Attack on RecSys: a method using Generative Nets

Vicente Castro Solar
Pontificia Universidad Católica de Chile
vvcastro@uc.cl

Nicolás Casassus
Pontificia Univerisdad Católica de Chile
ncasassus@uc.cl

Abstract

Los Sistemas Recomendadores se pueden encontrar en todo nivel de la interacción usuarios-plataforma, siendo estos uno de los generadores de utilidades más importantes para muchas empresas. En este sentido, la seguridad de estos sistemas debe estar asegurada frente a distintos ataques a los que estos modelos pueden ser sometidos. En este paper exploraremos los métodos y efectos de una forma específica de ataque: adversarial attacks. Es específico, exploramos primero un método de generación de usuarios falsos, cuya distribución de datos sea similar -indetectable- de la distribución de real de usuarios. Además, a partir de estos y a través distintas técnicas de aprendizaje de máquina tradicional, somos capaces de aprender tipos de ataque que, en general, tienen como objetivo shiftear de alguna forma los ratings o recomendaciones que reciben los usuarios reales de la aplicación. Los resultados de esta investigación parecen relevantes, pues pueden ayudar no solo a mejorar la seguridad de Sistemas Recomendadores, sino que también podrían servir para una mejor inicialización de estos modelos o atacar otros problemas comunes en estas implementaciones.

I. Introducción

¿Hasta qué nivel podríamos engañar a un Sistema Recomendador? y ¿Existirá una forma de hacer esto sin ser detectados? En la actualidad, todas las grandes aplicaciones con las que interactuamos cuentan con un algoritmo recomendador. Ya sea si está viendo una película, al momento de hacer una compra o estar navegando en una red social, un sistema recomendador está presente para ayudarte a encontrar lo que te interesa ver y, en cierta forma, también moldea tus gustos con ese objetivo. Es por esto que la seguridad en sistemas recomendadores es un tema problemático y del cuál nos debemos hacer cargo desde la construcción del modelo.

En particular, uno de los problemas de seguridad más estudiados en el área es el de los diversos "ataques" que se le pueden realizar a un RS (*Recommender System*) y, si bien existen diversos métodos, uno de los más empleados y estudiados es el de los denominados *shilling attacks*, esto es, la inyección de usuarios falsos en los datos de pre entrenamiento (proceso llamado *poisoning attack*) con el fin de que, en producción, el modelo tenga un desempeño no deseado. Este tipo de ataque ha pasado por distintas iteraciones a lo largo de los años, desde la inyección de usuarios con puros *ratings* positivos para los *items* a ideas un tanto más complejas, con usuarios que parecen más reales. Aún así, todos estos métodos se basan en la creación de usuarios "a mano", una técnica que, desde el trabajo de Li et al. [1], donde se genera por primera vez un ataque optimizado por algoritmos de *machine learning*, ha perdido peso. Este nuevo enfoque ha abierto la investigación a métodos adversarios de ataque, donde se busca una perturbación pequeña e imperceptible, que introducida en los datos de entrenamiento, puede causar el ataque esperado. Es esta última idea, también llamada *adversarial attacks* la que nosotros estudiamos en esta investigación.

En este sentido, en este trabajo nos centramos en dos procesos adversarios que son parte de la generación de este ataque: (1) cómo se pueden introducir usuarios falsos de forma no detectable, es decir, cómo hacer que el ataque pase desapercibido en el modelo y (2) cómo a partir de esta distribución de usuarios generados, podemos encargarnos de realizar un ataque. Para el primero de estos objetivos, exploramos distintos modelos generativos (específicamente GANs) y comparamos según métricas de distancia entre la distribución de usuarios

falsos y la distribución real. Luego, para el segundo objetivo rescatamos un método de aprendizaje propuesto en [2] (específicamente, el cálculo del gradiente), proponiendo unas mejoras -desviaciones- al algoritmo y comparamos empíricamente los resultados.

II. Trabajos Relacionados

Si bien existe una extensa literatura en cuanto a *shilling attacks* y también de métodos de seguridad frente a estos ataques, los trabajos relacionados a Sistemas Recomendadores con modelos generativos son escasos y enfocados, especialmente, en hacer recomendaciones.

Si nos enfocamos en esquemas de ataques, el método expuesto en [1] presenta bastantes similitudes con el que nosotros trabajaremos, pues si bien, el *sampleo* de la matriz de interacciones falsas difiere de nuestro método, la idea de aprender un ataque con un método de *machine learning* tradicional es prácticamente el mismo razonamiento aplicado en [2], que es el que da forma al algoritmo que nosotros utilizaremos.

Por otro lado, evaluando los trabajos que se basan en *adversarial attacks*, existen solo dos *papers* que relacionan con este tipo de ataques con GANs (en tabla 3 de [6]) y solo uno se basa en la generación de perfiles de usuarios (es decir, usuarios falsos). En este sentido [2], es un trabajo que se aproxima bastante a nuestro método y que nos da una gran base para llevar a cabo esta investigación. También ha un punto de referencia para comparar nuestros resultados.

III. Método

Teniendo dos objetivos primordiales: la generación de usuarios realistas y el entrenamiento de estos en pos de un ataque, decidimos atacarlos de forma separada. Así, podemos dividir nuestra implementación en dos partes: (1) la utilización de métodos generativos para la generación de usuarios falsos y "realistas", y (2) el uso de ML tradicional para el aprendizaje del ataque desde la optimización de una distribución de usuarios falsos:

Generación de usuarios

La generación de usuarios tenían como objetivo el aprendizaje y la imitación de la distribución de interacciones subyacentes a la distribución de datos reales que se le pasen al modelo. Para lograr este objetivo, decidimos centrarnos en el uso de *Generative Adversarial Networks* (GANs), un poderoso modelo generativo descrito por primera vez por Goodfellow en [3]. Si bien, al día de hoy, este tipo de redes están lo suficientemente desarrolladas como para atacar problemas bastante más complejos que el acá presentado, decidimos emplear dos arquitecturas más simples y tradicionales en el campo: *VanillaGAN* (como llamamos a la implementación original [3]) y *DCGAN* o *Deep Convolutional Generative Adversarial Networks* (modelo presentado en [3] y especialmente bueno en el trabajo con imágenes).

Este tipo de modelos cuentan con dos módulos de redes neuronales un *Generador* (G) y un *Discriminador* (D), siendo el objetivo del entrenamiento de ambos la competencia en lo que se llama un juego min-max. En nuestra aplicación, el generador deberá aprender a mapear, desde un vector de ruido, un vector de dimensiones $[1, m]$ (con m el número de *items* en el *dataset*) que se asemeje lo suficientemente a la distribución real de los datos (de esto se asegura la estructura interna del discriminador).

Si bien ambas implementaciones siguen una estructura similar a la publicada en sus *papers* originales, se hicieron modificaciones tanto en su arquitectura como en las funciones de pérdida del generador para mejorar directamente el ajuste a algunos aspectos de los datos. Por ejemplo se agregó a la pérdida de G un factor de corrección según la media y el número de interacciones en la distribución de cada *item* generado y su contra parte real. Esta "regularización" se puede calcular como el error cuadrático medio de ambas distribuciones

$$L = \frac{1}{|I|} \sum_{i=0}^{|I|} (F_i - R_i)^2 \tag{1}$$

Donde F_i y R_i son el vector de datos generados y los reales (y de la misma dimensión) para el *item* $i \in I$, respectivamente. Es importante notar que esta relación es necesaria hacerla a nivel de *items* y no de usuarios, Pues es este vector el que guarda la verdadera

relación que queremos aprender: la distribución de interacciones de los usuarios (todos los usuarios).

Aprendiendo el ataque

El segundo objetivo es el de modificar de forma superficial la matriz de *ratings* de los usuarios falsos generados en el paso anterior, de tal forma que estos afecten las recomendaciones de los usuarios originales de un modelo recomendador. Para esto se sigue como esquema general un aprendizaje tradicional sobre los valores (pesos) en la matriz de usuarios generados U. Para lo cuál se debe definir una *función de ataque* f_A, función objetivo a optimizar el ataque y la que, en definitiva, moldea todo el proceso de aprendizaje (ie: calcula el *rating* promedio de un *item* específico para un usuario específico).

Así, por cada época de entrenamiento se busca actualizar los valores de los *ratings* falsos según el gradiente de la función de ataque y la matriz de dicha iteración U_i. Este paso se puede describir por la siguiente ecuación:

$$U_{i+1} = U_i - \eta \nabla_{U_i} f_A(U_i)$$

La problemática de este proceso es bastante significativa, pues no hay un método directo para hacer el cálculo de la gradiente, recordemos que, para calcular f se requiere pasar por el modelo recomendador, lo que funciona como una "caja negra" a la hora de tratar de calcular cualquier gradiente en el grafo de computo.

A partir de esto optamos por estudiar la metodología estudiada en [2], cuyo problema basal es similar al nuestro y que tiene un forma para el cálculo del gradiente desde una descomposición matricial con `SVD`. El método que ellos proponen es el de calcular el gradiente de la función en distintas direcciones según la ecuación tradicional de derivada:

$$\nabla_{Ui} f_A(U_i) = \frac{1}{2} \sum_{k=0}^{K} \frac{f_A(U_i + \alpha \tilde{U}_i) - f_A(U_i)}{\alpha} \qquad (2)$$

A este método tradicional de cálculo lo llamaremos *Mean Latent Gradients*, también se ofrece un método extra: *Best Latent Directions*, junto con otras aproximaciones para el calculo de las direcciones (PCA, NMF y TruncatedSVD). El nuevo método propuesto consiste en, de un total de K direcciones posibles, seleccionar solo las b direcciones con mayor y menor gradiente, para luego ponderarlos de acuerdo a la ecuación (1).

IV. Experimentos

Tal como en nuestra metodología, dividimos nuestra experimentación en dos segmentos. El primero, orientado a medir la generalización que los distintos modelos de GANs utilizados nos entregaban y luego, un segundo *set* para medir la eficacia del método de aprendizaje para el ataque.

Dataset

Trabajamos con dos *datasets* creados para la investigación: MovieLens (100K) y MovieLens (1M). Ambos conformados por distintos usuarios de MovieLens y los *ratings* que estos hicieron sobre distintas películas presentes en la plataforma. En específico:

MovieLens(100K) es un dataset que contiene 943 usuarios, 1.682 películas y 100.000 interacciones, dando un *sparsity* de 93.695%, lo cual es relativamente bajo para los *datasets* usuales de recomendación. En la Figura 1, se puede ver la distribución de interacciones por usuarios, lo que es importante dado que es lo que nuestro generador tratará de aprender.

Por otra parte, MovieLens (1M) es un *dataset* bastante más grande, con 6.040 usuarios, 3.706 películas y 1.000.209 interacciones, dando un *sparsity* de 95.532%. Esto nos será útil ya que la cantidad de usuarios es bastante más elevada que en 100K lo que nos ayudará a analizar el desempeño de nuestro generador frente a un mayor número de datos (nos gustaría que el rendimiento medido mejorara).

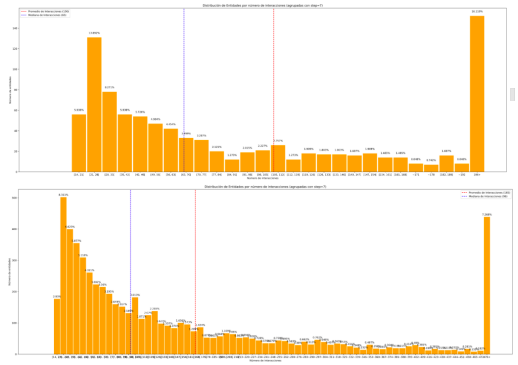

Figure 1: *Distribución de interacciones por usuarios en MovieLens100K (arriba) y Movielens1M (abajo)*

Evaluación (Generación de Usuarios)

Para evaluar la distribución de usuarios generados comparamos tres métricas: *Jensen-Shannon Divergence*, *Wasserstein Distance* y *Cosine Distance*. Siendo las dos primeras métricas de distancia entre distribuciones y la última una de distancia puntual. En este sentido, mientras menor sean los valores en estas métricas, las distribuciones están más cerca.

Finalmente, cada gráfico presentado representará el promedio de 5 iteraciones de entrenar el modelo por el número de épocas que cada uno señala.

Attack Learning

Para medir la eficacia del método de aprendizaje definimos tres funciones de ataque las cuales serán las funciones "a optimizar" durante el proceso de entrenamiento. La evaluación final se medirá de acuerdo a la variación de los valores de la función frente a las distintas épocas del entrenamiento. Las funciones a definir son:

1. Reducción del *rating* promedio en un grupo de *items* I.

2. Eliminar de las recomendaciones de un grupo de usuarios U los *items* de un subconjunto I.

Si bien se pueden definir y estudiar más funciones, como aumentar métricas o el *rating* promedio de una interacción, este tendrá como base común a una de esas funciones.

Finalmente, todos los métodos estudiados utilizaron el mismo modelo recomendador: *FunkSVD* al que se le agregó la utilización de *bias* por usuario e *item*.

V. Resultados

Generación de Usuarios: VanillaGAN

El primer experimento realizado fue utilizando la arquitectura VanillaGAN junto con el dataset de MovieLens 100K. Luego de un entrenamiento de alrededor de 20000 épocas, se obtuvieron los siguientes valores de las funciones de pérdida de ambos modelos (en Figura 2):

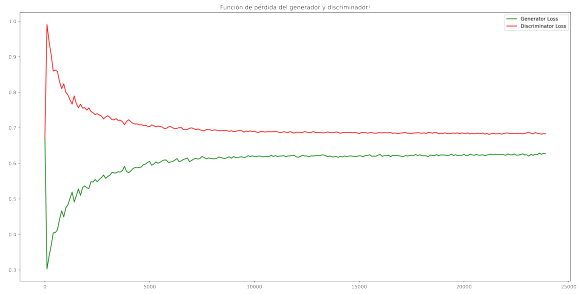

Figure 2: *VanillaGAN: Loss en MovieLens (100K)*

Podemos notar que ambas funciones de pérdida tienen tanto una simetría, como una gran desviación al inicio. Esto nos dice que el modelo no tiene una inicialización muy buena para el problema, aún así, al cabo de algunas épocas, se logra converger asintóticamente. Sobre las métricas evaluadas para este modelo (Figura 3):

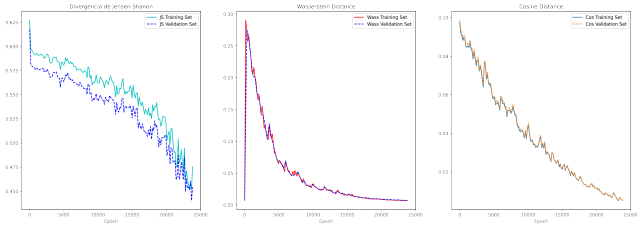

Figure 3: *VanillaGAN: métricas para MovieLens (100K)*

En los tres gráficos presentados, se alcanzan valores regularmente bajos, si bien el descenso no es tan pronunciado con *Jensen-Shannon*, esta métrica también tiene una clara tendencia a la baja. Esto, junto con lo excelente valores alcanzamos en *Wasserstein distance* nos dice que el modelo está efectivamente aprendiendo de buena forma la distribución de los datos. Finalmente, se testeo sobre este *dataset* la inclusión del parámetro de regularización (pérdida) en el generador según la distribución de datos reales:

Ambas imágenes representan un entrenamiento de 5000 épocas (promediadas 5 veces), junto con el número promedio de interacción y el promedio de *ratings* dado por los usuarios falsos. En general, se observa que el modelo con regularización genera usuarios con menos cantidad de *ratings* (orden de los 100 ratings por usuario), mientas que el modelo sin regularización tiene usuarios con una cantidad bastante mayor de usuarios. A partir de esto, podemos

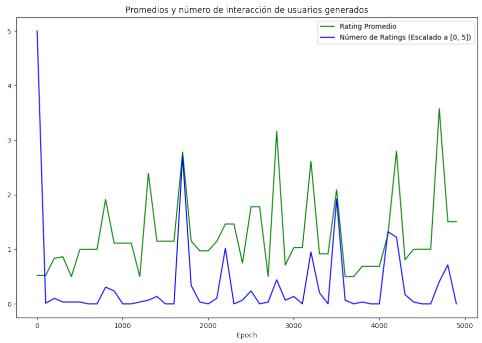

Figure 4: *VanillaGAN: con regularización para MovieLens (100K)*

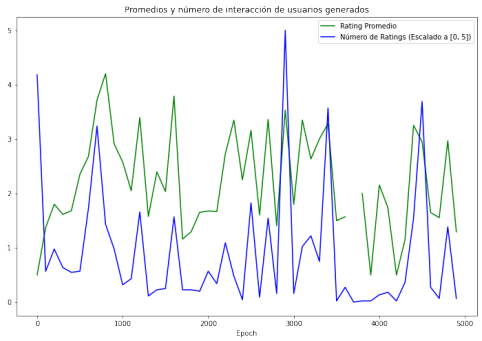

Figure 5: *VanillaGAN: sin regularización para MovieLens (100K)*

afirmar que, en general, el modelo con regularización presenta usuarios con una cantidad de interacciones más reales.

Luego, usando la misma arquitectura de GAN, pero con el *dataset* de MovieLens(1M). Análogamente visualizamos las pérdidas:

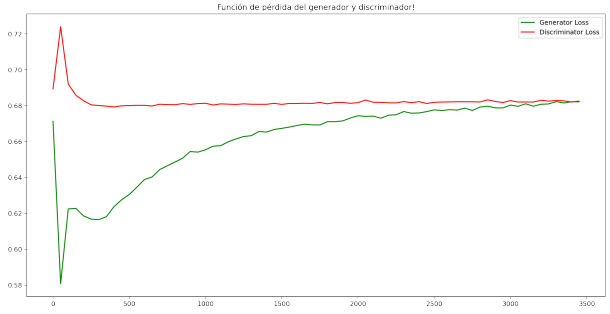

Figure 6: *VanillaGAN: Loss en MovieLens (1M)*

Se puede observar un comportamiento un tanto diferente al del modelo con 100K, acá, el discriminador tiene, durante casi todo su entrenamiento, una pérdida constante. Aún así, llega el momento en que el generador empieza a ser capaz de engañar correctamente al discriminador y convergen a un valor bastante similar en la pérdida.

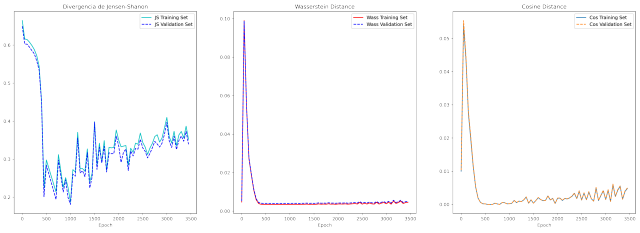

Figure 7: *VanillaGAN: métricas para MovieLens (1M)*

A diferencia de 100K, la métrica de *Jensen-Shanon* comienza con una bajada abrupta hasta la época 1000, pero a medida que se sigue el entrenamiento el valor la métrica empieza a deteriorarse. Aún así, las métricas de *Wasserstein* y Cosine se mantienen constantes y bajas durante casi toda la etapa de entrenamiento.

Generación de Usuarios: DCGAN

DCGAN es una arquitectura con bastante más complejidad que VanillaGAN, en este sentido, se tuvo que hacer un *reshape* del vector de *ratings*, pues, dado que se trabajan con convoluciones, el Discriminador del modelo debía recibir una matriz de valores (qué es lo mismo que una imagen de un solo canal). Para MoviLens100k se usó la dimensionalidad 29×58 ya que 29 era el mínimo divisor del número de *items*. En el entrenamiento, esta arquitectura resultó ser bastante lenta, por lo que se entrenó por 1500 épocas (x5 dadas las repeticiones para calcular un valor no azaroso). La pérdida en el entrenamiento se ve de la siguiente manera:

Se puede ver que, en gran parte del proceso el generador tuvo bastante problemas y el discriminador se mantuvo con una pérdida casi nula. Aún así, cerca de la época 800 empieza a disminuir considerablemente al pérdida de G y ha existir un leve aumento en las de D. Sobre las métricas de distancia:

Podemos que los resultados son bastante desalentadores (comparativamente) en las tres métricas presentadas, en este sentido, puede ser que el modelo empleado como generador no haya sido lo suficientemente complejo para enfrentarse al discriminador (dejándolo "incapaz de aprender"), o que recién cerca

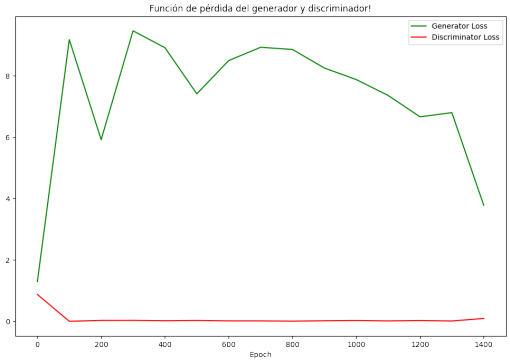

Figure 8: *DCGAN: Loss en MovieLens (100K)*

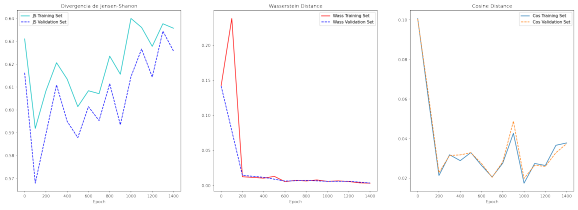

Figure 9: *DCGAN: métricas para MovieLens (100K)*

del final de las 1500 épocas entrenadas este haya logrado generalizar algo en los datos.

Por otro lado, si hacemos el mismo análisis hecho en VanillaGAN sobre las distribuciones de interacciones y promedios de *ratings* (Figuras 10 y 11), vemos una gran diferencia entre ambos modelos. DCGAN, tanto regularizado como no regularizado, tiene un *rating* promedio bastante más elevado que su contra parte y, además, tiene con una menor cantidad de interacciones por usuario.

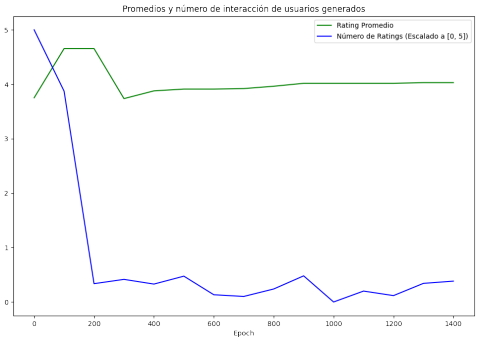

Figure 10: *DCGAN: con regularización para MovieLens (100K)*

Dado todos estos resultados, decidimos utilizar el modelo de VanillaGAN como generados para los próximos experimentos:

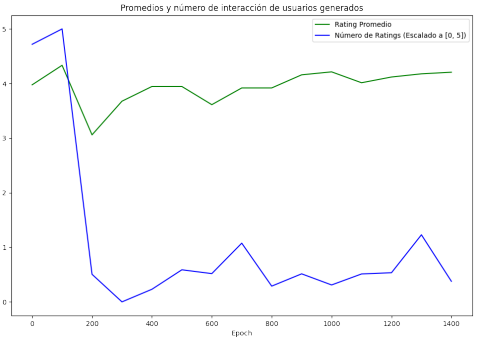

Figure 11: *DCGAN: sin regularización para MovieLens (100K)*

Resultados: Attack Learning

Los experimentos realizados se dividen tanto por la función objetivo que se busca optimizar, por el método mediante el cuál se calcula el gradiente y también, por el método de *Matrix Factorization* empleado.

La primera función estudiada, cómo reducir el *rating* promedio que un grupo arbitrario de usuarios asigna a un grupo de items, se resolvió con una función de ataque igual a la suma o promedio de todas las interacciones objetivos. A partir de esto, se obtuvieron los sigueinte resultado en el valor de la función en el tiempo:

Figure 12: *Mean Latent Gradients: con 16 direcciones*

Como se puede ver, en general el desempeño es bastante malo usando ambos algoritmos, por un lado, la Figura 13 como el algoritmo se queda atrapado en un mínimo local, mientras que, para *Mean Latent Gradients* la función no sigue un camino claro de convergencia.

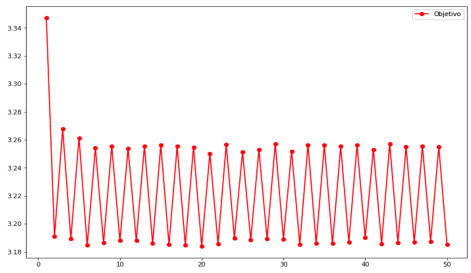

Figure 13: *Best Latent Directions: con 16 direcciones, seleccionando las 6 mejores*

Desde estos resultados surge la hipótesis de que la función de ataque, que en otras palabras es la pérdida del algoritmo, podría performar mejor al estar acompañada de algún término de regularización (por ejemplo, descantar el el descenso en los *ratings* que nos son de los grupos objetivos).

Para la segunda función objetivo: Eliminar de las recomendaciones de un grupo de usuarios la mayor cantidad de *items* de un grupo objetivo I. Empleamos como función de ataque al número de veces que los *items* de I aparecen en los recomendados de todos los usuarios de U (es la suma de de todos las apariciones). Se utilizaron tres combinaciones distintas, todas chequeando por 8 direcciones latentes de la matriz U (en referencia a los $\hat{U}_i$) y se obtuvieron los siguientes resultados:

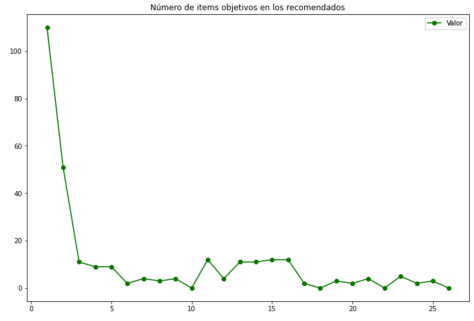

Figure 14: *Mean Latent Gradients: con 8 direcciones*

De estos, notemos que los primeros dos algoritmos tiene una convergencia relativamente rápida a valores pequeños de la función de ataque. En cambio, el tercer método no converge y se queda siempre rondando los 300 items.

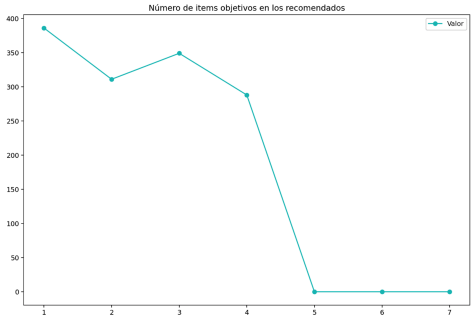

Figure 15: *Best Latent Directions: con 8 direcciones, seleccionando las 4 mejores*

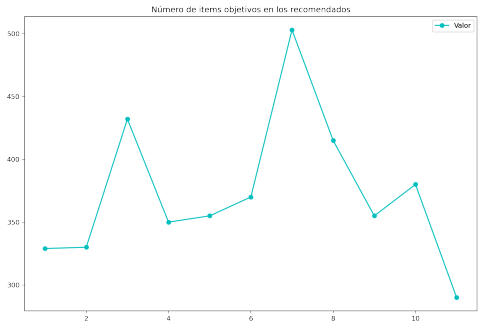

Figure 16: *Best Latent Gradients: con 8 direcciones, seleccionando la más positiva y con factorización NMF*

VI. Conclusiones y Trabajo Futuro

Estudiar la forma en la que se puede atacar a los sistemas recomendadores es un proceso vital para aprender a defender a estos modelos. Aún más, dado el impacto que esta técnicas pueden tener en un ambiente tan tecnologizado y polarizado como lo son las redes sociales, es necesario construir sistemas que sean capaces de identificar posibles amenazas y actuar ante ellas (en el caso de los *bots*, por ejemplo).

Se propuso el objetivo de generación usuarios falsos con una distribución de interacciones lo suficientemente parecida a un los usuarios reales de la aplicación como para ser "indetectables", para esto se propone el uso de métodos generativo, específicamente, dos arquitecturas de GAN, relativamente simples, pero que mostraron buenos resultados a la hora de analizar las métricas propuestas. En este sentido, los métodos de generación de usuarios falsos logran correctamente este objetivo, es decir, son indetectables de sus matriz de

interacciones.

Por otra parte, sobre el método de *attack learning* propuesto y los resultados obtenidos a partir de este. El esquema es bastante inestable y su eficiencia está condicionada, en gran parte, a la función que se quiera optimizar y a los grupos de usuarios que los que se quieran afectar. Aún así, si bien hay una clara ganancia por la simplicidad con la que se pueden definir las funciones de ataque, esta falta de complejidad juega en contra, dejando la función atrapada en mínimos locales y presentando otros problemas de optimización. Siendo esta la mayor área de avance que queda para una siguiente aproximación al campo.

Trabajo Futuro

En un futuro, como investigadores creemos que hay tres áreas en las que se puede avanzar. Primero, en la generación de usuarios falsos. Actualmente los usuarios generados si bien tienen una distribución "buena", esta está lejos de ser la óptima. Creemos que aprovechar otras arquitecturas, como D2GAN [5], puede llevar a una mejor generalización de los datos.

En segundo lugar, creemos que se puede mejorar el esquema de ataque, este actualmente es bastante deficiente y no aprovecha todo el poder que un sistema generativo le puede brindar. Creemos que este proceso podría ser unificado en el entrenamiento de la arquitectura GAN, logrando que todo el proceso sea mucho más eficiente y localizado.

Por último, en esta investigación solo se usó *explicit feedback* (datos de ratings de pelícuasl). Si bien la mayor parte de los métodos de *adversarial attackes* se basan en datos implícitos, hasta el momento no hay un métodos basados en GANs que se enfoquen en el problema, sería interesante ampliar el esquema al uso de este tipo de datos.

References

[1] Bo Li, Yining Wang, Aarti Singh, and Yevgeniy Vorobeychik. (2016). Data Poisoning Attacks on Factorization-Based Collaborative Filtering. *In NIPS.*

[2] Konstantina Christakopoulou and Arindam Banerjee (2019). Adversarial Atacks on an Oblivious Recommender. *In Thirteenth ACM Conference on Recommender Systems (RecSys '19)*

[3] Ian GoodFellow, Jean Pouget-Abadie, *et al*. (2014). Generative Adversarial Nets *In 27st Conference on Neural Information Processing Systems (NIPS)*

[4] Alec Radford, Luke Metz and Soumith Chintala. (2015). Unsupervised representa- tion learning with deep convolutional generative adversarial networks. *In ICLR*

[5] Nguyen, Tu Dinh. Le, Trung. Vu, Hung. Phung, Dinh.(2017). Dual Discriminator Generative Adversarial Nets. *In 31st Conference on Neural Information Processing Systems (NIPS 17)*

[6] Yashar Deldjoo, Tommaso di Noia and Felice Merra. (2020). Adversarial Machine Learning in Recommender Systems: State of the art and Challenges. *In 14th ACM Conference on Recommender Systems*