since the input label matrix H is constant, this process will repeat itself indefinitely. Fig. 5.8 presents how the safety weight $\mathbf{w}_1$ and the probability of choosing the safe action p_{a_2} vary with iteration. In this figure we can observe the periodic pattern mentioned.

It is essential to note the effect of multiplying by $1 - M$ in Equation (6.4) instead of M . If we take the characteristic function from *Safer NTM* (Equation (6.5)) and only negate B , we do not obtain the behavior we are after.

1. First, this behavior would take risky decisions. Therefore, $c = m_{2,1} = 0.7$ and $d = m_{3,1} = 0.7$.
2. When the first safe decision is taken $s_2cb_{2,1} + s_3db_{3,1} > |b_{1,1}|$.
3. Once the safe decision is taken the memory matrix is updated and $c' = m_{2,1} = 0.3$. Thus, $s_2c'b_{2,1} + s_3cb_{3,1} > s_2cb_{2,1} + s_3db_{3,1} > |b_{1,1}|$ and this behavior would keep taking safe decisions indefinitely as presented in Fig. 5.9.

This is due to the absolute value of the past increasing when a safe decision is made instead of decreasing to let the present take back control. This effect is solved by using $1 - M$ instead of M .



(a) Safety weight

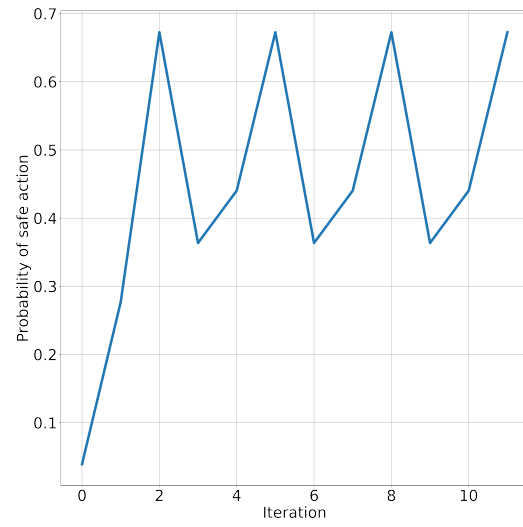
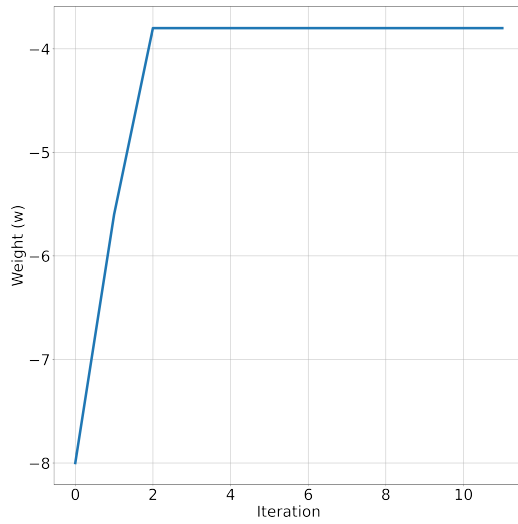
(b) Probability of safe action (a_2)

Figure 5.8: Example: Risky NTM behavior per iteration



(a) Safety weight

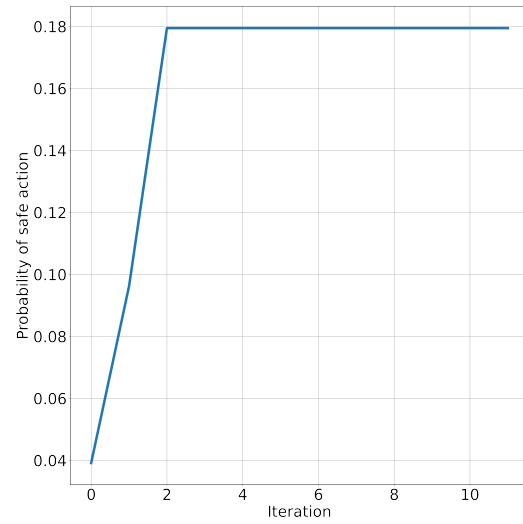
(b) Probability of safe action (a_2)

Figure 5.9: Example: Wrong Risky NTM behavior per iteration

5.3 Modules

This section introduces some modules that extend the components from Sec. 5.1 and 5.2.

5.3.1 Graph Simulator

The graph simulator is the most generic simulator. As mentioned in Sec. 5.1.2, all text games are characterized by their decision graph. Therefore, this module allows easy implementation of new games without creating a new simulator component for each one. It takes the decision graph of the story being implemented as input, which can be obtained manually or through a parser. Moreover, it has additional functionality such as webpage generation for the story or drawing its decision graph.

5.3.2 HTML Webpage Generation

Apart from being used for synthetic agents, the simulator can also be used for real players. However, the terminal access is not user-friendly. This is solved by generating an HTML webpage from the graph simulator module. By combining HTML generation and the graph simulator, game developers will only need to implement the story once, with its decision graph, instead of having to implement the simulator component and the webpage for real players separately.

The graph simulator obtains the HTML webpage from the decision graph following a simple set of rules:

- Each node creates a new HTML document.
- The state's text is added to the document.
- The actions are added to the document as options in an unordered list. These options are linked to the HTML document that holds the next state.
- The index HTML document is created with some helpful information and a single action ("Start the Game") that leads to the starting state.

5.3.3 Compound Agents

Compound agents can be used when implementing a behavior profile with sharp changes or when it is too complex to be characterized by a single agent. Compound agents are made up of simpler agents that transition from one to another according to a particular policy. These simpler agents are known as states, with some actively involved in decision-making and the rest idle. Two compound agents that cover most of these complex behavior profiles are Markov chains and state machines.

- **Markov Chain Agents.** This compound agent is useful when the transitions are non-deterministic, and their probabilities depend only on the previous active agent.
- **State Machine Agents.** This compound agent allows incorporating transitions with complex conditions.

6 | Implementation

In this chapter, we present an already existing text game and illustrate the process of implementing step-by-step the architecture presented in Ch. 5. We use [Python](#) for the implementation because of the numerous libraries available, which allows us to use the same language for the different components.

The architecture has been implemented on a serious text game [22] aimed at educating about cybercrimes, particularly *Online Grooming*. Thus, we are primarily interested in how players make the decisions rather than beating the game. Using a set of behavior profiles, we generate a synthetic dataset for this game. The generated dataset will later be used in Ch. 7 to train a variety of Machine Learning models. The trained models use the decisions taken throughout each run to classify each run by the behavior profile that generated it.

6.1 Simulator

Text games can come in a variety of formats. We will illustrate the process of obtaining the decision graph needed for the graph simulator from an already existing story.

6.1.1 Yarn Spinner Format

This game is available in [Yarn Spinner](#) format, which uses the Yarn language to build branching narrative and dialogue in games. It is a potent tool since it can be used with [Unity](#), which is extensively used for creating video games. In Yarn Spinner, the story is given in various files representing states. Each state has a storyline with a series of decisions leading to new states. This format uses variables

to keep track of the decisions taken and allows executing commands (for example, *If* and *Go* statements, or even user-created commands) to drive the player through the required story branch.

Code 6.1 shows an example snippet from the game's Yarn files. It can be observed that it has a dialogue format and that the dialog options are included starting with "->" (as in line 6). Choosing one option or another affects the story and the context text differently. This snippet also includes instructions embedded as «*command*» (as in line 7).

Listing 6.1: Yarn Spinner example

```

1 Mamá: ¿Estás jugando a estas horas?
2 Mamá: [name], me alegro de que te guste el regalo, pero ya
   es tarde y mañana tienes colegio.
3 Mamá: Anda, apágalo y vete a dormir.
4 User: Sí, mamá.
5 <<Hide @ Mamá>>
6 -> Seguir jugando
7     <<set $jugarTarde to true>>
8     Cuando mi madre cerró la puerta, me puse el pijama y me
       tumbé en la cama con el portátil
9     Para simular que estaba durmiendo, apagué las luces y
       seguí jugando a oscuras con los auriculares
10    No fue hasta la madrugada cuando me empezaron a doler
       los ojos y me fui a dormir.
11 -> Ir a dormir
12     <<set $jugarTarde to false>>
13     Mi madre tenía razón, mañana iba a estar muy cansada
14     Guardé la partida y cerré el ordenador.
15     Afortunadamente, terminé mis deberes a lo largo del fin
       de semana, por lo que no me tenía que preocupar por
       las entregas del día siguiente.
16     Me puse el pijama y me fui a dormir

```

6.1.2 Yarn Parser

The first step is to be able to play the game, which can be accomplished using a parser for the Yarn Spinner files. The parser reads the Yarn Spinner files and saves all the available states. It then parses each game state in sequence, presenting the state text and available actions, keeping track of the decisions taken, and executing

the commands given. We used as base a parser developed by Pfeiffer [17] and expanded it with some additional functionality needed to work with our story¹:

- Added history of choices taken.
- Added jumps when only one option available (automatically select that option).
- Added *Go* instruction.
- Fixed *Set* and *If* instructions for boolean values.
- Added inline *If* statements in actions.
- Added state attributes given in the header of Yarn Spinner files.
- Added action attributes given as key:value.

6.1.3 Decision Graph

Once we can play the game using the parser, we have two options to implement the simulator component. The first option is to use the parser directly by extending the simulator interface. The second option is to obtain the decision graph and use the graph simulator. Using the graph simulator is recommended since it does not require parsing the game each time we want to run it, thus making it faster. Moreover, using the graph simulator for all the games simplifies changing from one game to another.

Text games can be represented by their decision graph. Therefore it makes sense to use graph searching algorithms to obtain its graph representation, its decision graph. The standard algorithms used are Depth-First Search (DFS) and Breadth-First Search (BFS). The most suitable option to use in our situation is DFS since, even though both have the same time complexity $O(V + E)$, many parsers are not able to go backward in the story, thus having to restart over to visit all the options available at the same level. This makes BFS computationally more costly, being DFS faster and more suitable than BFS in our situation. Algorithm 2 presents the algorithm followed to obtain the decision graph from the story using DFS.

¹The code is available on <https://github.com/vibalcam/yarn-pygame>

Algorithm 2 Decision Graph using Depth First Search

Input. $G \leftarrow$ graph object to be filled with the text game
 $game \leftarrow$ parsed game that can be used to play

Main Program. Call $DFS(G, game, [])$

$DFS(G, game, path)$

$path \leftarrow$ list of actions taken to get to the current state

- 1: $state, actions \leftarrow game.read()$ $\triangleright$ current state and available actions with their attributes
 - 2: Add node $state$ to G
 - 3: **for** each action $a \in actions$ **do**
 - $\triangleright$ If $game$ and $path$ are not in the same state (check by comparing the past decisions in the $game$ and $path$), we need to restart the game and take the same decisions as in $path$.
 - 4: **if** $game.history() \neq path$ **then** $\triangleright history() \leftarrow$ current game's path
 - 5: $game.restart()$
 - 6: **for** k in $path$ **do**
 - 7: $game.transition(k)$
 - 8: **end for**
 - 9: **end if**
 - 10: $game.transition(a)$ $\triangleright$ select action a and advance the game
 - 11: $child, _ \leftarrow game.read()$ $\triangleright$ only read the current state
 - 12: Add edge $(state, child)$ to G
 - 13: **if** $child \notin G$ **then**
 - 14: $DFS(G, game, game.history())$
 - 15: **end if**
 - 16: **end for**
-

Algorithm 2 is implemented using two libraries that help deal with graphs, [NetworkX](#) and [Graphviz](#). NetworkX is capable of working with ease with graphs and is used by the graph simulator to represent the decision graph. Graphviz is used to render the graphs neatly and easily.

It is important to note that to obtain the decision graph for our game, in line 2.13, we must consider the value of the variables saved by the game when checking if the *child* state is already in graph G . The decision graph obtained for our game is presented in Appendix B. As it can be observed, this graph is considerably large since it considers every change to represent the whole game correctly. Moreover, some decisions taken in the game’s early stages do not have consequences until the game’s ending stages. Therefore, the decision graph splits very early even though most of the stages will be repeated in both branches, which increases the graph’s size considerably.

Instead, We can ignore the variables saved by the game in line 2.13 to obtain a graph representing the story’s narrative. This graph does not faithfully represent the game, meaning that following a path in the graph does not mean that this same path can be followed in-game. Instead, if two states are connected in the narrative graph, the possibility of going from the source state to the destination state exists (with just the narrative graph, we would not know which path to take to open this possibility). Therefore, the narrative graph is useful to show an overview of the game’s storyline. Fig. 6.1 presents the graph narrative of the game. When comparing it with the narrative given by the original author (Fig. 6.2) we can observe that the graph narrative does show the overall narrative of the story.

6.1.4 HTML Webpage Generation

Once we have the decision graph and the graph simulator, we can easily obtain the HTML webpage for the game as mentioned in Sec. 5.3.2. Fig. 6.3 shows an example of a generated HTML webpage. If needed, a webpage style (CSS) can be easily added through the graph simulator, so the style is incorporated into all the HTML documents. However, this goes beyond the scope of this implementation.

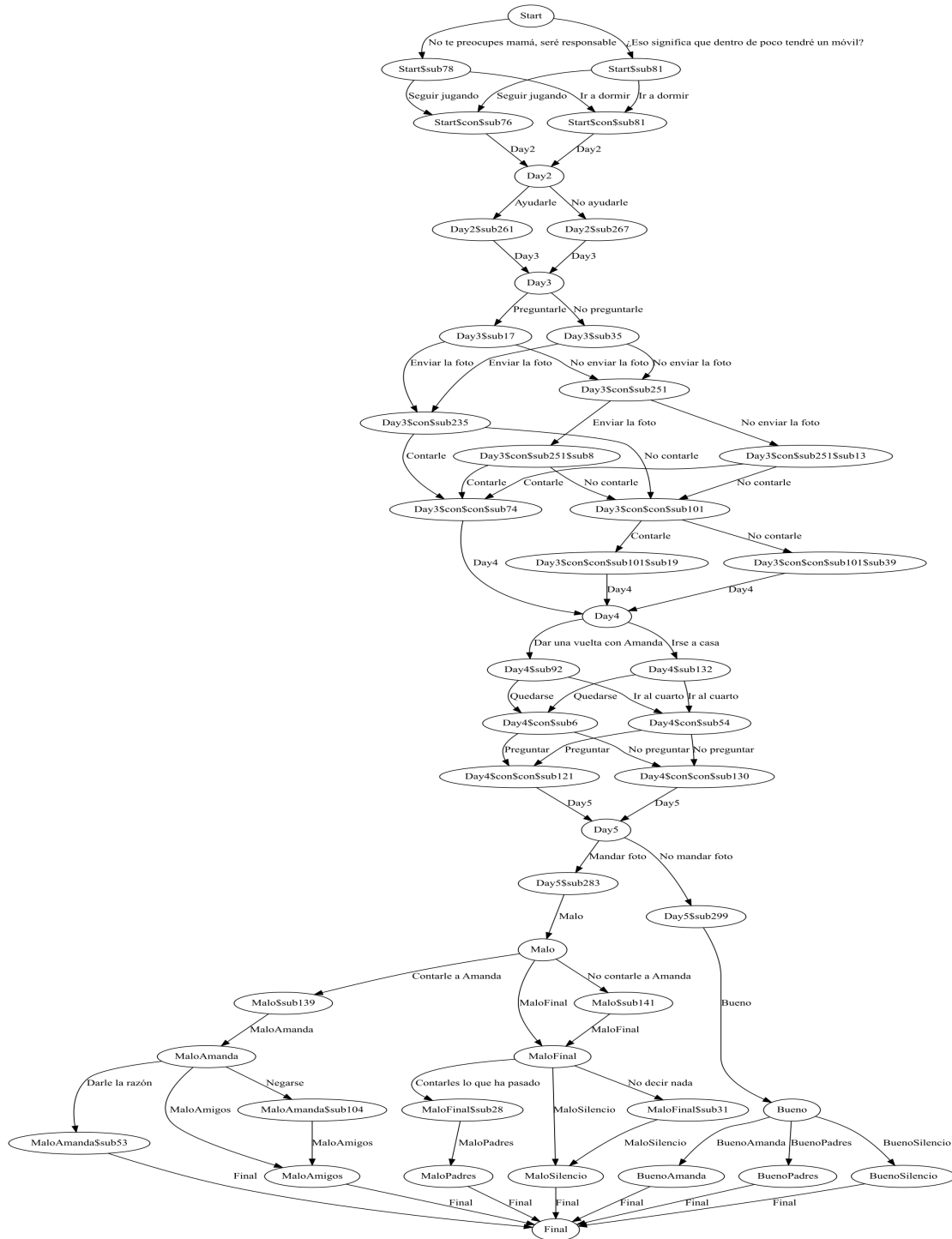


Figure 6.1: Graph narrative obtained from the game

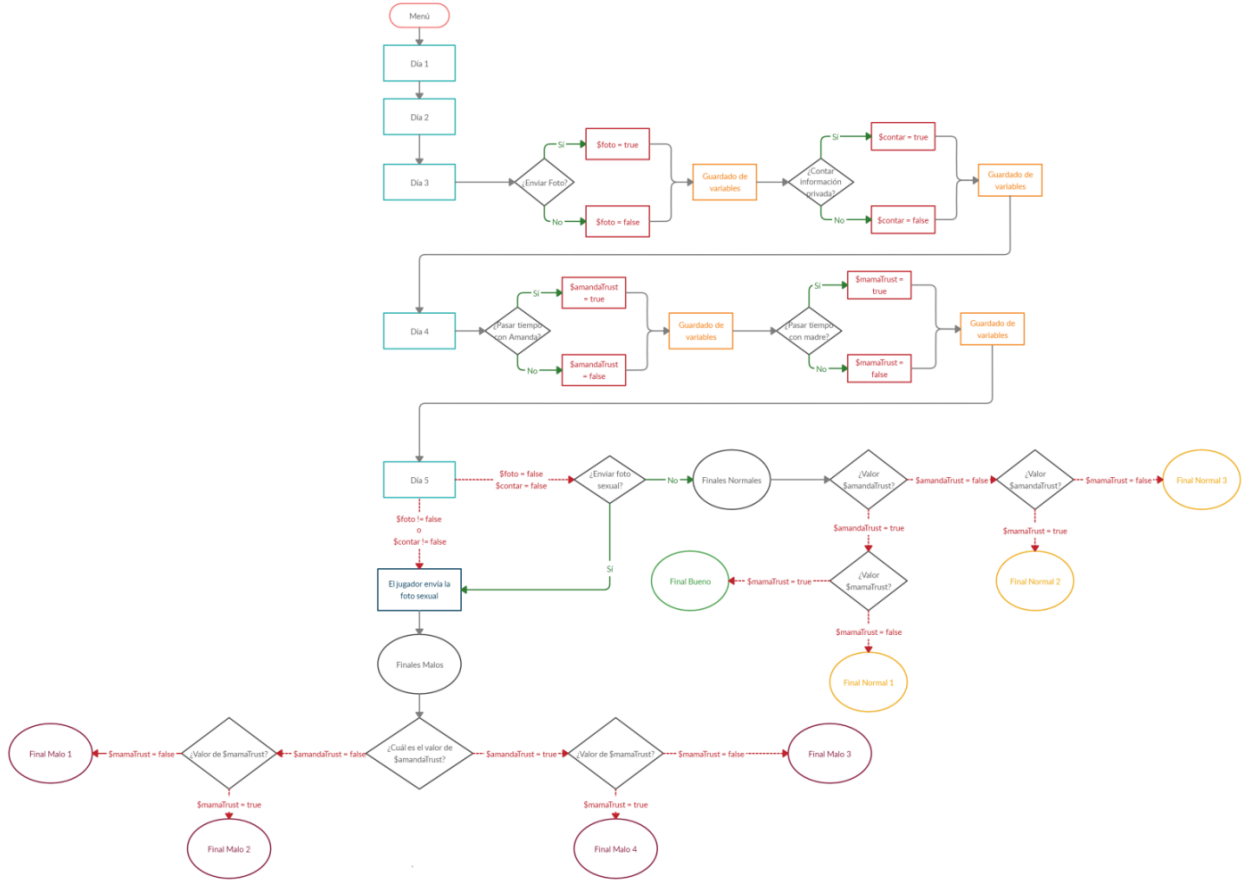


Figure 6.2: Original narrative of the game [22]

6.2 NLP-Based Label Predictor

We will now continue by implementing the agent. First, we will implement the label predictor, which tackles the NLP task. The preferable choice to reduce bias is to use a pre-trained model on a similar dictionary or game. When there is no suitable pre-trained model for our task, we can use tools such as TextWorld [1] to train our model on a controlled set of games with our dictionary. However, for the sake of simplicity, we will train the model in our game.

User: Me da un poco de verguenza...
HallyHealer07: ¿Por qué?!
HallyHealer07: Te prometo que no subiré la foto a ningún lado.
HallyHealer07: Mira si quieres te mando una mía primero.
Antes de que pudiera contestarle que no hacía falta, vi como Hally me envió una foto por el chat.
Con mucha curiosidad, abrí la imagen.
Hally era un chico que más o menos aparentaba mi edad, en la foto se le veía sonriendo, con unos cascos puestos y sentado en frente del escritorio, como si se la hubiera sacado ahora mismo mientras jugaba.
HallyHealer07: Ahora tú uwu

- [Enviar la foto](#)
- [No enviar la foto](#)

Figure 6.3: Example generated HTML document

6.2.1 Labeling

The first step is labeling the game if it is not already labeled. Remember from Sec. 5.2.2 that the objective of the label predictor is, given the current state and available action, to analyze the alignment between the action and the character traits used to define the behavior profiles. Thus, we must label the game's actions according to whether they are aligned or not with the given traits. This process can be done automatically by an existing model or manually by the game developers.

For our implementation, we have manually labeled the decisions. Due to the game's simplicity, we have used a single trait (label). In the game, the player is presented with a set of situations in which he could be potentially be victim of *Online Grooming*. Therefore, we will label the actions by their safety. Thus, the actions have been labeled as follows:

- **Label 0** for actions considered unsafe or risky.
- **Label 1** for actions considered safe.

The labels can be added directly to the decision graph as edge attributes. However, using the extended functionality added to the Yarn Spinner parser developed in Sec. 6.1.2 and letting Algorithm 2 add them as edge attributes is faster. We just need to add the labels as action attributes to the Yarn files. This can be done by appending to the action a key-value parameter, given as "`__k:v`" (k is the key

and v is the value). Fig. 6.2 presents an example of how the edge attributes are added to the Yarn files so the parser can read them.

Listing 6.2: Action with attributes in Yarn file

```
1 -> Seguir jugando__r:1__q:4
```

6.2.2 Data

The data used to train the label predictor model is obtained from the decision graph. This data consists of a list of $(state, action)$ tuples. Each element represents the current state and an available action that can be taken. The game in hand contains a total of 1052 labeled unique state-action pairs. It is essential to consider that this does not mean unique states or unique actions but unique pairs.

6.2.3 Model

For the model, we have followed the architecture presented in Sec. 5.2.2. Given the large amount of data required to train a model from scratch, we take advantage of transfer learning using a pre-trained model. It would be best to use a model pre-trained in a similar field. However, we will tackle the general case using a general pre-trained model such as BERT. BERT is a state-of-the-art model widely used for fine-tuning many different tasks in NLP, making it an excellent choice for our task. Moreover, a pre-trained BERT model for the language of our choice can be obtained from Hugging Face with minimal effort. We have chosen to use the *BERT multilingual cased* model since it allows the model to work seamlessly in different languages. However, we could have chosen the pre-trained model specific to our game's language.

Once we have chosen the pre-trained model to fine-tune, the tokenizer and embeddings used must be compatible with it. Hugging Face makes it easy by providing a tokenizer for the BERT model. In addition, the pre-trained BERT model already has an embeddings layer incorporated. Furthermore, we must comply with the maximum length allowed by our pre-trained model. The BERT model we have chosen has a maximum input length of 512 tokens. Thus, we have truncated the inputs by taking the last 512 tokens without splitting a sentence in two (using the periods as separators) to maintain grammatical correctness.

The different layers of our architecture are the following:

- **Tokenizer.** BERT tokenizer.
- **Shared layer.** BERT (embeddings, encoder, and pooler) with a hidden size of 768 followed by a 768 to 768 fully connected layer, a ReLU, and a 768 to *shared_out_dim* fully connected layer.
- **Individual layers.** The state and action layers consist of a fully connected layer (*shared_out_dim* to *state_layers/action_layers*).
- **Output layer.** A fully connected layer with a single output dimension and a sigmoid activation function.

Several models have been trained and tested with different dimensions and dataset augmentations.

- *shared_out_dim* $\in \{75, 125, 200\}$
- *state_layers* $\in \{20, 30\}$
- *action_layers* $\in \{20, 30\}$
- Augmented dataset. For every (*state*, *action*) tuple, and augmented tuple is added. This augmented tuple has the same state and its action has been negated. In other words, the first letter of the action is lowercased and "No" is added in front. In addition, the label l^* of the augmented sample is the opposite from the original l ($l^* = 1 - l$).

In addition, we have also trained a similar model that uses an LSTM instead of the BERT pre-trained model. By doing this, we can verify that the transfer learning provided by the pre-trained model does produce better results. Both the pre-trained BERT and the LSTM models use the BERT tokenizer and embeddings layer. Fig. 6.4 presents diagrams comparing both models.

6.2.4 Training

The model is developed and trained using [PyTorch](#) on an NVIDIA GPU with 8 GB. The model also makes use of [Hugging Face](#) to retrieve the pre-trained model

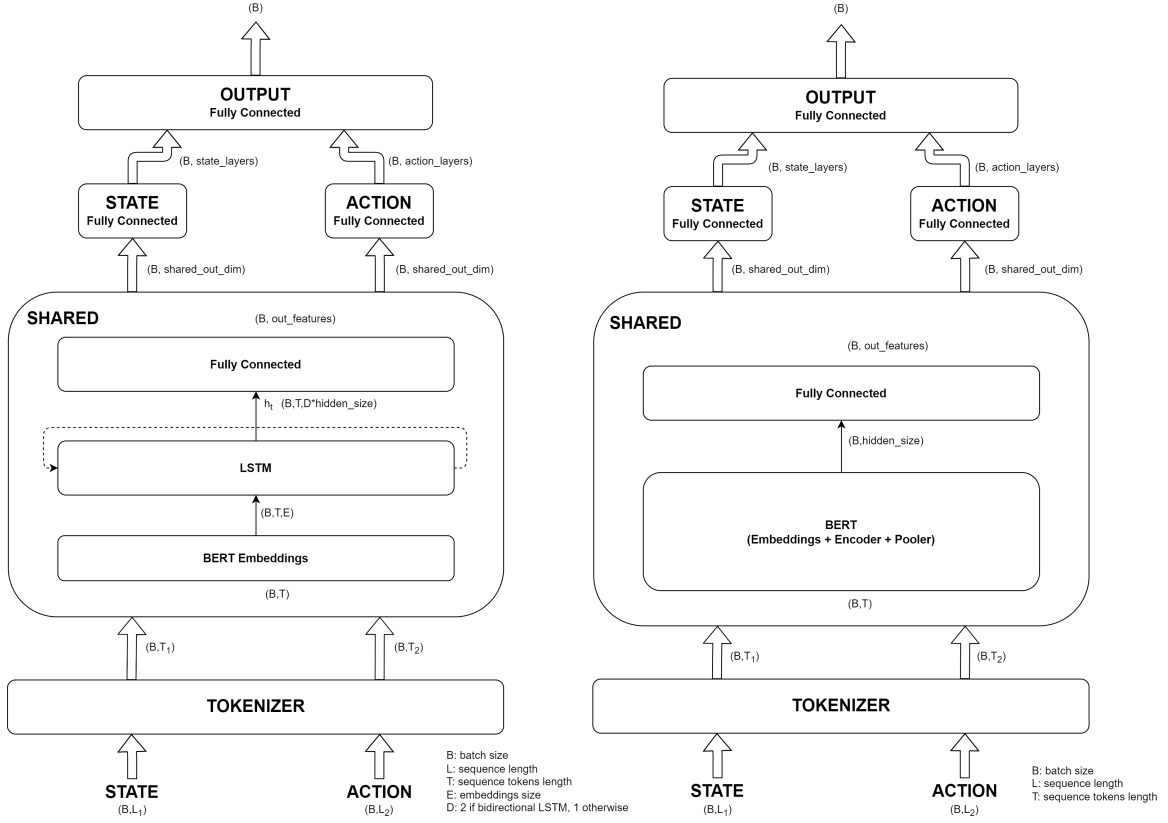


Figure 6.4: Label predictor models comparison between pretrained BERT (right) and LSTM (left)

used in the shared layer (Sec. 5.2.2). A random seed has been used to ensure reproducible results to the extent possible.

The descriptions of the training hyper-parameters are summarized in Table 6.2. The dataset has been randomly split into training, validation, and test subsets with 75%, 15%, and 1% of the dataset, respectively (Table 6.1). These percentages have been chosen considering the small size of the dataset, 1052 state-action pairs, which reinforces the need to use a pre-trained model. The loss function used is the Binary Cross Entropy loss (BCEWithLogitsLoss, which combines the sigmoid output layer with a Binary Cross Entropy loss for enhanced numerical stability). This loss is suitable for multi-label classification tasks. The loss optimizer used is AdamW [10]. A learning rate scheduler has been used, which reduces the learning rate when the validation accuracy does not change for *scheduler_patience* epochs.

Table 6.1: Size of training, validation and test sets

Set	Total	Train	Validation	Test
% of total	100	75	15	10
Samples	1052	789	158	105

Table 6.2: Training hyper-parameters

Hyper-parameter	Description
lr	starting learning rate for the optimizer
optimizer_name	optimizer used for training
n_epochs	number of epochs of training
batch_size	size of batches used in training
scheduler_mode	scheduler mode to use for the learning rate scheduler
freeze_bert	whether to freeze BERT during training
balanced_actions	whether to ensure that the model is trained with balanced actions
augment_negative	whether to augment the dataset with a negated version of the actions
scheduler_patience	patience for the learning rate scheduler

The model has been loaded to the GPU for faster training. However, the model with the pre-trained BERT is too large to be trained with our current GPU. Therefore, we have frozen BERT’s parameters and taken advantage of the additional fully connected layer added after BERT in the shared layer for fine-tuning. In addition, we have tokenized and cropped to the maximum allowed length of all the samples beforehand to speed up training.

6.2.5 Results

Table 6.3 summarizes the best LSTM-based and BERT-based models’ accuracy metrics. It can be observed that the LSTM-based model is not capable of learning the task and behaves as a nearly random model. This result was expected due to the small amount of data available, which is not enough to train the model from scratch. In contrast, the BERT-based model obtains perfect results in all three sets. This result confirms the need to use a pre-trained model to take advantage of transfer learning, which helps substantially the model’s learning. This is even more true when the amount of training data is small.

Apart from training the models in the dataset, another set of models has been trained using negated data augmentation. This augmentation was presented in

Table 6.3: Label predictor LSTM vs BERT accuracy

Model	Train	Validation	Test
LSTM	0.545	0.536	0.557
BERT	1	1	1

Table 6.4: Label predictor BERT vs BERT with negative augmentation

Model	Mean Certainty ^a			Mean Certainty Balanced ^b			Test Acc.
	Predict 0	Predict 1	Abs. Diff.	Predict 0	Predict 1	Abs. Diff.	
BERT	98.45	96.37	2.07	90.59	91.79	1.2	1
BERT Augmented	92.12	90.19	1.92	90.94	86.03	4.91	1

^a The model's predicted probability of assigning the predicted label

^b Each unique action is equally represented in the calculation

Sec. 6.2.3 and consists in duplicating the state-action tuples by negating the action. This is done by lowercasing the first letter of the action and adding "No" in front. In addition, the label l^* of the augmented sample is the opposite from the original l ($l^* = 1 - l$). Since both the models trained on the original dataset and the augmented dataset achieve perfect accuracy and MCC (Matthews Correlation Coefficient) scores, the model selection is based on the model's certainty. The model's certainty is the predicted probability of assigning the predicted label. Substantial changes in the model's certainty depending on the predicted label signify bias. This bias can be harmful to our model since it can result in higher disparities in action probabilities when favoring a character trait (label 1 provides the highest score) compared to disfavoring it (label 0 provides the highest score)². Therefore, we have chosen the model with the lowest certainty changes. Since some actions appear in the dataset more frequently than others, we have calculated the change in certainty ensuring that each unique action is equally represented. The BERT-based model trained on the original data achieves the best results, which are summarized in Table 6.4. This model's hyperparameters are presented in Table 6.5.

The selected model obtains perfect results in all three sets, a sign of overfitting. This may lead to bias towards our labeled data. A way to alleviate this bias is training the model on multiple games, on a larger dataset with a similar vocabulary, or using a model pre-trained on a similar vocabulary. However, this is not a significant concern since the decision-making process relies on the predicted

²Remember from Sec. 5.2.3 that larger predicted probabilities result in larger action scores, thus increasing the disparity in the action probabilities.

Table 6.5: Selected label predictor model hyper-parameters

Hyper-parameter	Description
shared_out_dim	125
state_layers	30
action_layers	20
lr	0.001
optimizer_name	AdamW
n_epochs	100
batch_size	8
scheduler_mode	Maximum validation MCC
balanced_actions	False
augment_negative	False
scheduler_patience	20

Table 6.6: Label predictor certainty^a statistics estimated over the actions^b

Mean	Std	Min	25%	50%	75%	Max
88	11	64.4	79.6	91.8	97	99.5

^a The model's predicted probability of assigning the predicted label^b Each unique action is equally represented

probabilities for each label rather than the predicted label. Therefore, we are primarily interested in the predicted probabilities variability, which is a sign of the predictions not being too biased. Table 6.6 confirms that the selected model does have variability in its predicted probabilities.

6.3 Decision Maker

The last step of the process is to select the set of behavior profiles given to the agents. Remember from Sec. 5.2.3 that we need to provide a set of functions that characterize these profiles. Since there is only one character trait, safety, we only need to provide a single function $f(M, S)$ for that character trait³ and the number N of past steps to consider.

³ M is the memory matrix $((N + 1) \times 1)$ and S is the steps column vector $((N + 1) \times 1)$

The behavioral profiles implemented are presented below. These profiles are the same as in Examples 5.2.3. For convenience, we include the characteristic function of each of them. However, for a deeper explanation, refer back to the examples. In addition, each profile has been assigned an id that will be later used to identify it in the dataset.

0. Random

No-memory profile ($N = 0$) characterized by

$$f(M, S) = 0 \tag{6.1}$$

This profile does not give any importance to safety, taking decisions randomly.

1. Risky

No-memory profile ($N = 0$) characterized by

$$f(M, S) = -2 \tag{6.2}$$

This profile disfavors safety, favoring risk.

2. Safe (risk aversion)

No-memory profile ($N = 0$) characterized by

$$f(M, S) = 2 \tag{6.3}$$

This profile favors safety.

3. Risky NTM (but not too much)

Profile with $N = 2$ characterized by

$$\begin{aligned}
f(M, S) &= b_{1,1} + S^T((1 - M) \odot B) \\
&= b_{1,1} + \sum_{i=2}^{N+1} s_i(1 - m_{i,1})b_{i,1} \\
&= b_{1,1} + s_2(1 - m_{2,1})b_{2,1} + s_3(1 - m_{3,1})b_{3,1}
\end{aligned} \tag{6.4}$$

where $\odot$ is the element-wise multiplication, 1 is the $(N + 1) \times J$ matrix of ones, and B is

$$B = \begin{bmatrix} -8 \\ 8 \\ 6 \end{bmatrix}$$

This profile favors risk by nature but becomes safer the more risk it has taken in the past 2 actions.

4. Safe NTM (but not too much)

Profile with $N = 2$ characterized by

$$\begin{aligned}
f(M, S) &= S^T(M \odot B) \\
&= \sum_{i=1}^{N+1} s_i m_{i,1} b_{i,1} \\
&= b_{1,1} + s_2 m_{2,1} b_{2,1} + s_3 m_{3,1} b_{3,1}
\end{aligned} \tag{6.5}$$

where $\odot$ is the element-wise multiplication and B is

$$B = \begin{bmatrix} 8 \\ -8 \\ -6 \end{bmatrix}$$

This profile is the opposite of the *Risky NTM*. It favors safety by nature but becomes riskier the safer it has been in the past 2 actions.

Table 6.7: Example synthetic data collected

Run	Order	Decision Id	Action Taken	Label	Profile
0	0	1	0	NaN	0
0	1	2	1	1	0
0	2	3	1	0	0

6.4 Synthetic Dataset

Once the whole architecture has been implemented, we let the agents play the game by following the cycle of interactions presented in Sec. 4.2 while collecting the desired data as you would with a real player.

We let each agent play 100,000 times⁴(runs) the game without locking the random seed so the runs are not deterministic. We record the actions taken by the agents for each decision and use that information to build the dataset. Table 6.7 presents an example of the generated dataset. The following information has been recorded:

- Number of the run
- Order in which the decisions are taken
- Decision id
- character trait labels assigned to the action taken (safety for this example)
- Id of the agent's behavior profile

⁴We apply the label predictor model to the game's decision graph beforehand to obtain the predicted character traits for every action. Doing so speeds up the simulation substantially since it frees the model from performing the exact predictions for every run.

7 | Results

In this chapter we use the synthetic dataset generated in Ch. 6 and use it to train a variety of Machine Learning models. The trained models use the decisions taken throughout each run to classify each run by the behavior profile that generated it. By doing so, these models can later be used on real data to find the degree of similarity of the players' behavior with respect to the given behavior profiles.

7.1 Model Training and Analysis

We will now test the generated synthetic dataset by training a few Machine Learning models on it. The models trained will use the set of decisions taken throughout each run to classify each run by the behavior profile that generated it.

The data used by the models is the set of decisions taken throughout each run. This data can be obtained from the data collected (Table 6.7). Table 7.1 presents the shape of the training dataset. The models are given as input the decision taken for each question in a run (or -1 if the question does not apply for that run) and are trained to predict which behavior profile from Sec. 6.3 generated that run. Therefore, this task will be supervised since we have information about which behavior profile generated each run. This allows training models on the synthetic data and then using them on real data to classify the real players by behavior profile or to find the degree of similarity of the players' behavior with respect to the given behavior profiles. This is a strong point in favor of using this generation technique compared to other techniques, which would entail this task being unsupervised.

We have used the [scikit-learn](#) library to train and test a range of Machine Learning models on this task: decision tree classifier (DT), logistic regression

Table 7.1: Training dataset: choices taken per run^a

Run	Question 1	Question 2	...	Question 17	Profile
0	1	1	...	-1	0
1	1	1	...	1	0
2	0	1	...	-1	0

^a Number of the action taken or -1 if it does not apply

(LR), AdaBoost [6] classifier (Ada), random forest classifier (RF), k-nearest neighbors classifier (KNN), and multi-layer perceptron classifier (MLP). In the following analysis, it is important to remember that the dataset generated is balanced, having the same number of samples for each profile, which entails that the accuracy is reliable as a test metric. Table 7.2 summarizes the test metrics obtained by these models. All the tested models obtain substantially better results than random (0.2 global accuracy since there are five classes). The AdaBoost model obtains the worst results with 0.647 accuracy and 0.63 f1-score, while the best results are obtained by the decision tree and MLP models with 0.8 accuracy and 0.8 f1-score.

It can also be observed that the results vary depending on the class. Of particular interest are the results obtained for the random class, around 0.5 accuracy and f1-score. An explanation for this is that, since this profile acts completely random, it has no decision-making pattern. Therefore, each of the runs generated by this profile may be similar to those generated by other profiles, leading the models to classify those runs incorrectly. This is not a problem in the data but an inherent problem of random behavior. This is confirmed when comparing each model's results on the random profile with the rest of the profiles. Other models to differentiate random behavior from the rest must be trained to cope with this problem.

Except for the random profile, all classes obtain similar scores with around 0.8–0.9 accuracy and f1-scores. Moreover, the best per-class score is 0.93 accuracy and 0.87 f1-score (Risky NTM class, DT, and RF models). Except for AdaBoost, the worst per-class score is 0.79 accuracy and 0.78 f1-score (Risky class, KNN model). With these results, we can conclude that the generation of synthetic data has successfully trained models capable of using the decisions taken to differentiate the behavior profiles.

Once trained, these models can be used on real data to find the degree of similarity of the players' behavior with respect to the behavior profiles. Table 7.3 illustrates the prediction made by a model given the decisions a real player made.

Table 7.2: Behavior profile classification models test metrics

Profile	Metric	DT ^a	LR ^b	Ada ^c	RF ^d	KNN ^e	MLP ^f
Random	F1	0.583	0.506	0.488	0.581	0.548	0.577
	Accuracy	0.527	0.444	0.426	0.522	0.517	0.51
Risky	F1	0.819	0.805	0.466	0.819	0.784	0.82
	Accuracy	0.81	0.815	0.333	0.81	0.789	0.817
Safe	F1	0.842	0.83	0.765	0.842	0.82	0.843
	Accuracy	0.825	0.811	0.681	0.826	0.819	0.833
Risky NTM	F1	0.862	0.809	0.671	0.861	0.841	0.861
	Accuracy	0.928	0.858	0.912	0.929	0.86	0.921
Safe NTM	F1	0.881	0.821	0.762	0.88	0.861	0.881
	Accuracy	0.926	0.88	0.886	0.928	0.888	0.935
Global	F1	0.797	0.753	0.63	0.796	0.77	0.796
	Accuracy	0.802	0.761	0.647	0.802	0.774	0.803

^a Decision tree^b Logistic regression^c AdaBoost^d Random forest^e K-nearest neighbors^f Multi-layer Perceptron using adam optimizer

Table 7.3: Example prediction: Probabilities of run being generated by a policy

Random	Risky	Safe	Risky NTM	Safe NTM
0.069	0.083	0.001	0.846	0.001

The model's predictions provide the probability of those decisions being taken by a specific profile. This can be interpreted as the similarity between the real player's decisions and each given behavior profile. In the example presented, we could conclude that the real player presented in this run a *Risky NTM* profile (84.6% similarity), followed by a *Risky* and *Random* profile (8% and 7% respectively).

8 | Conclusions and Future Work

This last chapter gives an overview of this work's findings and conclusions. We also present future areas of research.

Some key contributions of this work have been incorporated into a scientific article titled "A Child's Play: An Agent-based Simulator to Protect Minors Online" [16], which has been accepted at a leading international conference within the field of computational social science. This article will be presented at the 8th International Conference on Computational Social Science (*IC²S²*) held at the University of Chicago (July 19–22, 2022).

8.1 Conclusions

Data generation is a technique that has been used for a long time to cope with the lack of real data and to improve the models trained on that data. However, most of the work done on data generation for games is unsuitable for games where the focus is on decision-making or where rewards are not applicable, such as serious games. We have provided a new approach suitable for these games and expect that this work will open up the exploration of data generation for decision-making games.

We have presented a model to generate synthetic data in games where the focus is on decision-making rather than obtaining the highest score or where rewards are not applicable (no "good" or "bad" decisions). This model meets the requirement stated in Sec. 1.1. Our model generates synthetic data by creating agents that imitate real players' behavior and gathering the data as they play the game. The agents are each given a behavior profile regarding several character traits which establishes the agent's policy. The agents select the actions that best

align with their profile’s character traits while also considering past actions. The behavior profiles used, the character traits that characterize them, and the number of past steps to consider are hyperparameters that allow the model to adjust to various applications and games. This approach does not require pre-existing data. Therefore, it does not suffer from the intrinsic bias and lack of variability that most techniques based on pre-existing data suffer. Furthermore, the agents do not follow a reward-driven approach since this would be unsuitable for games where the focus is on decision-making or rewards are not applicable. In addition, our model incorporates behavior information into the generated data and, therefore, into the models trained with the synthetic data. These models can then detect or analyze these behavior profiles on real data. In other words, our model transforms the task of analyzing the real players’ behavior from an unsupervised into a supervised task.

The model presented is divided into two main components, the environment or simulator (Sec. 5.1) and the agent (Sec. 5.2). Each of these components are independent and interact with each other through an interface (Sec. 4.2). This modular approach eases interdisciplinary collaboration by separating the story, NLP, and behavior profile modeling subtasks into independent components. The environment component takes care of the game’s story, while the agent component takes care of the decision-making. The agent component is subdivided into the label predictor (Sec. 5.2.2) and the decision maker (Sec. 5.2.3). The label predictor extracts the alignment between the character traits and a certain action given the current game state. The decision maker takes these alignment values and chooses the action that best aligns with its policy, given by the agent’s behavior profile while considering past actions. Thus, the label predictor takes care of the NLP task, while the decision maker characterizes the behavior profile to establish the policy followed by the agent.

The simulator, label predictor, and decision maker components can be modified or completely substituted without having to modify the other two. For example, only the simulator component would have to be changed when applying the same set of profiles to two different games. This makes our model highly reusable by reducing the effort needed to make modifications. Furthermore, even though this work has focused on text games, it can be easily extended into decision-making games with other types of information such as images or audio. The simulator component would then provide the visual or audio information as extra information so the label predictor can process it. The label predictor would then incorporate this data into its model as presented in Sec. 5.2.2.

We have illustrated the model’s application and advantages following the steps

from Sec. 1.1. We have implemented it on an existing serious text game (Sec. 6). We first adapted the text game, which is given in Yarn Spinner format, so it can be played programmatically using Python. This has been done by extending a Yarn-to-Python parser¹ (Sec. 6.1.2) and using Depth First Search to obtain the game’s decision graph (Sec. 6.1.3). The decision graph is then incorporated into the model through the environment component. Next, we built and trained an NLP model for the label predictor (Sec. 6.2) following the architecture presented in Sec. 5.2.2. We confirmed the need to use transfer learning for our model to learn, given the small size of the dataset (1052 samples). Thus, we incorporated a pre-trained BERT model for this purpose. For the decision maker, we implemented three no-memory profiles and two memory profiles (Sec. 6.3). With the five agents implemented, we let each of them play 100,000 runs while gathering data to build the synthetic dataset (Sec. 6.4).

The synthetic dataset generated is then used to train various Machine Learning models: decision tree classifier, logistic regression, AdaBoost classifier, random forest classifier, k-nearest neighbors classifier, and multi-layer perceptron classifier. These models are given as input the decisions taken on each run and are trained to predict the behavior profile that generated it (Sec. 7.1). Since our dataset does have this information, this task is supervised. This allows training models on the synthetic data and then using them on real data to classify the real players by behavior profile or to find the degree of similarity of the players’ behavior with respect to the given behavior profiles. This is a strong point in favor of using this generation technique compared to other techniques, which would entail this task being unsupervised. The results obtained by the models are substantially better than random (see Table 7.2), reaching global scores of 0.8 accuracy and f1-score. Furthermore, except for the random profile and AdaBoost, all profiles and models obtain similar scores with around 0.8–0.9 accuracy and f1-scores. With these results, we can conclude that the generation of synthetic data has successfully trained models capable of using the decisions taken to differentiate the behavior profiles.

8.2 Future Work

This section provides an overview of some areas of research aimed at improving the quality of the generated synthetic data and increasing the model’s applicability

¹The code is available on <https://github.com/vibalcam/yarn-pygame>. This parser has been extended from a base parser [17].

and usability in decision-making games.

Label Predictor. This area seeks to improve the label predictor model to adjust better to the player’s perception of the actions.

- Many text games have actions that appear many times throughout the game producing imbalances in the training of the label predictor model. To cope with this, we can organize the dataset by unique actions and give the model one random sample of each unique action at each epoch.
- To reduce the label predictor’s bias towards the game being used, we can train the model on a variety of games that cover similar fields or have similar dictionaries. Another option is to use tools such as TextWorld [1] to generate the various games used for the training.
- Other models and techniques can be developed to tackle this task.

Decision Maker Memory. This area seeks to improve the memory used by the decision maker to adjust better to the real player’s perception of past decisions. As mentioned in Sec. 5.2.3, the agent’s perception of past decisions can also be influenced by the context, or change in context, obtained after an action has been taken. By analyzing the context after an action, the agent perceives the consequences of the action they have chosen. We can incorporate this information into the model by using another NLP model that analyzes the changes in context after an action is taken. This information can be included as an additional column in the memory matrix (Equation (5.1)) as if it were an additional character trait.

Decrease Implementation Time. This area seeks to decrease the time needed to implement the model in a text game. The graph simulator and the NLP model presented shorten the implementation times. However, further improvements can be made in this aspect. For example, manually labeling the actions for the label predictor model can be too time-consuming when the game is too large. For these situations, we need models or techniques capable of labeling the actions given a set of character traits in an automated manner. These techniques do not need to be perfect, but good enough to cover the cost of not having to label the data manually.

Other Decision-making Games. This area consists in testing this model’s effectiveness in other decision-making games. The label predictor presented in Sec. 5.2.2 is capable of dealing with other types of information apart from text, making it possible to apply the same architecture to other kinds of decision-making

games. By doing so, we can test if this technique can deliver consistent positive results for various games.

Real Data Comparison. This area consists in testing this model’s effectiveness in a real scenario. Testing the synthetic data generated against real data assures the effectiveness and practicality of this technique for data generation in text games.

Bibliography

- [1] Marc-Alexandre Côté et al. “TextWorld: A Learning Environment for Text-based Games”. Version 2. In: (2019). DOI: [10.48550/ARXIV.1806.11532](https://doi.org/10.48550/ARXIV.1806.11532). URL: <https://arxiv.org/abs/1806.11532> (visited on 05/31/2022) (cit. on pp. 8, 54, 72).
- [2] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. Version 2. In: (2018). DOI: [10.48550/ARXIV.1810.04805](https://doi.org/10.48550/ARXIV.1810.04805). URL: <https://arxiv.org/abs/1810.04805> (visited on 06/05/2022) (cit. on p. 14).
- [3] Cem Dilmegani. *Synthetic Data Generation: Techniques, Best Practices & Tools*. AI Multiple. July 15, 2020. URL: <https://research.aimultiple.com/synthetic-data-generation/> (visited on 03/11/2022) (cit. on p. 8).
- [4] Ian J. Goodfellow et al. “Generative Adversarial Networks”. Version 1. In: (2014). DOI: [10.48550/ARXIV.1406.2661](https://doi.org/10.48550/ARXIV.1406.2661). URL: <https://arxiv.org/abs/1406.2661> (visited on 05/31/2022) (cit. on p. 8).
- [5] Ramy Hammady and Sylvester Arnab. “Serious Gaming for Behaviour Change: A Systematic Review”. In: *Information* 13.3 (2022). ISSN: 2078-2489. DOI: [10.3390/info13030142](https://doi.org/10.3390/info13030142) (cit. on p. 1).
- [6] Trevor Hastie et al. “Multi-Class AdaBoost”. In: *Statistics and Its Interface* 2.3 (2009), pp. 349–360. ISSN: 19387989, 19387997. DOI: [10.4310/SII.2009.v2.n3.a8](https://doi.org/10.4310/SII.2009.v2.n3.a8). URL: <http://www.intlpress.com/site/pub/pages/journals/items/sii/content/vols/0002/0003/a008/> (visited on 06/29/2022) (cit. on p. 66).
- [7] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1, 1997), pp. 1735–1780. ISSN: 0899-7667, 1530-888X. DOI: [10.1162/neco.1997.9.8.1735](https://doi.org/10.1162/neco.1997.9.8.1735). URL: <https://direct.mit.edu/neco/article/9/8/1735-1780/6109> (visited on 06/05/2022) (cit. on p. 12).

-
- [8] Ida Bagus Kerthyayana Manuaba. “Text-Based Games as Potential Media for Improving Reading Behaviour in Indonesia”. In: *Discovery and innovation of computer science technology in artificial intelligence era: The 2nd International Conference on Computer Science and Computational Intelligence (ICCSCI 2017)* 116 (Jan. 1, 2017), pp. 214–221. ISSN: 1877-0509. DOI: [10.1016/j.procs.2017.10.041](https://doi.org/10.1016/j.procs.2017.10.041). URL: <https://www.sciencedirect.com/science/article/pii/S1877050917320896> (cit. on p. 1).
- [9] L. R. Rabiner. “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition”. In: *Proceedings of the IEEE* 77.2 (Feb. 1989), pp. 257–286. ISSN: 1558-2256. DOI: [10.1109/5.18626](https://doi.org/10.1109/5.18626) (cit. on p. 6).
- [10] Ilya Loshchilov and Frank Hutter. “Decoupled Weight Decay Regularization”. Jan. 4, 2019. arXiv: [1711.05101](https://arxiv.org/abs/1711.05101) [cs, math]. URL: <http://arxiv.org/abs/1711.05101> (visited on 02/13/2022) (cit. on p. 58).
- [11] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. Version 1. In: (2013). DOI: [10.48550/ARXIV.1312.5602](https://doi.org/10.48550/ARXIV.1312.5602). URL: <https://arxiv.org/abs/1312.5602> (visited on 05/31/2022) (cit. on p. 6).
- [12] Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. “Language Understanding for Text-based Games Using Deep Reinforcement Learning”. Version 2. In: (2015). DOI: [10.48550/ARXIV.1506.08941](https://doi.org/10.48550/ARXIV.1506.08941). URL: <https://arxiv.org/abs/1506.08941> (visited on 03/11/2022) (cit. on p. 8).
- [13] Kolby Nottingham. *Reinforcement Learning in Text-Based Games*. Voice Tech Podcast. June 15, 2019. URL: <https://medium.com/voice-tech-podcast/reinforcement-learning-in-text-based-games-f9ddad9e87fd> (visited on 03/11/2022) (cit. on p. 7).
- [14] Christopher Olah. *Understanding LSTM Networks*. Aug. 27, 2015. URL: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> (visited on 06/05/2022) (cit. on pp. 7, 12, 13, 14).
- [15] Adam Paszke. *Reinforcement Learning (DQN) Tutorial — PyTorch*. URL: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html (visited on 03/11/2022) (cit. on p. 6).
- [16] Jaime Pérez et al. “A Child’s Play: An Agent-based Simulator to Protect Minors Online”. In: IC2S2 (8th International Conference on Computational Social Science). Chicago, 2022 (cit. on p. 69).
- [17] Robert Pfeiffer. *Yarn.Py, a Python Runtime for Yarn Dialogue*. Oct. 29, 2020. URL: <https://github.com/robertpfeiffer/yarn-pygame> (visited on 06/17/2022) (cit. on pp. 50, 71).
- [18] RAYUELA - A Fun Way to Fight Cybercrime. RAYUELA. URL: <https://www.rayuela-h2020.eu/> (visited on 07/06/2022) (cit. on p. 78).

-
- [19] Alex Sherstinsky. “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network”. In: *Physica D: Nonlinear Phenomena* 404 (Mar. 2020), p. 132306. ISSN: 01672789. DOI: [10.1016/j.physd.2019.132306](https://doi.org/10.1016/j.physd.2019.132306). arXiv: [1808.03314](https://arxiv.org/abs/1808.03314) [cs, stat]. URL: <http://arxiv.org/abs/1808.03314> (visited on 05/29/2022) (cit. on p. 7).
- [20] Luciano Strika. *Markov Chains: How to Train Text Generation to Write like George R. R. Martin*. Towards Data Science. Oct. 25, 2019. URL: <https://towardsdatascience.com/markov-chains-how-to-train-text-generation-to-write-like-george-r-r-martin-cdc42786e4b6> (visited on 03/11/2022) (cit. on p. 6).
- [21] Tech-Effigy. *Markov Chains - Explained*. Aug. 1, 2015. URL: <https://techeffigyutorials.blogspot.com/2015/01/markov-chains-explained.html> (visited on 03/11/2022) (cit. on p. 6).
- [22] Ana Leticia Urbistondo Murua. “Creación de Un Videojuego Interactivo Para Educar En El Uso Responsable de Las Redes Sociales”. BA thesis. Universidad Pontificia Comillas, Escuela Técnica Superior de Ingeniería (ICAI), 2020. URL: <https://repositorio.comillas.edu/xmlui/handle/11531/41473> (visited on 02/13/2022) (cit. on pp. 48, 54).
- [23] Ashish Vaswani et al. “Attention Is All You Need”. Dec. 5, 2017. arXiv: [1706.03762](https://arxiv.org/abs/1706.03762) [cs]. URL: <http://arxiv.org/abs/1706.03762> (visited on 03/24/2022) (cit. on pp. 7, 14).
- [24] Christopher J. C. H. Watkins and Peter Dayan. “Q-Learning”. In: *Machine Learning* 8.3 (May 1, 1992), pp. 279–292. ISSN: 1573-0565. DOI: [10.1007/BF00992698](https://doi.org/10.1007/BF00992698). URL: <https://doi.org/10.1007/BF00992698> (cit. on p. 6).

A | Alignment with the Sustainable Development Goals (SDGs)

The [Sustainable Development Goals](#) are a collection of global goals that world leaders collectively adopted to promote prosperity while protecting the planet. They recognize that ending poverty must go hand-in-hand with strategies that build economic growth and address a range of social needs including education, health, social protection, and job opportunities, while tackling climate change and environmental protection. This project encompasses 17 individual as part of the 2030 Agenda for Sustainable Development which set out a 15-year plan to achieve them.

This work presents a model to generate synthetic data on decision-making games. Therefore, its alignment with the SGD goals matches that of the project for which this data generation technique is used. Since this work is framed within the European H2020 [RAYUELA](#) project, we will focus on the goals tackled by this work in the context of the RAYUELA project.

- **Goal 4 Quality Education.** This goal seeks to ensure inclusive and equitable quality education and to promote lifelong learning opportunities for all. This is one of RAYUELA's primary goals. RAYUELA seeks to contribute by making the Internet a safer and better place for children. It seeks to accomplish this goal by educating young people on the benefits, risks, and threats of the internet through a text game.
- **Goal 3 Good Health and Well-being.** This goal seeks to *ensure healthy lives and promote well-being for all at all ages*. This is one of RAYUELA's primary goals. In addition to educating young people, RAYUELA seeks to understand better the drivers and human factors affecting specific relevant ways of cybercriminality. By doing so, it can analyze risk factors and protect those who are most vulnerable. For this purpose, it will gather data from

the text game and train models with this goal. This is where this work can impact the most, in the task of training the models that will be used to analyze risk factors.

- **Goal 11 Sustainable Cities and Communities.** This goal seeks to *make cities inclusive, safe, resilient and sustainable*. The objective of educating and analyzing human drivers and factors affecting cybercriminality is to prevent and mitigate this kind of behavior. "51% of EU citizens feel not at all or not well informed about cyber threats and 86% of Europeans believe that the risk of becoming a victim of cybercrime is rapidly increasing. On the other hand, Law enforcement has noted that more and more teenagers and young people are increasingly committing cybercrimes" [18]. Therefore, there is no doubt that cybercriminality is a major focus for cities to be safe.

B | Decision Graph of the Online Grooming Game

