Facultad de Ingeniería
Tópicos Avanzados de la Analítica
Caso de Negocio – Segundo Semestre 2023
Carlos Andrés Másmela Pinilla
Daniela Isabel Zárate Bonilla
Yudy Tatiana Pedraza Torres
Adrián Ernesto Esquinas Álvarez

1. Business Understanding

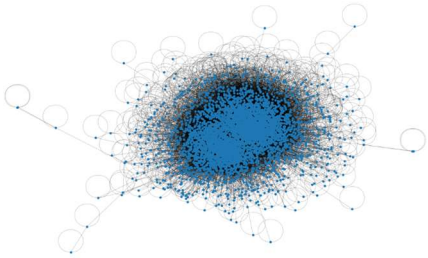
Twitch es una plataforma de streaming de video, en la que los creadores pueden realizar sus emisiones y cualquier persona registrada en la plataforma puede entrar a verlas y hasta interactuar con ellos. Esta centra principalmente en la transmisión de contenido relacionado con videojuegos, incluyendo transmisiones de videojuegos en vivo, discusiones sobre juegos, y otros tipos de contenido relacionado con el mundo de los videojuegos y el entretenimiento.

Como en esta plataforma se transmiten diferentes tipos de contenido y cada streamer tiene su estilo propio, se hace necesario considerar la incorporación de técnicas de análisis de grafos para mejorar la comprensión de las dinámicas entre estos streamers y los espectadores. Mediante la construcción y análisis de grafos se pueden identificar comunidades temáticas dentro de Twitch y detectar posibles focos de contenido inapropiado para ciertas audiencias, especialmente por el tipo de lenguaje usado.

Los datos utilizados en este artículo fueron obtenidos en [1], estos datos inicialmente fueron analizados en el siguiente papper Multi-scale Attributed Node Embedding de Cornell Univeristy en el apartado de Twitch [2], esta base de datos corresponde a nodos que representan usuarios de Twitch y los enlaces de amistades mutuas, la tarea asociada es la clasificación binaria de si un streamer utiliza lenguaje explícito o no.

2. Entendimiento de los datos

En primera instancia se realiza la visualización con Spring Layout con el fin de entender la estructura y las relaciones de la red social de manera más clara. En dicha visualización es notable que la mayoría de los nodos se encuentran en un cúmulo en el centro y pocos en la parte más externa de la red.



La red por analizar cuenta con 4.648 nodos, 64.030 conexiones y con todos los nodos conectados entre sí. En promedio un nodo está conectado a 27 nodos y para conectarnos de un nodo a cualquier otro se tendrían que atravesar 9 aristas o menos, teniendo en cuenta que en promedio se atraviesan aproximadamente 3 aristas para llegar de un nodo a otro.

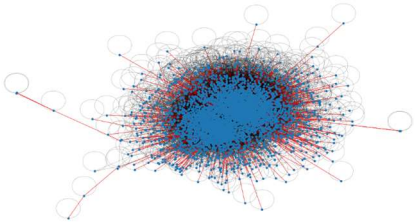
Por otro lado, el análisis de centralidad arrojó lo resultados que se muestran en la siguiente tabla. Los nodos con mayor grado de centralidad son los nodos que tienen mayor cantidad de conexiones, en Twitch hace referencia a los influencers de la red. El nodo con mayor grado de centralidad tiene conexión con alrededor del 22% de toda la red, siendo 1024 vecinos. Al ver la distribución de todos los usuarios, la mayoría tiene menos de 0,0125, eso quiere decir que muchos tienen centralidades extremadamente bajas y no están muy interconectados en la red. Solo determinados nodos como los observados en la tabla destacan por tener muchas conexiones.

La métrica de centralidad de intermediación nos dice que, aunque el nodo 4397 no sea parte de los 8 nodos de mayor grado de centralidad (de los más populares), tiene influencia entre los amigos destacados cuando trata de difundir información. La mayoría de los nodos no actúan como puentes en los caminos más cortos.

Grado de centralidad		Centralidad de intermediación		Centralidad de cercanía		Centralidad del vector propio	
Nodos	Grado	Nodos	Grado	Nodos	Grado	Nodos	Grado
1819	0,22	3719	0,11	3719	0,52	3719	0,15
3719	0,22	1819	0,11	1819	0,52	1819	0,15
2475	0,16	2475	0,05	1565	0,49	2475	0,12
596	0,13	1565	0,04	2475	0,49	982	0,12
1565	0,13	596	0,03	4397	0,48	1565	0,11
982	0,12	2864	0,03	982	0,47	4397	0,11
4142	0,12	982	0,03	724	0,47	596	0,11
2864	0,10	4397	0,03	596	0,47	724	0,11

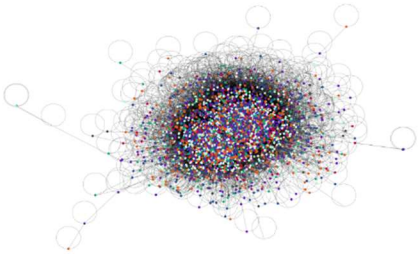
Adicionalmente, se exploraron las métricas de segregación. En primera instancia se calcula el coeficiente de agrupación promedio, el cual mide cuanto de los vecinos de un nodo están conectados entre sí y para evaluar la conexión entre los grupos, esta métrica dio 0,22 lo que significa que la red no cuenta con fuerte agrupaciones o comunidades de nodos. Lo anterior concuerda con el resultado de transitividad que cuantifica la presencia de grupos de nodos interconectados, l cual fue de 0,08, siendo un valor muy bajo e indicando que la red tiene muy pocas agrupaciones de nodos interconectados.

La red tiene 301 puentes y no hay puentes locales, lo que indica que no existen conexiones directas entre nodos que conecten diferentes agrupaciones o comunidades de nodos en la red y que los nodos dentro de cada grupo están más conectados entre sí que con nodos fuera de su grupo. En la siguiente visualización se pueden observar lo puentes en color rojo y los puentes locales en verde, que en realidad son inexistentes.



En las métricas de resiliencia, el coeficiente de asortatividad de -0,17 sugiere que la red es disasortativa. En una red disasortativa, los nodos tienden a conectarse con otros nodos que tienen características diferentes o contrastantes en lugar de características similares. En otras palabras, los nodos con ciertas propiedades, como los intereses, edades o ubicaciones, tienden a estar conectados a nodos que tienen propiedades diferentes.

Por último, para detectar comunidades se utilizó el algoritmo de comunidades fluidas asincrónicas, donde los nodos se distribuyen en 8 grupos que se pueden visualizar en el siguiente gráfico. La presencia de 8 comunidades distintas sugiere que la plataforma de Twitch es diversa y atrae a usuarios con una amplia variedad de intereses. Cada comunidad puede estar relacionada con un tipo específico de contenido o género de transmisiones, y los usuarios se agrupan en función de sus preferencias. En última instancia, el análisis de comunidades y el valor negativo de asortatividad reflejan la dinámica de una red diversa y conectada en Twitch.



3. Preparación de los datos

3.1 Matriz de adyacencia

Se identifica que la red de Twitch tiene 4.648 nodos, 64.030 conexiones y al representarlo en un grafo se observa que es de alta complejidad, mediante la representación de la matriz de adyacencia podemos identificar características adicionales de las conexiones para identificar comunidades o grupos que están interconectados o relacionados. Adicionalmente al representar el grafo con la matriz de adyacencia permite optimizar los resultados de los modelos predictivos mejorando las métricas.

3.2 Node2Vec

Node2Vec se basa en el concepto de embeddings, que son representaciones vectoriales densas de nodos en un grafo, que se basa en caminatas aleatorias y capturar las características de los nodos de un grafo y generar representaciones vectoriales densas. Para este caso en particular se definieron algunos parámetros de caminata para realizar el entrenamiento del Node2vec, dichos parámetros fueron: número de caminatas, distancia de la caminata, p y q.

```
import gensim
import networkx as nx
from gensim.models.word2vec import Word2Vec
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
import random

walks = []
num_walks = 1
walk_length = 10
p = 2.0
q = 0.5

for _ in range(num_walks):
    for node in G.nodes:
        walk = random_walk(node, walk_length, p, q)
        walks.append(walk)

# Create and train Word2Vec for DeepWalk
node2vec = Word2Vec(walks,
                    hs=1,   # Hierarchical softmax
                    sg=1,   # Skip-gram
                    vector_size=100,
                    window=10,
                    workers=2,
                    min_count=1,
                    seed=0)
node2vec.train(walks, total_examples=node2vec.corpus_count, epochs=30, report_delay=1)
```

4. Modelos

4.1 Sistema de recomendación

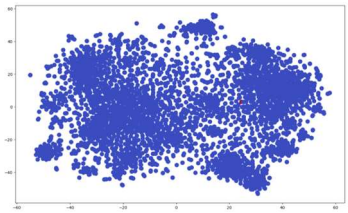
Se creó un sistema de recomendación que utiliza el algoritmo de Node2vec entrenado previamente para aprender las representaciones vectoriales del contenido del grafo de Twitch. Estas representaciones vectoriales fueron utilizadas para calcular la similitud entre los nodos, que contienen el detalle de los streamers, categorías de tipo de lenguaje utilizado, entre otras informaciones.

```
def recommend(node_id):
    recommendations = model.wv.most_similar(str(node_id))
    return recommendations
```

```
recommendations = recommend(1)
print(recommendations)
```

A este sistema de recomendación se le indica un nodo de interés y él devuelve los 10 nodos más similares al ingresado.

```
[('1492', 0.9299674034118652), ('357', 0.8683314323425293), ('3497', 0.8629679083824158), ('4235', 0.84214848279953), ('386', 0.8408998847007751), ('4172', 0.8324305415153503), ('753', 0.8137957453727722), (
```



4.2 Clasificación de los usuarios con lenguaje explícito

Para poder realizar la clasificación de los usuarios con lenguaje explícito se emplearon tres data sets: X que contemplaba el problema directamente, matriz de adyacencia y Node2Vec. Este fue el primer hiperparámetro a considerar para todos los métodos con los cuales se realiza la clasificación.

Adicionalmente, se aclara que para todos los algoritmos con los cuales se realizó esta clasificación se empleó el 80% de la data para entrenamiento, 10% para validación y 10% para test. Además, se aclara que al conjunto de test sólo se observó la combinación de hiperparámetros que generó mejor resultado en el conjunto de validación.

A partir de todo lo mencionado anteriormente, se procedió a realizar siete (7) modelos de Inteligencia Artificial con el fin de darle solución al problema en cuestión, los algoritmos empleados fueron los siguientes:

4.2.1. Regresión Logística

La regresión logística es un método estadístico utilizado para predecir la probabilidad de un evento binario, como sí/no o éxito/fracaso, en función de una o más variables independientes. Funciona aplicando una transformación logarítmica a la relación lineal entre las variables independientes y la variable dependiente, lo que resulta en una curva sigmoide que modela la probabilidad de pertenecer a una de las dos categorías. A través de un proceso de entrenamiento, la regresión logística ajusta los coeficientes de las variables independientes para maximizar la verosimilitud de los datos observados, lo que permite hacer predicciones sobre la probabilidad de pertenecer a una categoría específica.

Para este modelo se utilizaron los siguientes hiperparámetros, Penalty, C y max_iter. El primero, controla la regularización aplicada al modelo, itera en 'l1', 'l2', 'elasticnet' o 'none'; esta regularización es útil para prevenir el sobreajuste. El hiperparámetro C, hace que valores altos reduzcan la regularización y valores bajos aumentan la regularización. Y el hiperparámetro max_iter es el número máximo de iteraciones permitidas para la convergencia del algoritmo. En la siguiente tabla se observan los hiperparámetros utilizados.

Hiper-parámetros	Opciones del hiper-parámetro
Penalty	l1 , l2
C	0.01, 0.1, 0.2, 0.5, 1, 2
Max_iter	1000 , 2000

A continuación, se presenta la función que fue utilizada para realizar el modelo:

```
def reg_log(X_train, y_train, X_val, y_val) -> dict:
  # Loop para ajustar un modelo con cada combinación de hiperparámetros
  resultados_rl = {'params': [], 'Accuracy': [], 'F1': []}

  #Se entrena el modelo con x y y train
  #Se mira calidad del resultado con el set de validación
  for params in rl_param_grid:
        rl = LogisticRegression(
            random_state=42,
            solver= 'saga',
            **params)
        rl.fit(X_train, y_train)
        y_pred = rl.predict(X_val)
        resultados_rl['params'].append(params)
        resultados_rl['Accuracy'].append(accuracy_score(y_val, y_pred))
        resultados_rl['F1'].append(f1_score(y_val, y_pred))

        print(f"Modelo: {params} \u2713")
  return resultados_rl
```

Las mejores soluciones, en el conjunto de prueba, para cada dataset empleando todos los hiperparámetros mencionados anteriormente fueron las siguientes:

- X original del dataset: 0.7096
- Matriz de Adyacencia: 0.7376
- Node2Vec: 0.7204

4.2.2 Random Forest

Se realiza un modelo de Random Forest que en resumen es un conjunto de árboles de decisión que se entrena en muestras de datos aleatorias con reemplazo (bootstrapping) y utiliza el promedio de las predicciones de los árboles individuales para hacer predicciones. Esto hace que el modelo sea robusto y resistente al overfitting.

Se utilizan los siguientes hiperparámetros para ajustar y controlar la complejidad del modelo y su capacidad de generalización: n_estimators: para controlar el número de árboles en el bosque, max_depth: Definiendo la profundidad máxima de cada árbol en el bosque, min_samples_split: que establece el número mínimo de muestras requeridas para dividir un nodo interno en el árbol, min_samples_leaf: para definir el número mínimo de muestras requeridas en una hoja del árbol y por último max_features: que controla el número de características a considerar cuando se busca la mejor división en un nodo.

A continuación, se muestran los hiperparámetros usados:

Hiper-parámetros	Opciones del hiper-parámetro
n_estimators	50, 100, 200
max_depth	3, 5, None
min_samples_split	2, 5, 10
min_samples_leaf	1, 2, 4
max_features	Auto, sqrt, log2

A continuación, se muestra el modelo de Random Forest empleado para hacer la clasificación:

```
# Loop para ajustar un modelo con cada combinación de hiperparámetros

resultados_rf = {'params': [], 'Accuracy': [], 'F1': []}

# Se entrena el modelo con X_train e y_train
# Se mira la calidad del resultado con el set de validación
for params in rf_param_grid:
    rf = RandomForestClassifier(
        random_state=42,
        **params
    )
    rf.fit(X_train, y_train)
    y_pred = rf.predict(X_val)
    resultados_rf['params'].append(params)
    resultados_rf['Accuracy'].append(accuracy_score(y_val, y_pred))
    resultados_rf['F1'].append(f1_score(y_val, y_pred))

    print(f"Modelo: {params} \u2713")
```

Las mejores soluciones (accuracy), en el conjunto de prueba, para cada dataset empleando todos los hiperparámetros mencionados anteriormente fueron las siguientes:

- X original del dataset: 0.7225
- Matriz de Adyacencia: 0.7548
- Node2Vec: 0.7096

4.2.3 SVM

El tercer algoritmo ejecutado para dar solución al problema propuesto fue el Support Vector Machine, este algoritmo de aprendizaje supervisado utilizado principalmente para tareas de regresión y clasificación. Se busca encontrar un hiperplano que mejor separe dos clases de datos.

En este problema se hizo una búsqueda de hiperparámetros a través de un GridSearch, se consideraron los siguientes hiperparámetros: Kernel (función matemática usada para permitir que los datos sean separables linealmente en un espacio de mayor dimensionalidad), C (Termino de regularización, para controlar el equilibrio entre el margen y la clasificación correcta de las clases) y Gamma (afecta la capacidad del modelo para ajustar los datos de entrenamiento y generalizar los datos no vistos). A continuación, se muestran los hiperparámetros probados:

Hiper-parámetros	Opciones del hiper-parámetro
Kernel	Linear, RBF
C	0.5, 0.7
Gamma	1.3, 1.5

A continuación, se muestra el código del modelo SVM que fue empleado para realizar esta clasificación:

```
def support_vector_machine(X_train, y_train, X_val, y_val) -> dict:
  # Loop para ajustar un modelo con cada combinación de hiperparámetros
  resultados_svm = {'params': [], 'Accuracy': [], 'F1': []}

  #Se entrena el modelo con x y y train
  #Se mira calidad del resultado con el set de validación
  for params in svm_grid:
        svm = SVC(
            random_state= 42,
            ** params
            )
        svm.fit(X_train, y_train)
        y_pred = svm.predict(X_val)
        resultados_svm['params'].append(params)
        resultados_svm['Accuracy'].append(accuracy_score(y_val, y_pred))
        resultados_svm['F1'].append(f1_score(y_val, y_pred))

        print(f"Modelo: {params} \u2713")

  return resultados_svm
```

Las mejores soluciones, en el conjunto de prueba, para cada dataset empleando todos los hiperparámetros mencionados anteriormente fueron las siguientes:

- X original del dataset: 0.7096
- Matriz de Adyacencia: 0.7096
- Node2Vec: 0.7096

4.2.4 XGBoost

El cuarto algoritmo empleado para solucionar este problema fue el XGBoost, este contempla una mejora al algoritmo de árboles aplicando bagging a los mismos, el funcionamiento de este algoritmo se basa en ir aprendiendo de los errores de los modelos y al final unir la solución generando la menor cantidad de errores posibles.

En el problema en particular se empleó la búsqueda de hiperparámetros mediante Grid Search de los siguientes argumentos: tasa de aprendizaje (empleada para ver a que tasa va aprendiendo el algoritmo de sus errores pasados), cantidad de número de árboles (se emplea con el fin de ver qué tanto generaliza a mayor o menor cantidad de árboles), cantidad de variables (se emplea con el fin de reducción del ρ para poder observar la mejor distribución de variables para optimizar el algoritmo), máxima profundidad del árbol (con el fin de restringir hasta donde puede un árbol realizar diferenciación por categorías) y muestra de columnas por árbol (es un submuestreo de las columnas por cada nivel). A continuación, se presentan los hiperparámetros buscados:

Hiper-parámetros	Opciones del hiper-parámetro
Learning Rate	0.01, 0.1, 0.2
Profundidad	3, 5, None
Árboles	50, 100, 200
Cantidad de Variables	0.7, 1.0
Muestra por árbol	0.5, 0.7, 1.0

A continuación, se presenta el código del modelo XGBoost que fue empleado para realizar esta clasificación.

```python
# Loop para ajustar un modelo con cada combinación de hiperparámetros

resultados_xgb = {'params': [], 'Accuracy': [], 'F1': []}

#Se entrena el modelo con x y y train
#Se mira calidad del resultado con el set de validación
for params in xgb_param_grid:
      xgb = XGBClassifier(
          eval_metric='mlogloss',
          random_state= 42,
          ** params
          )
      xgb.fit(X_train, y_train)
      y_pred = xgb.predict(X_val)
      resultados_xgb['params'].append(params)
      resultados_xgb['Accuracy'].append(accuracy_score(y_val, y_pred))
      resultados_xgb['F1'].append(f1_score(y_val, y_pred))

      print(f"Modelo: {params} \u2713")
```

Las mejores soluciones (accuracy), en el conjunto de prueba, para cada dataset empleando todos los hiperparametros mencionados anteriormente fueron las siguientes:

- X original del dataset: 0.7118
- Matriz de Adyacencia: 0.7333
- Node2Vec: 0.7096

4.2.5 GNN

Se realiza un modelo de GNN que es una red neuronal y se utiliza para el aprendizaje en grafos y datos estructurados. Es útil en aplicaciones donde los datos tienen una estructura de grafo como en el caso de Twitch. Las capas GNN permiten que la red aprenda representaciones en el contexto de las relaciones y conexiones en el grafo.

En el problema en particular se empleó la búsqueda de hiperparámetros mediante Grid Search de los siguientes argumentos: activacion: para determinar la función de activación que se utiliza en las capas de la red neuronal. Las opciones son "relu" y "tanh", dropout: El valor de dropout especifica la probabilidad de que se desactive una unidad (neurona) en una capa durante el entrenamiento. Un valor de 0.4, 0.5 o 0.6 significa que, en promedio, el 40%, 50% o 60% de las neuronas se apagarán aleatoriamente durante cada iteración de entrenamiento y por último el número de capas, donde cada elemento de la lista es una capa en la red y está representado como una lista que contiene dos valores: el número de neuronas en la capa y el número de capas GNN relacionadas con esa capa.

A continuación, se presenta los hiperparámetros empleados:

Hiper-parámetros	Opciones del hiper-parámetro
Capas	[32, 0], [32, 16]
Activación	Relu, tanh
Dropout	0.4, 0.5, 0.6

A continuación, se muestra la clase GNN con la cual se entrenó el algoritmo durante 100 épocas.

```python
class MLP(torch.nn.Module):
    """Multilayer Perceptron"""
    def __init__(self, dim_in, dim_h, dim_h_2=0, dim_out=1):
        super().__init__()
        self.linear1 = Linear(dim_in, dim_h)
        if dim_h_2 > 0:
          self.linear2 = Linear(dim_h, dim_h_2)
          self.linear3 = Linear(dim_h_2, dim_out)
          self.capas = 3
        else:
          self.linear2 = Linear(dim_h, dim_out)
          self.capas = 2

    def forward(self, x, activacion, dropout):

        if activacion == "relu":
          x = self.linear1(x)
          x = torch.relu(x)
          x = F.dropout(x, p=dropout, training=self.training)
          x = self.linear2(x)
          if self.capas > 2:
            x = torch.relu(x)
            x = F.dropout(x, p=dropout, training=self.training)
            x = self.linear3(x)
          return F.log_softmax(x, dim=1)
```

Después de ejecutar el algoritmo, las mejores soluciones, en el conjunto de prueba, para cada dataset empleando todos los hiperparámetros mencionados anteriormente fueron las siguientes:

- X original del dataset: 0.7225
- Matriz de Adyacencia: 0.7311
- Node2Vec: 0.7268

4.2.6 GCN

El modelo Graph Convolutional Network (GCN) son redes neuronales convolucionales (CNN) aplicadas a grafos, adicionalmente se menciona que este tipo de GNN son las más populares en grafos. Las GCN no toman en consideración los diferentes vecinos que tiene cada nodo, es decir, suman sin coeficientes de normalización.

La búsqueda de hiperparámetros que se realizó para poder optimizar este algoritmo fueron: el número de capas (que muestra qué tan profunda puede llegar a ser la red) activación (empleada para que el modelo pueda aprender labores complejas) y dropout (método de regularización que apaga neuronas con el fin de evitar overfitting). A continuación, se presenta los hiperparámetros empleados:

Hiper-parámetros	Opciones del hiper-parámetro
Capas	3, 4
Activación	Relu, tanh
Dropout	0.4, 0.5, 0.6

A continuación, se muestra la clase GCN con la cual se entrenó el algoritmo durante 100 épocas.

```
class GCN(torch.nn.Module):
    """Graph Convolutional Network"""
    def __init__(self, dim_in, dim_h, dim_out, capas_conv):
      super().__init__()
      if capas_conv == 3:
        self.gcn1 = GCNConv(dim_in, dim_h*4)
        self.gcn2 = GCNConv(dim_h*4, dim_h*2)
        self.gcn3 = GCNConv(dim_h*2, dim_h)
        self.linear = torch.nn.Linear(dim_h, dim_out)
      elif capas_conv == 4:
        self.gcn1 = GCNConv(dim_in, dim_h*8)
        self.gcn2 = GCNConv(dim_h*8, dim_h*4)
        self.gcn3 = GCNConv(dim_h*4, dim_h*2)
        self.gcn4 = GCNConv(dim_h*2, dim_h)
        self.linear = torch.nn.Linear(dim_h, dim_out)
```

```
def forward(self, x, edge_index, capas_conv, activacion, dropout):
  if capas_conv == 3:
    if activacion == "relu":
      h = self.gcn1(x, edge_index)
      h = torch.relu(h)
      h = F.dropout(h, p=dropout, training=self.training)
      h = self.gcn2(h, edge_index)
      h = torch.relu(h)
      h = F.dropout(h, p=dropout, training=self.training)
      h = self.gcn3(h, edge_index)
      h = torch.relu(h)
      h = self.linear(h)
      return h
    if activacion == "tanh":
```

Del anterior código se puede explicar que dependiendo la cantidad de capas, depende la cantidad de neuronas de salida que se tendrán en cada capa, se puede apreciar que la cantidad de neuronas crece en exponencial de 2. Adicionalmente se muestra que dependiendo de los hiperparámetros mencionados con anterioridad el forward se activa de distintas maneras. Después de ejecutar el algoritmo, las mejores soluciones, en el conjunto de prueba, para cada dataset empleando todos los hiperparámetros mencionados anteriormente fueron las siguientes:

- X original del dataset: 0.7419
- Matriz de Adyacencia: 0.7462
- Node2Vec: 0.7505

4.2.7 Modelos de Atención

El último modelo empleado para poder dar solución al ejercicio en cuestión fue la red de gráficos de atención (GAT por sus siglas en inglés) donde el principal propósito de este modelo es reconocer qué nodos son más

importantes que otros y para ello los pesos de las capas que produce este modelo también consideran la importancia de las características del nodo.

La búsqueda de hiperparámetros que se realizó para poder optimizar este algoritmo fueron: el número de capas (que muestra qué tan profunda puede llegar a ser la red) activación (empleada para que el modelo pueda aprender labores complejas) y dropout (método de regularización que apaga neuronas con el fin de evitar overfitting). A continuación, se presenta los hiperparámetros empleados:

Hiper-parámetros	Opciones del hiper-parámetro
Capas	2, 3, 4
Activación	Elu, relu
Dropout	0.4, 0.5, 0.6

A continuación, se muestra la clase GAT con la cual se entrenó el algoritmo durante 100 épocas.

```
class GAT(torch.nn.Module):
  def __init__(self, dim_in, dim_h, dim_out, capas_, heads=16):
    super().__init__()
    if capas_ == 2:
      self.gat1 = GATv2Conv(dim_in, dim_h, heads=int(heads/2))
      self.gat2 = GATv2Conv(dim_h*int(heads/2), dim_out, heads=1)
    elif capas_ == 3:
      self.gat1 = GATv2Conv(dim_in, dim_h, heads=heads)
      self.gat2 = GATv2Conv(dim_h*heads, dim_h, heads=int(heads/2))
      self.gat3 = GATv2Conv(dim_h*int(heads/2), dim_out, heads=1)
    elif capas_ == 4:
      self.gat1 = GATv2Conv(dim_in, dim_h, heads=heads)
      self.gat2 = GATv2Conv(dim_h*heads, dim_h, heads=int(heads/2))
      self.gat3 = GATv2Conv(dim_h*int(heads/2), dim_h, heads=int(heads/4))
      self.gat4 = GATv2Conv(dim_h*int(heads/4), dim_out, heads=1)
```

```
def forward(self, x, edge_index, capas_, activacion, dropout):
  if capas_ == 2:
    if activacion == 'elu':
      h = F.dropout(x, p=dropout, training=self.training)
      h = self.gat1(h, edge_index)
      h = F.elu(h)
      h = F.dropout(h, p=dropout, training=self.training)
      h = self.gat2(h, edge_index)
      return F.log_softmax(h, dim=1)
    elif activacion == 'relu':
```

Del anterior código se puede explicar que dependiendo la cantidad de capas, depende la cantidad de cabezas en el constructor de clase. Adicionalmente se muestra que dependiendo de los hiperparametros mencionados con anterioridad el forward se activa de distintas maneras. Después de ejecutar el algoritmo, las mejores soluciones, en el conjunto de prueba, para cada dataset empleando todos los hiperparametros mencionados anteriormente fueron las siguientes:

- X original del dataset: 0.7376
- Matriz de Adyacencia: 0.7440
- Node2Vec: 0.7376

5. Evaluación

Posterior a entrenar todos los modelos con todos los hiperparametros y datasets, y posteriormente validar su solución en el conjunto de validación, se realizó un ordenamiento de la data a partir del "Accuracy" generado en el conjunto de test, donde se logró encontrar que el mejor algoritmo fue el Random Forest con un valor de 0.7548 y una combinación de hiperparámetros de la siguiente manera: dataset de matriz de adyacencia, n_estimator de 200, no posee máxima profundidad, min_sample_split de 10, min_sample_leaf de 1 y max_feature igual a auto. A continuación, se presenta tabla con resumen de las mejores corridas de cada algoritmo.

Algoritmo	Accuracy
Random Forest	0.7548
GCN	0.7505
GAT	0.7440

Regresión Logística	0.7376
XGBoost	0.7333
GNN	0.7311
SVM	0.7096

6. Recomendaciones de negocio y conclusiones

Basándonos en los resultados de los diferentes modelos desarrollados se generan las siguientes conclusiones y recomendaciones de negocio:

El modelo de Random Forest y GCN han demostrado el mejor rendimiento en términos de precisión para la clasificación cuando un streamer utiliza lenguaje explícito o no. Por lo tanto, se recomienda utilizar el modelo Random Forest como el principal clasificador, ya que tiene la precisión más alta.

El uso de la matriz de adyacencia en los algoritmos de clasificación entrego el mejor resultado en la mayoría de los casos, incluso superando al Node2vec. De la misma forma, el uso de X original del dataset generó los peores resultados en la totalidad de los casos.

En cuanto a la Identificación de Contenido Inapropiado, dado que Twitch es una plataforma con un enfoque en videojuegos y entretenimiento, es importante mantener un ambiente seguro y apropiado para los espectadores. El uso de modelos de clasificación de lenguaje explícito ayuda a identificar y filtrar contenido inapropiado para ciertas audiencias.

El Análisis de Grafos incorpora técnicas de análisis y es fundamental para comprender las dinámicas entre los streamers y los espectadores. Identificar comunidades temáticas dentro de Twitch puede ayudar a dirigir contenido específico a grupos de audiencia interesados y a mejorar la interacción entre streamers y espectadores. Por lo anterior, la creación del sistema de recomendación genera gran utilidad, al recomendar los 10 streamers con mayor similitud al streamer de interés.

Dentro de otras conclusiones para el negocio en el futuro y dado que Twitch es una plataforma en constante evolución con una gran cantidad de contenido generado diariamente, se recomienda implementar un sistema de monitoreo continuo para detectar y abordar cualquier cambio en el comportamiento de los streamers y la calidad del contenido.

Como conclusión final, los resultados de los modelos desarrollados proporcionan una base sólida para mejorar la calidad y la seguridad en la plataforma Twitch. La implementación de modelos de ML, en particular Random Forest, puede contribuir a la identificación de contenido inapropiado y al análisis de la dinámica de la comunidad, lo que a su vez puede respaldar estrategias comerciales y de seguridad.

Referencias

[1] «Pytorch Geometric,» [En línea]. Available: https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html.

[2] BENEDEK ROZEMBERCZKI, CARL ALLEN, RIK SARKAR, «Multi-scale Attributed Node Embedding,» 23 March 2021. [En línea]. Available: https://arxiv.org/abs/1909.13021.