



UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE CIENCIAS EXACTAS Y NATURALES

Caracterización de trayectorias educativas a partir de producciones de código

Tesis de Licenciatura en Ciencias de Datos

Carla de Erausquin

Director: Matías López y Rosenfeld

Codirector: Pablo Turjanski

Buenos Aires, 2024

CARACTERIZACIÓN DE TRAYECTORIAS EDUCATIVAS A PARTIR DE PRODUCCIONES DE CÓDIGO

Con el creciente interés en la Ciencia y la Tecnología en los últimos años, los docentes enfrentan el desafío de corregir grandes volúmenes de código sin perder de vista el nivel de conocimiento alcanzado por los diferentes grupos de estudiantes. En este contexto, contar con herramientas que permitan predecir de manera temprana el riesgo de abandono en cursos introductorios de programación se vuelve crucial para enfocar la atención en aquellos alumnos que más lo necesitan.

Para ello se analizó un conjunto de datos correspondiente a un curso de Introducción a la Programación en Python. Se estudiaron exclusivamente sus producciones de código, transformadas a *embeddings* con ayuda del modelo *CodeBERT*. Utilizando técnicas de reducción de la dimensionalidad y, posteriormente, *clustering* se logró caracterizar a parte de los alumnos que abandonarían el curso de manera temprana. En particular, al grupo de estudiantes que abandonarían el curso una vez llegada la primera instancia de evaluación. No se logró diferenciar, para los ejercicios propuestos, entre aquellos alumnos que finalizarían el curso y aquellos que prosiguieron con las entregas más allá del primer examen.

Palabras clave: Clustering, CodeBERT, Embeddings, Enseñanza, Python, Programación.

CARACTERIZACIÓN DE TRAYECTORIAS EDUCATIVAS A PARTIR DE PRODUCCIONES DE CÓDIGO

With the growing interest in Science and Technology in recent years, teachers face the challenge of correcting large volumes of code while keeping track of the knowledge level achieved by different groups of students. In this context, having tools that allow early prediction of dropout risk in introductory programming courses becomes crucial to focus attention on those students who need it most.

For this purpose, a dataset corresponding to an Introduction to Python Programming course was analyzed. Only their code productions were studied, transformed into embeddings with the help of the *CodeBERT* model. Using dimensionality reduction techniques and, subsequently, clustering, it was possible to characterize some of the students who would drop out of the course early. In particular, the group of students who would abandon the course upon reaching the first evaluation instance. For the proposed exercises, it was not possible to differentiate between those students who would complete the course and those who continued with submissions beyond the first exam.

Keywords: Clustering, CodeBERT, Embeddings, Teaching, Python, Programming.

AGRADECIMIENTOS

A Pablo y Matias! (Placeholder)

Una dedicatoria

Índice general

1. Introducción	1
1.1 Motivación	1
1.2 Modelos de lenguaje y el ambito estudiantil	2
1.3 CodeBert	2
1.4 Propuesta de la tesis	3
2. Materiales y métodos	4
2.1 Fuentes de datos	4
2.1.1 Proceso ETL de los datos	4
2.1.1.1 Extracción	5
2.1.1.2 Transformación	5
2.1.1.3 Carga	8
2.2 Etapas de trabajo para el Análisis de los Datos	10
2.2.1 Reducción de la dimensionalidad	10
2.2.2 Extracción de información adicional	11
2.2.3 Agrupamiento	12
2.2.3.1 Kmeans	13
2.2.3.2 Elección del número de <i>clusters</i>	13
2.2.3.3 Fuzzy-c-means	14
2.2.3.4 Evaluación	14
2.2.4 Consideraciones sobre múltiples entregas	16
3. Resultados y discusión	17
3.1 Análisis individual de los ejercicios	17
3.2 Evaluación del agrupamiento de manera global	19
3.2.1 Métricas: ARI y NMI	20
3.2.2 Métricas: Índice de Jaccard	20
3.3 Comparación por clusters en común	21

3.3.1	Clique máximo	22
3.3.2	Separación por grupos	25
3.4	Nuevos enfoques para mejorar los resultados	26
3.4.1	Eliminación del código repetido	26
3.4.2	Inserción de datos de interés	28
3.4.3	Usando <i>embeddings</i> normalizados	28
3.4.4	Fuzzy-2-means	29
3.5	Representando a los alumnos como un único vector	30
3.5.1	Vectores de ceros	30
3.5.2	Función vacía	31
3.5.3	Análisis de la proporción de entregas	33
4.	Conclusión	35
5.	Trabajo futuro	37

1. INTRODUCCIÓN

1.1. Motivación

La programación, de manera directa e indirecta, forma parte de la vida cotidiana, y cada día aumenta su influencia. Su estudio constituye un aspecto central para la innovación en los procesos de enseñanza y aprendizaje en contextos de educación formal e informal. Además de ser una herramienta clave para el futuro, también es de vital importancia para la construcción de ciudadanía en un mundo atravesado por la tecnología.

Esto fue reflejado por el Consejo Federal de Educación cuando definió que se lleven a cabo las acciones necesarias para que se incorpore la enseñanza de la programación en la escolaridad obligatoria *con el fin de fortalecer el desarrollo económico y social de nuestro país* [1].

En particular, como se puede observar en la Figura 1.1, dentro de la Facultad de Ciencias Exactas y Naturales (FCEN) de la Universidad de Buenos Aires (UBA), la cantidad de alumnos que se inscriben a carreras afines a la programación, como lo es la Carrera en Ciencias de Datos, está en constante crecimiento.



Fig. 1.1: Cantidad de inscriptos por carrera en la FCEN - UBA [2]

En 2021, primera cohorte de inscripciones a Ciencias de Datos, alrededor de 300 personas se anotaron para comenzar sus estudios. En 2023, se anotaron más de 450 personas

a dicha carrera, a las cuales les correspondería cursar Introducción a la Programación en su primer año de carrera [3]. Es entonces con este incremento en el alumnado que nace la necesidad de poder corregir la producción de código que generan los alumnos de manera rápida y eficiente.

En un análisis exploratorio realizado por alumnos de la facultad basado en Censos de la FCEN-UBA, Encuestas Docentes y el Sistema de Inscripciones, se observó que hay indicios de que la deserción está relacionada con el rendimiento del alumno en estas primeras materias [4]. Por ello, consideramos importante la temprana detección de grupos más proclives a abandonar los cursos en cuestión.

1.2. Modelos de lenguaje y el ámbito estudiantil

En los últimos años se observaron muchos esfuerzos en desarrollar herramientas para la enseñanza de la programación que pudieran capturar las características individuales para potenciar el aprendizaje. Algunas de estas líneas han buscado usar la noción de similitud como cantidad de cambios a realizar en un código para proponer corregir errores en ejercicios de manera automática [7].

Gracias a los avances tecnológicos previamente mencionados, es posible trabajar con *embeddings* de manera *más sencilla* y poder proyectar texto en un espacio multidimensional. Una aplicación fue capturar semejanzas entre entregas de estudiantes y, organizándolas en orden por similitud, facilitar la tarea de corrección [8]. Además, este tipo de análisis sirve para identificar el conocimiento adquirido en temas específicos de un curso [9], así como la evolución a lo largo del mismo [10]. Esto abre la oportunidad para poder planificar intervenciones docentes en momentos tempranos que potencien el aprendizaje, que en cursos masivos antes resultaba imposible.

En su investigación usando *code2vec*, Brigante evaluó si el código fuente en sí mismo, sin ejecutar el código o evaluar su salida, era suficiente para estimar la calidad de las respuestas de los estudiantes. Los resultados sugieren que, aunque no siempre se alcanzaron los resultados esperados, el análisis del código fuente ofrece algunos indicios útiles para evaluar la calidad del desarrollo del estudiante [9].

1.3. CodeBert

Se trata de un modelo preentrenado diseñado para entender y generar tanto lenguaje natural (NL) como lenguajes de programación (PL). Se basa en una arquitectura *Transform-*

mer, similar a *BERT* y *RoBERTa*, y fue entrenado usando datos bimodales (pares NL-PL) y unimodales (código o texto sin parejas), en seis lenguajes de programación (Python, Java, JavaScript, PHP, Ruby y Go) usando datos de repositorios públicos en GitHub. Esto último es particularmente relevante siendo que los cursos mencionados en la (FCEN-UBA), son dictados en el lenguaje Python.

CodeBERT genera *embeddings* o representaciones vectoriales de fragmentos de código mediante su arquitectura basada en *Transformers*. Al alimentar un fragmento de código como entrada, se convierte el código en una secuencia de *tokens*, que se luego procesa a través de capas de redes neuronales para producir una representación contextualizada para cada *token*. Estas representaciones logran capturar tanto la sintaxis como la semántica del programa.

El *embedding* final para un fragmento completo puede obtenerse utilizando el vector asociado al *token* especial [CLS], que representa el significado agregado de la secuencia. Estos *embeddings* pueden emplearse para tareas como búsqueda de código, clasificación, o generación de documentación, ya que contienen información relevante tanto del contenido del código como de su contexto.

CodeBERT demostró un rendimiento sobresaliente en tareas de búsqueda de código y generación de documentación en comparación con modelos preentrenados solo en lenguaje natural o código [5]. Esto, sumado al público y al fácil acceso del modelo [6], hicieron que se optara por él para el análisis de este trabajo.

1.4. Propuesta de la tesis

Utilizando el modelo *CodeBERT*, proponemos caracterizar las trayectorias académica de los estudiantes de un curso introductorio de programación, basándonos exclusivamente en sus producciones de código. Esto puede ser particularmente útil para los docentes, ya que podría permitir identificar a los grupos más vulnerables en la cursada y saber en cuáles concentrar los esfuerzos.

2. MATERIALES Y MÉTODOS

2.1. Fuentes de datos

Los datos empleados para este análisis son el resultado de entregas de alumnos de la materia Programación en Python de la Universidad Nacional de San Martín (UNSAM), dictada de manera virtual en contexto de pandemia. La materia contó con más de 1,200 preinscriptos, incluyendo un 40 % de personas que residen fuera del Área Metropolitana de Buenos Aires (AMBA).

Esta materia se dictó con el objeto de *enseñar los fundamentos del lenguaje Python orientado al manejo de datos, a la escritura de scripts y a una organización adecuada de los programas; enseñar algunos rudimentos de la teoría de algoritmos, incluyendo conceptos básicos de la teoría de la complejidad y algunas estructuras de datos no triviales; e introducir la programación orientada a objetos* [11].

Con un total de 12 unidades y una duración de 3 meses, se realizó una evaluación permanente con entregas semanales de entre cuatro y seis ejercicios obligatorios por semana; llegando en total a 56 ejercicios seleccionados a lo largo de todo el curso. Además de evaluar estas entregas semanales, se tomaron dos exámenes parciales de forma virtual y sincrónica con modalidad de opción múltiple lo que permitió una corrección automatizada. Si bien contamos con los datos sobre los resultados de la evaluación de este curso, no fueron considerados ya que se realizó foco en predecir quienes sencillamente lo habían finalizado. Para aprobar la materia resultó necesario haber realizado adecuadamente el 70 % de los ejercicios obligatorios y haber aprobado ambos exámenes parciales [11].

El enfoque pedagógico de este curso, tanto en sus contenidos como en su metodología de enseñanza, tiene estrecha similitud con las materias introductorias que se dictan en nuestra facultad, particularmente con la cátedra de Introducción a la Programación. Esto nos permite inferir que los resultados de este análisis podrían ser extrapolables al contexto más amplio de la UBA.

2.1.1. Proceso ETL de los datos

Para desarrollar nuestra propia fuente de datos a partir de otras fuente, se utilizó el proceso ETL (Extract/Estracción, Transform/Transformación y/and Load/Carga) (ETL).

A continuación se detalla cada una de estas tres etapas.

2.1.1.1 Extracción

Los ejercicios fueron entregados a través de *google-forms*. Las ejercitaciones de interés se descargaron de un *google-drive* de uno de los docentes de la materia. De las 12 unidades que constituyeron al curso, para este análisis se tuvieron en cuenta únicamente las primeras 5, como se observa en la figura 2.1. Al finalizar la unidad 4 fue la presentación del primer examen. Siendo deseable la caracterización temprana de aquellos estudiantes que abandonarán el programa y con intenciones de que esto pueda replicarse en cursos futuros, se consideró estudiar únicamente la primera mitad del curso.

En la figura 2.1 se pueden observar los contenidos de la materia a analizar.

Unidad	Contenido
Introducción a Python	El intérprete interactivo de Python; Crear, editar y ejecutar pequeños programas;
	Algunos tipos de datos de Python: número enteros, números de punto flotante, cadenas y listas.
Funciones y estructuras	Archivos; Funciones; Tipos y estructuras de datos; Contenedores.
Trabajando con datos	Secuencias; Contadores; Formatos de impresión; Uso de un entorno de desarrollo integrado; Caso de estudio: Arbolado porteño.
Crear y operar con listas	Errores y excepciones; Listas; Comprensión de listas; El problema de la búsqueda en una lista; La búsqueda lineal; Todo es un objeto; Caso de estudio: Arbolado porteño.
Aleatoriedad	El debugger de Python; El módulo random; El módulo numpy; Caso de estudio: Álbum de Figuritas; Algo sobre gráficos en Python.

Fig. 2.1: Contenidos evaluados en las unidades extraídas que fueron utilizadas como fuente de datos para el caso de estudio [11]

2.1.1.2 Transformación

A los códigos extraídos de los 31 ejercicios comprendidos por estas unidades para los más de 800 alumnos de los cuales los obtuvimos, se los anonimizó. Para ello asociamos cada entrega con un identificador creado por nosotros, *id_anon*.

Se obtuvieron entonces en una primera instancia tres conjuntos de datos:

- **Anonimizador** que describe por entrada, para cada identificador, el identificador propio del ejercicio en cuestión, el nombre del archivo y la fecha y hora de entrega. Esto último es particularmente útil siendo que los alumnos podían realizar más de

una entrega del mismo ejercicio.

- **Código** archivos *.py* con los ejercicios con el mismo nombre que en la tabla anterior. Los nombres eran identificadores únicos de los ejercicios lo que permitía vincularlos con el Anonimizador.
- **Resultados** de la cursada, con el identificador del estudiante, la nota de las entregas, los dos parciales y el trabajo práctico y un valor *booleano* haciendo referencia a si este aprobó o no la cursada. Dentro de este conjunto se encuentran únicamente aquellos alumnos que finalizaron la materia.

Webster et al. definen *token* como una unidad básica en el procesamiento del lenguaje natural (NLP) que no necesita ser descompuesta en etapas posteriores del procesamiento. Puede ser un elemento como una palabra, un modismo o una expresión fija, que se trata como un átomo indivisible para los propósitos de la computación. La tokenización implica identificar estas unidades básicas en el texto, que son esenciales para realizar análisis o generación de texto posteriores [12].

El modelo de interés, *CodeBERT*, tiene un límite de 512 *tokens* para la entrada. Esto significa que cualquier secuencia de código a procesar tiene que ser truncada o dividida para que no exceda esta cantidad. Es debido a esta limitación del modelo que se tomó la decisión de realizar una limpieza del código, eliminando así los comentarios del mismo. Además, por cada ejercicio a entregar se contaba con una función *objetivo*, que fue la que se consideró para la extracción. Cabe destacar que de realizarse testeos automatizados, aquellos archivos que no contaran con dichas funciones *objetivo* (o estuvieran mal nomencladas), habrían fallado. Esto se consideró como un criterio adicional para discriminar entre qué datos incorporaríamos al análisis subsiguiente y cuáles no.

```

1 import copy
2
3 # Función para invertir una lista
4 def invertir_lista(lista):
5     invertida = [] # Inicializa una lista vacía para almacenar los elementos invertidos
6     i = 0 # Contador para recorrer la lista original
7     for e in lista: # Itera sobre cada elemento en la Lista original
8         # Agrega el elemento desde el final de la lista original hacia el principio
9         invertida = [invertida + [lista[([len(lista) - 1] - 1)]]]
10        i += 1 # Incrementa el contador
11    return invertida # Retorna la lista invertida
12
13 # Función para propagar valores en una lista
14 def propagacion(lista):
15     propagacion = False # Bandera para indicar si la propagación está activa
16     i = 0 # Contador para recorrer la lista
17     for x in lista: # Itera sobre cada elemento en la lista
18         if not propagacion: # Si La propagación no está activa
19             if x == 1: # Si el elemento es 1
20                 propagacion = True # Activa la propagación
21             if propagacion: # Si La propagación está activa
22                 if x == 0: # Si el elemento es 0
23                     lista[i] = 1 # Cambia el 0 a 1 para propagar el efecto
24                 if x == -1: # Si el elemento es -1
25                     propagacion = False # Detiene la propagación
26             i += 1 # Incrementa el contador
27     return lista # Retorna la lista modificada
28
29 # Función para propagar valores en la lista en ambas direcciones
30 def propagar(lista):
31     b = copy.deepcopy(lista) # Hace una copia profunda de la lista original
32     list_partel = propagacion(b) # Propaga los valores en la lista original
33     list_inver = invertir_lista(list_partel) # Invierte la lista resultante
34     list_parte2 = propagacion(list_inver) # Propaga los valores en la lista invertida
35     list_inver2 = invertir_lista(list_parte2) # Invierte nuevamente la lista
36     return list_inver2 # Retorna la lista final

```

```

1 def propagar(lista):
2     b = copy.deepcopy(lista)
3     list_partel = propagacion(b)
4     list_inver = invertir_lista(list_partel)
5     list_parte2 = propagacion(list_inver)
6     list_inver2 = invertir_lista(list_parte2)
7     return list_inver2

```

Fig. 2.2: Código entregado por un estudiante como resolución del ejercicio *propagar* antes y después de su limpieza. A modo de ejemplo, utilizando *nlkt.tokenize* la cantidad de *tokens* del código de la izquierda es de 337 y el de la derecha, 38.

Mencionan Glassman et al. en su trabajo *OverCode: visualizing variation in student solutions to programming problems at scale* [13] en relación a la limpieza de comentarios de código a analizar que “...la variación en los comentarios es tan grande que agruparlos y resumirlos requerirá un diseño adicional significativo”. Este es otro motivo para eliminar el texto adicional de las producciones de alumnos para la limpieza realizada, como se observa en el ejemplo de la Figura 2.2.

Para el código en sí, se extrajo el *Abstract Syntax Trees (AST)*. Un árbol de sintaxis abstracta es una representación estructurada de un programa, que retiene la estructura esencial del árbol de análisis sintáctico pero elimina los nodos innecesarios. Mantiene la precedencia y el significado de las expresiones, simplificando la representación al omitir la mayoría de los nodos de los símbolos no terminales [14]. Esto permitió un rápido filtrado de las llamadas funciones *objetivo* y la eliminación de los comentarios de manera mucho más eficiente que utilizando expresiones regulares. Para realizar el proceso mencionado anteriormente se utilizó la biblioteca *ast* [15] en Python.

Código a embeddings

Como se mencionó anteriormente, se usó el modelo *CodeBERT* para pasar del código a los *embeddings*. Desde la biblioteca *transformers* [16] es posible importar tanto el *tokenizador* como el modelo en sí a partir del código mostrado en el Listado 2.1.

Listing 2.1: Importación de CodeBERT

```
from transformers import RobertaTokenizer, RobertaModel
tokenizer = RobertaTokenizer.from_pretrained("microsoft/
```

```
codebert-base")
model = RobertaModel.from_pretrained("microsoft/codebert-base")
```

Luego, para cada archivo `.py`, una vez limpiado como se mencionó en secciones anteriores, bastó con realizar el procedimiento descrito en el Algoritmo 1 para obtener sus *embeddings* a partir de *CodeBERT*.

Algorithm 1 Generación de embeddings a partir de código fuente

```

1: Inicializar lista vacía embeddings
2: for cada código en la lista de códigos do
3:   Intentar:
4:     Tokenizar el código
5:     Crear secuencia de tokens:
6:       [CLS] + tokens_código + [SEP] + [EOS]
7:       // CLS: token de inicio de secuencia
8:       // SEP: token separador
9:       // EOS: token de fin de secuencia
10:    Convertir tokens a IDs
11:    Calcular embedding usando el modelo
12:    Añadir {código, embedding} a embeddings
13:   Si hay error:
14:     Imprimir mensaje de error
15: end for
```

Además, se almacenaron los *embeddings* normalizados. La normalización de *embeddings* ajusta los vectores para que tengan una magnitud uniforme, comúnmente de longitud 1. Esto facilita comparaciones y cálculos posteriores al eliminar diferencias en escalas. Para ello usamos la biblioteca *normalize* de *scikit-learn* [17].

2.1.1.3 Carga

Para almacenar los tres conjuntos de datos previamente mencionados, se optó por un tipo de base No-SQL, MongoDB que se creó localmente. Los motivos son varios:

- **Flexibilidad en el esquema:** MongoDB, al ser una base de datos NoSQL, permite trabajar con un esquema flexible. Esto es especialmente útil en este trabajo, ya que los datos presentan diferentes tipos y estructuras que pueden evolucionar con el tiempo. La flexibilidad de MongoDB permite la adaptación del modelo de datos sin necesidad de realizar cambios complejos en el esquema.
- **Facilidad de uso:** La sintaxis de MongoDB es sencilla e intuitiva, lo que simplifica su uso y permite concentrarse en el análisis y manipulación de los datos en lugar de

en la configuración del entorno.

- **Documentación abundante:** MongoDB cuenta con una amplia documentación lo que facilitó mucho el trabajo con esta herramienta.
- **Manejo eficiente de grandes volúmenes de datos:** Dado que MongoDB almacena los datos en formato BSON (una representación binaria de JSON), es capaz de manejar grandes volúmenes de datos de manera eficiente.
- **Capacidad de escalabilidad horizontal:** MongoDB permite la escalabilidad horizontal mediante la adición de nodos al clúster de base de datos. Esto es una ventaja importante si se prevé un crecimiento en el volumen de datos o en la demanda de las consultas. A sabiendas de que en un futuro podría agregarse información adicional por parte de los análisis posteriores, esto resulta particularmente útil [18].
- **Integración con herramientas de análisis y desarrollo:** MongoDB se integra fácilmente con una amplia variedad de herramientas de análisis de datos, lenguajes de programación y bibliotecas, lo que facilita el desarrollo de aplicaciones y análisis de datos en entornos como Python. En particular, a la hora de implementar un conector para la base de datos creada localmente, se usó la biblioteca *pymongo* [19].

Luego, La base final obtenida finalmente, luego de aplicar el proceso de ETL, se organizó con la estructura descrita en la Figura 2.3.

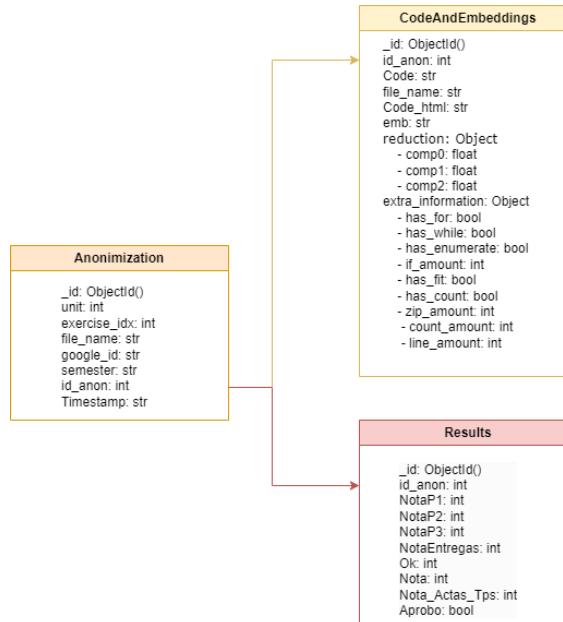


Fig. 2.3: Esquema del diseño de la base obtenida luego de aplicar el proceso ETL..

Donde gran parte de los campos contenidos en la base de datos de la Figura 2.3 fueron ya explicados en secciones anteriores. En cuanto a la colección **CodeAndEmbeddings**,

se ahondará más adelante.

La base de datos para guardar todos los datos pertinentes a este estudio fue *hosteada* localmente. Esta decisión resultó conveniente por varias razones: primero, permitió un acceso rápido y eficiente a los datos; segundo, facilitó el control total sobre la seguridad y la integridad de la información; y tercero, eliminó la dependencia de servicios externos.

Además, el *hosting* local de la base de datos proporcionó flexibilidad para realizar consultas complejas y actualizaciones en tiempo real, lo que fue crucial para el análisis iterativo de los datos durante el desarrollo del proyecto.

2.2. Etapas de trabajo para el Análisis de los Datos

2.2.1. Reducción de la dimensionalidad

El análisis de componentes principales (PCA) es una técnica de reducción de dimensionalidad que transforma datos de alta dimensión en un espacio de menor dimensión, conservando la mayor parte de la varianza original. Para *embeddings* como los de *CodeBERT*, es posible usar PCA para reducir la dimensionalidad de los vectores, preservando la información relevante y facilitando su visualización y análisis [20].

PCA ofrece ventajas claras para *embeddings*. Permite reducir los vectores a dos o tres dimensiones, permitiendo la detección de patrones en los datos [20]. También ayuda a eliminar ruido y redundancias, mejorando la calidad del análisis posterior. Sin embargo, como señala Basirat [21], tiene limitaciones con datos que no se ajustan a una distribución normal.

Otra desventaja es el costo computacional de calcular la matriz de covarianza, especialmente para grandes volúmenes de datos. Esto puede ser un obstáculo en proyectos extensos de, por ejemplo, procesamiento del lenguaje natural [21]. Sin embargo, esta sigue siendo una herramienta valiosa para la reducción de dimensionalidad y la optimización en el manejo de *embeddings*, motivo por el cual se usó para los análisis posteriores.

Teniendo en cuenta las ventajas y limitaciones mencionadas anteriormente, se usó la técnica de PCA para reducir la dimensionalidad de los *embeddings*, los cuales en su estado original poseen 768 dimensiones

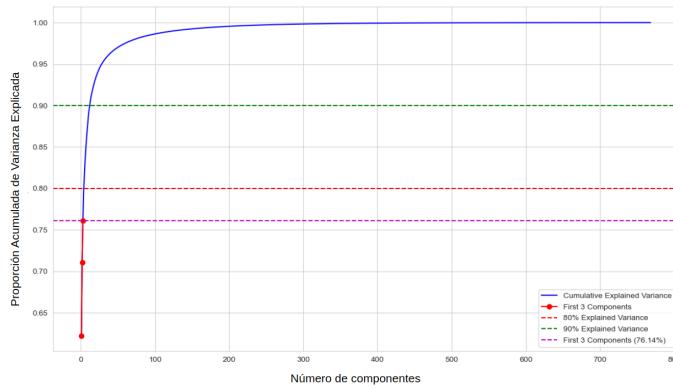


Fig. 2.4: Varianza explicada en base a la aplicación de PCA a los *embeddings*.

Luego de aplicar la técnica de PCA, la varianza explicada observada en la Figura 2.4 se la comparó con la de trabajos similares, como el de *user2-code2vec: Embeddings for Profiling Students Based on Distributional Representations of Source Code* [22] en el que proponen una metodología para perfilar a estudiantes individuales de ciencias de la computación según su diseño de programación, utilizando *embeddings*. Azcona et al. mencionan que, hablando de los *embeddings* con los que estaban trabajando, *Los vectores se transforman a 2 dimensiones utilizando PCA. La varianza retenida es muy baja (entre 2 % y 6 %)*. Cabe destacar igualmente que, si bien a raíz de lo anterior no se esperaba una varianza explicada muy alta, en el trabajo de Azcona et al. el tamaño de los vectores generados es casi veinte veces mayor que el de los generados por *CodeBERT*.

2.2.2. Extracción de información adicional

Sabemos que *CodeBERT* captura la conexión semántica entre el lenguaje natural y el lenguaje de programación [5]. En un análisis preliminar se investigó qué factores podían llegar a generar *embeddings* diferentes para códigos semánticamente similares o idénticos. Con esto se esperaba encontrar características en el código sobre las cuales hacer foco para mejorar las futuras predicciones, de ser necesario.

Para ello, se tomó una producción de código de un alumno al azar del ejercicio de la primer unidad, *tiene-a*, el cual debería retornar *True* de contener el *string* de entrada la letra *a*. A este se le hicieron pequeñas modificaciones, como, por ejemplo:

1. Modificación del nombre de las variables.
2. Cambio del *while* por un *for*.
3. Inversión en el recorrido de la secuencia (en vez de principio a fin, de fin a principio).

4. Inclusión de prints.

5. Simplificación del código.

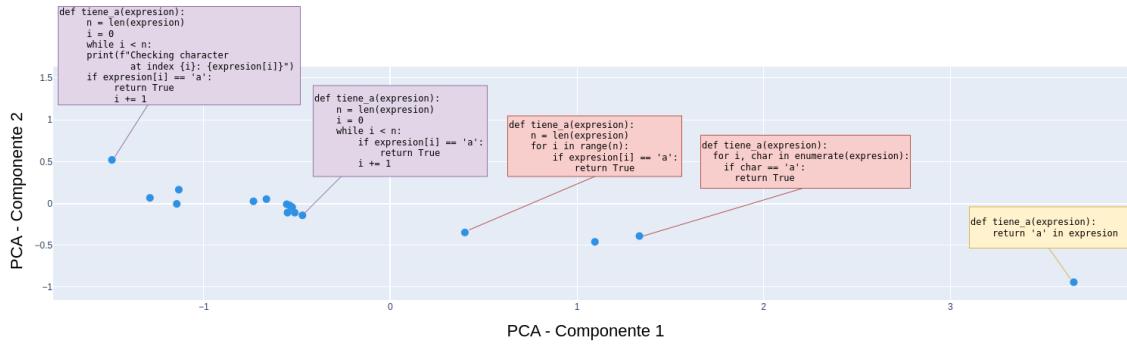


Fig. 2.5: Representación de los dos primeros componentes de PCA sobre los *embeddings* de diferentes variantes del ejercicio *tiene_a*, donde el código original es el segundo de izquierda a derecha. El pequeño grupo formado alrededor de este corresponde a variaciones muy leves en el código, como inversión en el recorrido de la lista o cambio en el nombre de variables.

Como era de esperarse, como muestra la Figura 2.5, se observó que variantes muy similares del código (como, por ejemplo las propuestas por los ítems 1) y 3)) no se distanciaban mucho las unas de las otras. En la figura se puede observar también una clara tendencia en cuanto a la longitud del código y su representación en el plano. Además, diferencias como el uso de *for* (los tres puntos comprendidos entre las dos cajas rojas) y el *while* (aquellos comprendidos entre las cajas violetas) o la ausencia de ambos, tuvieron también un impacto significativo en los *embeddings* y su representación en el plano de dos dimensiones.

A partir de lo anterior, siendo que utilizar *for vs while* puede cambiar la posición que ocupa el punto en el gráfico 2.5, entonces se decidió incorporar el uso o no de estas primitivas a la tabla (ver Figura 2.3, tabla CodeAndEmbeddings). No sólo hemos incorporado el dato de si se ha utilizado *for/while* sino que también hemos incorporado el dato de la cantidad de *if* que se ha utilizado, la cantidad de líneas que posee el código fuente, entre otros datos.

2.2.3. Agrupamiento

La próxima sección abordará las técnicas de *clustering* implementadas para agrupar estudiantes, basándose en los *embeddings* y la reducción de dimensionalidad mediante la técnica de PCA mediante PCA.

2.2.3.1 Kmeans

El algoritmo *K-means* es una técnica de aprendizaje no supervisado ampliamente utilizada para resolver problemas de agrupamiento. Funciona dividiendo un conjunto de datos en k *clusters* ó grupos distintos.

Primero, el algoritmo selecciona aleatoriamente k centroides iniciales, que representan los centros de los *clusters*. Luego, asigna a cada punto de datos al *cluster* cuyo centro esté más cercano, basado en la distancia euclídea. Una vez asignados todos los puntos, se recalculan los centroides como la media de los puntos en cada *cluster*. Este proceso de asignación y actualización de centroides se repite iterativamente hasta que las posiciones de los centroides ya no cambien significativamente, indicando que el algoritmo convergió. [23]

Sin embargo, el algoritmo *K-means* presenta muchos desafíos que afectan negativamente su rendimiento. En primer lugar, en el proceso de inicialización del algoritmo uno debe especificar *a priori* la cantidad de grupos en un conjunto de datos dado (el número k), mientras que los centros de los grupos iniciales se seleccionan aleatoriamente. Además, el rendimiento del algoritmo es susceptible a la selección de este grupo inicial y, para conjuntos de datos grandes, determinar el número óptimo de grupos con el que comenzar se vuelve desafiante. Más aún, la selección aleatoria de los centros de los grupos iniciales a veces resulta en una convergencia local mínima debido a que se trata de un algoritmo *greedy*. [24].

Este algoritmo y otras variantes que serán mencionadas a continuación se utilizarán para intentar discriminar entre qué alumnos finalizarán o no el curso.

2.2.3.2 Elección del número de *clusters*

Teniendo en cuenta las limitaciones mencionadas anteriormente, se eligió el **método del codo** para elegir el número k a la hora de realizar los *clusterings* que detallaremos en secciones posteriores. Esta es una técnica visual que ayuda a optimizar el número de grupos a utilizar en *K-means*. Su objetivo principal es ayudar a tomar una decisión informada sobre cuántos *clusters* elegir.

El método funciona de la siguiente manera:

1. Se ejecuta el algoritmo de *clustering* varias veces con diferentes números de *clusters*.
2. Para cada k , se calcula la suma de los errores cuadráticos dentro del *cluster* (*WCSS*, por sus siglas en inglés).
3. Se grafica el *WCSS* en función del número de *clusters*.

4. Se busca un “codo” en el gráfico, que es un punto donde la disminución del WCSS comienza a nivelarse.

El “codo” en el gráfico representa el punto donde agregar más clústeres no mejora significativamente la calidad del agrupamiento. Este punto se considera el número óptimo de clústeres. Para ello, se calcula el error cuadrático total (SSE, por sus siglas en inglés) para diferentes cantidades de *clusters*. El SSE mide la suma de las distancias cuadradas de cada punto al centroide más cercano, y se grafica frente al número de *clusters*. En el gráfico, inicialmente el SSE disminuye rápidamente con el aumento de *clusters*, pero en cierto punto la disminución se ralentiza, formando un “codo” en la curva. Este punto de inflexión indica el número óptimo de *clusters*.

El método es beneficioso porque optimiza la elección del número de *clusters* eliminando la necesidad de depender de suposiciones previas [25].

2.2.3.3 Fuzzy-c-means

El algoritmo *Fuzzy-c-means* es una técnica de agrupamiento que optimiza una función objetivo. Esta función minimiza la distancia entre los puntos de datos y los centros de los *clusters*, ponderada por los valores de pertenencia difusa. El algoritmo itera entre actualizar los valores de pertenencia y los centros de los *clusters* hasta alcanzar la convergencia.

Entre las ventajas del *Fuzzy-c-means* se encuentra su capacidad para permitir la pertenencia parcial a múltiples *clusters*, lo que lo hace más flexible que los métodos de agrupamiento tradicionales. Además, maneja bien *clusters* de formas no esféricas y es más robusto ante *outliers* en comparación con el algoritmo *K-means* que mencionamos anteriormente.

Sin embargo, *Fuzzy-c-means*, al igual que *K-means*, también requiere que se especifique el número de grupos *a priori*. En consecuencia, también es sensible a la inicialización, lo que puede afectar los resultados finales. Además, tiene un mayor costo computacional en comparación con *K-means*.

Chan et al. en “Clustering of clusters” introducen una variante llamada *Recursive Fuzzy-2-Mean* (RF2M), que extiende el *Fuzzy-c-means* para agrupar *clusters* existentes. Esta variante introduce una tercera clase para los puntos que no encajan bien en ninguno de los dos *clusters* principales, añadiendo así más flexibilidad al método original [26].

2.2.3.4 Evaluación

Adjusted Rand Index

El *Adjusted Rand Index* (ARI) es una métrica utilizada para determinar la similitud entre dos resultados de agrupamiento (*clustering*). La fórmula del ARI es:

$$\text{ARI} = \frac{\text{RI} - \text{esperadoRI}}{\max \text{RI} - \text{esperadoRI}} \quad (2.1)$$

donde RI representa el índice de Rand, que calcula la similitud entre dos resultados de *clustering* al considerar todos los puntos identificados dentro del mismo clúster. El valor del ARI es igual a 0 cuando los puntos se asignan aleatoriamente a los clústeres y es igual a 1 cuando los dos resultados de *clustering* son idénticos. Esta métrica es útil para evaluar si los resultados de agrupamientos reducidos dimensionalmente son similares entre sí [27].

Normalized Mutual Information

El **NMI** (Normalized Mutual Information) es una métrica utilizada para evaluar la calidad de las particiones en la detección de comunidades en redes. Esta métrica mide la similitud entre la partición obtenida por un algoritmo de detección de comunidades y una partición de referencia (también llamada *ground truth*). El NMI está normalizado para variar entre 0 y 1, donde 1 indica una coincidencia perfecta entre las particiones y 0 indica que no hay relación alguna entre ellas.

La fórmula para calcular el NMI es la siguiente:

$$\text{NMI} = \frac{2 \cdot I(X; Y)}{H(X) + H(Y)}$$

donde $I(X; Y)$ representa la información mutua entre las particiones X y Y , y $H(X)$ y $H(Y)$ son las entropías de X e Y , respectivamente. Esta métrica es utilizada para comparar las divisiones obtenidas por diferentes métodos y verificar qué tan similares son a la partición de referencia, proporcionando una medida cuantitativa de la precisión en la detección de comunidades [28].

Similitud de Jaccard

El índice de similitud de Jaccard es una medida utilizada para calcular la similitud entre dos conjuntos. Dados dos conjuntos A y B , el índice de Jaccard se define como el tamaño de la intersección dividido por el tamaño de la unión de los conjuntos. Formalmente, se expresa como:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

donde:

- $|A \cap B|$ representa el número de elementos que son comunes a ambos conjuntos A y B .
- $|A \cup B|$ representa el número total de elementos únicos presentes en al menos uno de los conjuntos.

Una forma alternativa de expresar esta fórmula es:

$$J(A, B) = \frac{m_{11}}{m_{10} + m_{01} + m_{11}}$$

donde:

- m_{11} es el número de elementos presentes en ambos conjuntos.
- m_{10} es el número de elementos presentes en A pero no en B .
- m_{01} es el número de elementos presentes en B pero no en A .

El índice de Jaccard es útil para medir la similitud entre conjuntos en diversas aplicaciones, como la minería de datos y la comparación de características en grandes espacios de atributos. La métrica ignora las no-ocurrencias compartidas entre los conjuntos, ya que estas no proporcionan información relevante sobre su similitud [29].

2.2.4. Consideraciones sobre múltiples entregas

Finalmente, mencionamos que los estudiantes tenían la posibilidad de entregar un ejercicio las veces que quisieran. Para simplificar el análisis, si bien se cargaron a la base todas las entregas de las unidades mencionadas, se consideraron únicamente las entregas finales de cada alumno. Este filtrado se realizó rápidamente ya que contábamos con el *Timestamp* de cada archivo.

3. RESULTADOS Y DISCUSIÓN

3.1. Análisis individual de los ejercicios

En una primera instancia, se evaluó para cada uno de los ejercicios entregables la distancia euclídea entre los puntos generados después de realizar la mencionada reducción de la dimensionalidad para los alumnos en cada una de las 31 funciones entregadas. Con esto se esperaba ver grupos bien definidos y una correlación en cada instancia de evaluación entre ellos y los grupos de interés (esto es, estudiantes que terminaron *vs* los que no).

Para ello, se graficaron varios *heatmaps* esperando ver en ellos distintos grupos de alumnos.

Se observó que, en ejercicios que podrían considerarse más sencillos como, por ejemplo, el ejercicio *tirar* de la unidad 5 que consistía en obtener el resultado de tirar un dado equilibrado de seis caras n veces, o *crear_album* de la misma unidad que consistía en crear un “álbum” (un vector de ceros) de n figuritas, habían grupos claramente definidos, como se observa en la Figura 3.1. Dada la simpleza de los ejercicios en cuestión, no se encontraron grandes variaciones de código en general para este tipo de entregas.

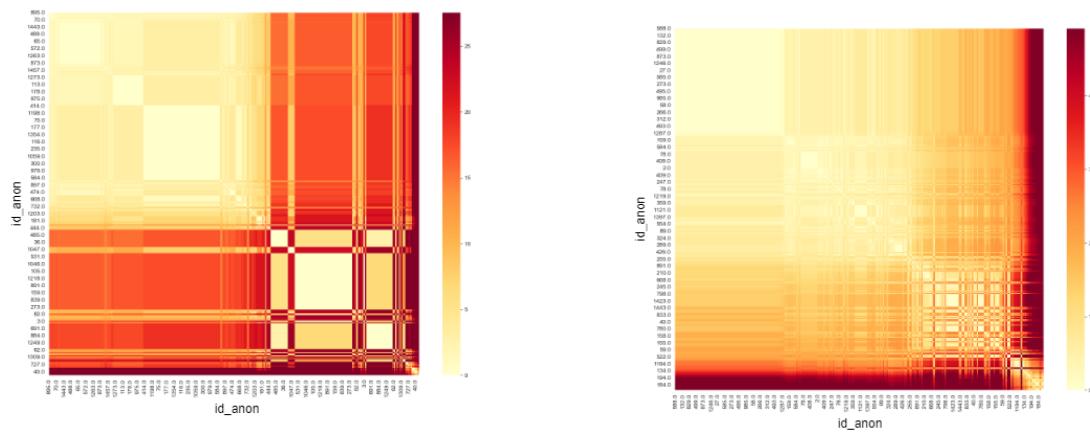


Fig. 3.1: Ejemplos de ejercicios con grupos marcados, analizando las distancias entre las entregas de cada alumno con cada alumno

En otros ejemplos, como los de la Figura 3.2, en los cuales los ejercicios podían llegar

a presentar una mayor dificultad o ambigüedad para los alumnos, los grupos no fueron tan claros. Este es el caso, por ejemplo, de los ejercicios *es_generala* de la unidad cinco y *propagar* de la unidad cuatro. Para la primera, el comportamiento esperado para la función era que retornara *True* si y sólo si los cinco datos de la lista tirada eran iguales. Para el segundo mencionado, los estudiantes debían escribir una función que recibiera una lista de fósforos (0: nuevo, 1: encendido, -1: carbonizado) y devolviera la lista con el fuego propagado a los fósforos nuevos adyacentes.

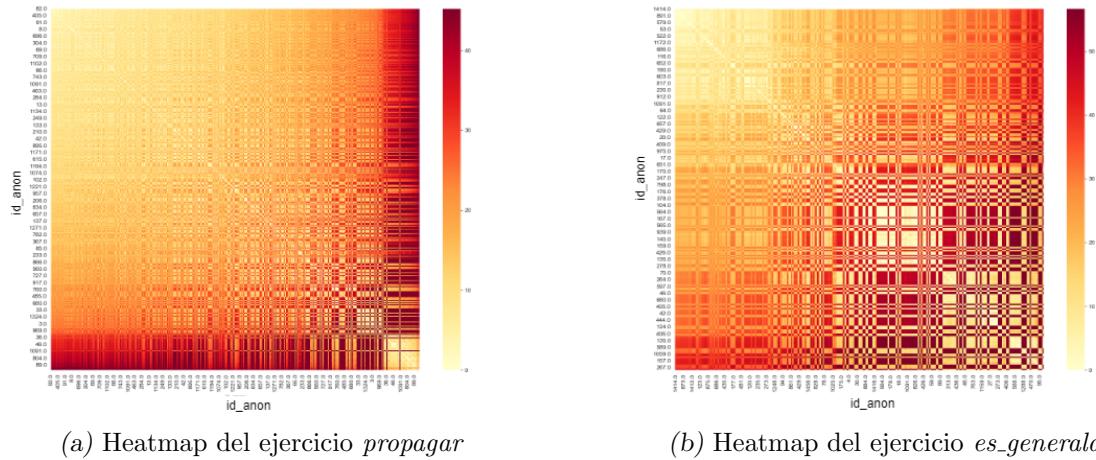


Fig. 3.2: Ejemplos de ejercicios con grupos no marcados, analizando las distancias entre las entregas de cada alumno con cada alumno

Cabe destacar que, a excepción de otros ejercicios típicamente introductorios en cursos como estos (como *sumar*, *máximo*, *mínimo* dentro de otros), los gráficos como los que muestran las figuras anteriores fueron los predominantes, dando poca información.

Se decidió entonces hacer foco en aquellos ejercicios en los que podían llegar a haber grupos mejor definidos. En un análisis posterior se exploró si estos grupos separaban de alguna manera a aquellos estudiantes que habían finalizado el curso de los que no. Para ello, se añadió como información adicional a las Figuras anteriores, quiénes habían finalizado el curso (*id_anon*) y quiénes no, como muestra la Figura 3.3.

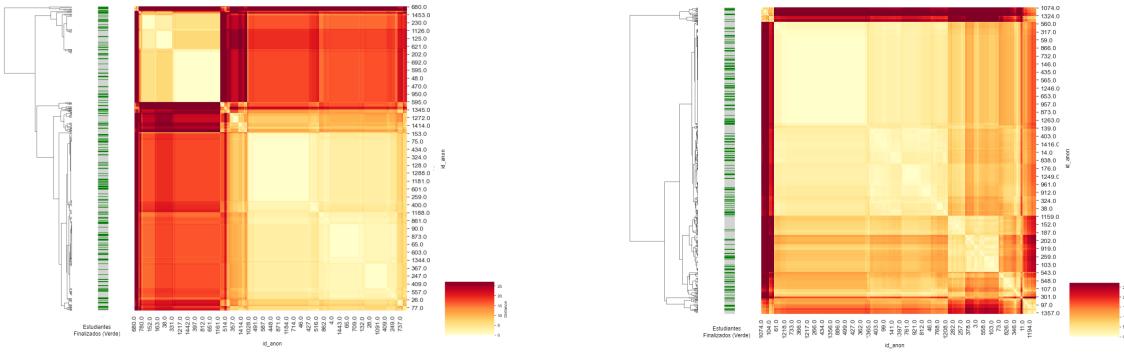


Fig. 3.3: Ejemplos de ejercicios con grupos marcados, analizando las distancias entre las entregas de cada alumno con cada alumno con dendograma e indicador de qué estudiante finalizó el curso y cuál no.

Se observó que, como muestra la Figura 3.3 , con los grupos obtenidos, no era posible distinguir entre los dos grupos deseados analizando los ejercicios de manera individual. Esto se ve evidenciado en que los alumnos que abandonaron el curso se encuentran entremezclados con los que sí lo hicieron. Esta tendencia (o falta de ella) se vio en todos los *heatmaps* que se realizaron. Tampoco fue posible extraer (al menos todavía) cuál fue el criterio por el cual se agruparon esos alumnos.

En un intento adicional por obtener información a partir de los ejercicios observados de manera atómica, se realizaron dos *heatmaps* por ejercicio, ahora separando manualmente ambos grupos esperando ver sub-grupos diferenciados. Pero, como era de esperarse basándonos en el resultado anterior, la distribución de ambos grupos era similar, obteniéndose figuras muy similares a pesar de estar forzando esta distinción.

3.2. Evaluación del agrupamiento de manera global

Para esta sección, se analizaron varias métricas para evaluar el *clustering* realizado. La evaluación rigurosa de los resultados del agrupamiento es fundamental por múltiples razones: permite validar la calidad de las agrupaciones obtenidas, asegura que los patrones descubiertos son significativos y no aleatorios, y facilita la comparación objetiva entre diferentes configuraciones del algoritmo. Además, dado que el *clustering* es una técnica de aprendizaje no supervisado, estas métricas proporcionan una forma cuantitativa de verificar si los grupos identificados representan verdaderamente estructuras naturales en los datos.

3.2.1. Métricas: ARI y NMI

Como se mencionó anteriormente, para cada ejercicio se realizó una *clusterización* usando el método del codo. En el análisis de las agrupaciones generadas, se evaluó el grado de correspondencia entre las particiones obtenidas para cada ejercicio con las de los demás ejercicios. Para esto, se utilizaron dos métricas principales: el *Indice de Información Mutua Normalizado* (NMI) [28] y el *Indice Rand Ajustado* (ARI) [27]. Los resultados de estas comparaciones pueden observarse en la Figura 3.4, donde los valores de NMI y ARI para cada ejercicio se representan en relación con los demás.

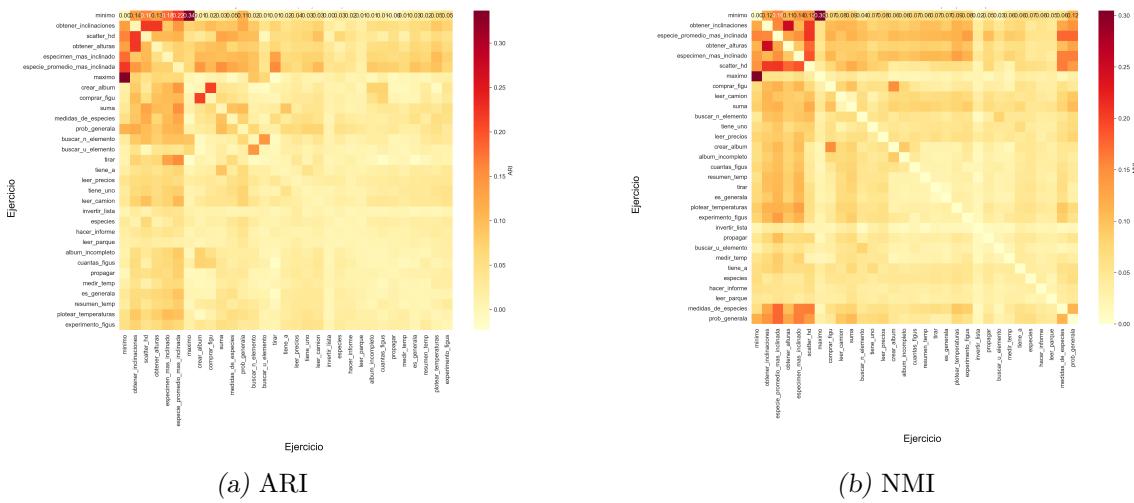


Fig. 3.4: Heatmaps del ARI y NMI de todas las particiones generadas para cada ejercicio para evaluar su grado de correspondencia con los demás ejercicios.

Los bajos valores de NMI, observables en la Figura 3.4, sugieren que la información compartida entre las particiones detectadas y la referencia es mínima. Esto significa que las divisiones obtenidas por el algoritmo no están alineadas con las categorías del otro ejercicio con las que se las está comparando, lo que indica una posible falta de coherencia en la identificación de las comunidades o grupos en los datos.

Por otro lado, los bajos valores de ARI implican que la similitud entre las asignaciones de elementos a *clusters* es baja en comparación con la asignación de referencia (o, como es este caso, a la partición generada para otro ejercicio).

3.2.2. Métricas: Índice de Jaccard

Para esta evaluación se analizaron las particiones de manera individual. Esto es, obviando a qué ejercicio pertenecían, se calculó el Índice de Similitud de Jaccard [29] para

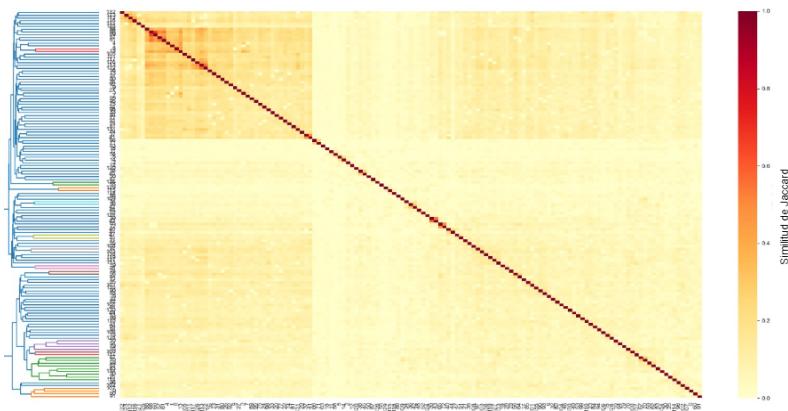


Fig. 3.5: Índice de similitud de Jaccard entre todos los *clusters* generados ejercicio a ejercicio.

cada uno de los más de cien *clusters* generados a lo largo de los 31 entregables.

Mencionan Loginova et al. sobre la caracterización de dos grupos de alumnos (*low* y *high achievers*) a través de *embeddings* usando PCA y *Kmeans* que ...*la mayoría de los clusters no son fácilmente interpretables, ya que contienen una mezcla de ambas clases, siguiendo aproximadamente la distribución de las etiquetas objetivo* [30].

Cabe destacar que es razonable que entre *clusters* de un mismo ejercicio, como se puede ver en la Figura 3.5, el Índice de Jaccard sea bajo. Esto puede ser un indicador de una buena separación entre los *clusters*. Sin embargo, hubiera sido deseable que entre distintos ejercicios el Índice de Jaccard fuera alto, indicando que estas particiones se sostienen a lo largo del curso e, idealmente, un indicador de a qué grupo (ya sea deserción o finalización) del curso pertenecen los estudiantes.

3.3. Comparación por clusters en común

Alineados con la idea de encontrar un patrón a lo largo de todos los ejercicios a analizar, se observaron ahora los *clusters* en común para cada estudiante. Esto es, en vez de calcular *cluster* a *cluster* qué estudiantes tenían en común, se indagó sobre, estudiante a estudiante, qué *clusters* tenían en común.

Para analizar la similitud en las agrupaciones de estudiantes a lo largo de todo el curso, se generó entonces un *heatmap* que muestra la cantidad de *clusters* compartidos entre cada par de ejercicios. En este gráfico, cada celda representa el número de agrupaciones comunes entre los estudiantes, lo que permite visualizar patrones de consistencia en las asignaciones de *clusters* a lo largo de los distintos ejercicios.

Este enfoque facilita identificar si ciertos grupos de estudiantes tienden a mantenerse unidos en las diferentes particiones generadas o si las agrupaciones varían significativamente entre ejercicios.

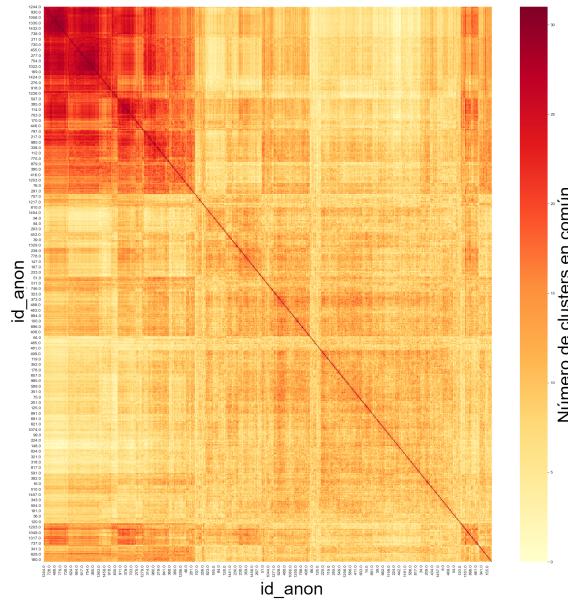


Fig. 3.6: Cantidad de *clusters* compartidos por estudiante.

Se puede observar en la Figura 3.6 un grupo que comparte una gran cantidad de *clusters* en la esquina superior izquierda. Se buscó una manera de caracterizar este grupo, a partir de la matriz de *clusters* compartidos generada para la realización del *heatmap*.

3.3.1. Clique máximo

Usando la biblioteca Networkx [31], se buscó para diferentes cantidades de *clusters* compartidos, el *clique* más grande con el siguiente código:

Algorithm 2 Encontrar el clique más grande con n clusters compartidos

```

1: function ENCONTRARCLIQUEMASGRANDE( $n$ , matriz_de_clusters_compartidos)
2:    $G \leftarrow$  Crear grafo vacío
3:   id_anons  $\leftarrow$  índices de matriz_de_clusters_compartidos
4:   Añadir nodos id_anons a  $G$ 
5:   for cada par  $(i, j)$  de nodos en  $G$  do
6:     if matriz_de_clusters_compartidos[ $i, j$ ]  $\geq n$  then
7:       Añadir arista  $(i, j)$  a  $G$ 
8:     end if
9:   end for
10:  cliques  $\leftarrow$  Encontrar todos los cliques máximos en  $G$ 
11:  clique_mas_grande  $\leftarrow$  Clique más grande en cliques
12:  return clique_mas_grande
13: end function

```

El código garantiza que todos los estudiantes en el *clique* resultante comparten al menos n *clusters* entre sí debido a dos aspectos clave de su implementación. Primero, en la construcción del grafo, solo se crea una arista entre dos estudiantes si comparten n o más *clusters*. Esto significa que la existencia de una conexión en el grafo es equivalente a compartir al menos n *clusters*.

Segundo, el algoritmo busca el *clique* más grande en este grafo. Por definición, un *clique* es un subgrafo completamente conectado, lo que significa que todos los nodos en el *clique* están conectados directamente entre sí. Como en este grafo cada conexión representa compartir al menos n *clusters*, todos los estudiantes en el *clique* resultante necesariamente comparten al menos n *clusters* con todos los demás miembros del *clique*. Así, la combinación de la construcción selectiva del grafo y la búsqueda de *cliques* asegura la propiedad deseada.

En cuanto a la ejecución del algoritmo anterior, basados en los datos con los que se generó la Figura 3.6 se comenzó a iterar buscando el *clique* más grande para cada n comenzando desde el muy optimista número de más de veintitrés *clusters* en común, descendientemente. Cabe destacar que estas corridas se hicieron por separado debido a los extensos tiempos de ejecución que estas llevaron y se eligió este número por representar “más de la mitad del curso”. Más específico aún, se trata del número de la última entrega de la última unidad anterior al primer parcial (el ejercicio *medidas especies*), momento para el cual sería deseable tener una noción de quiénes continuarán con el curso.

Como es de esperarse, a medida que se decrementa dicho n , la cantidad de alumnos en el *clique* aumenta ya que se vuelve más laxa la condición para formar parte de él. Notamos sin embargo que, a medida que se reducía el n , la proporción de alumnos que *no habían terminado el curso* dentro del *clique*, incrementaba. Esta tendencia se observó

hasta alcanzar los 13 *clusters* en común, número a partir del cual la proporción comenzaba a disminuir nuevamente.

De este grupo, denominado *Clique-13*, compuesto por 222 estudiantes y representando un 27% del alumnado, solo dos finalizaron el curso. El 99% restante de este grupo corresponde a más de la mitad (54%) de los estudiantes que no completaron el programa.

Una vez caracterizada esta mayoría, fue necesario analizar qué entregas habían realizado de manera similar, con el objetivo de identificar en el futuro aquellos ejercicios que podrían servir de guía para realizar estas estimaciones.

A partir de esto, se intentó clasificar a los estudiantes en tres grupos:

1. Los pertenecientes a *Clique-13*, que en su vasta mayoría no finalizaron el curso
2. Los estudiantes que no pertenecían a *Clique-13* y tampoco completaron la materia
3. Aquellos que continuaron con el programa hasta su finalización

3.3.2. Separación por grupos

Con el objetivo de caracterizar cada subgrupo, se analizaron los patrones de entrega de ejercicios de los tres grupos identificados. Los resultados de este análisis se visualizaron en el *heatmap* de la Figura 3.16, donde se representa la distribución temporal de las entregas realizadas por cada grupo a lo largo del curso.

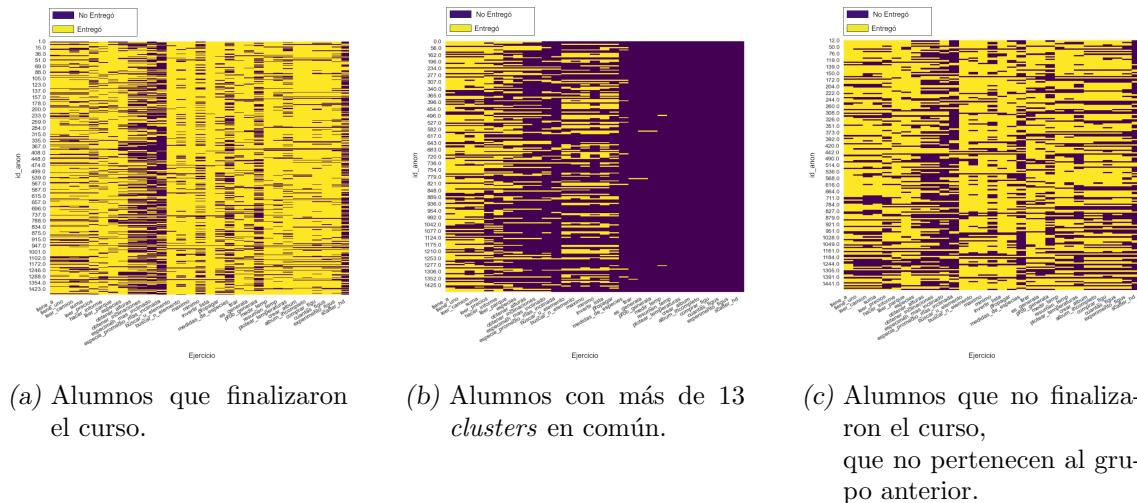


Fig. 3.7: Entregas por alumno según el ejercicio, ordenados cronológicamente.

Se puede observar entonces en la Figura 3.7b que los alumnos que componen este grupo de interés no solamente realizaron entregas similares a lo largo del curso si no que se trata del grupo que dejó de realizar entregas a partir del primer parcial (ejercicio *medidas_de_especies*). Esto podría servir entonces como un indicador para saber quiénes continuarán pasada esta primera instancia de evaluación y quiénes no, basado únicamente en las producciones de código de alumnos.

En cuanto a los estudiantes que continuaron con las entregas pasada este examen y aquellos que sí terminaron el curso, se observó que los ejercicios entregados por ambos grupos son similares. Siguiendo el lineamiento de la Figura 3.6, se analizaron los grupos en común, ahora específicamente para distinguir entre los estudiantes que no abandonaron después del primer parcial, como muestra la Figura 3.8.

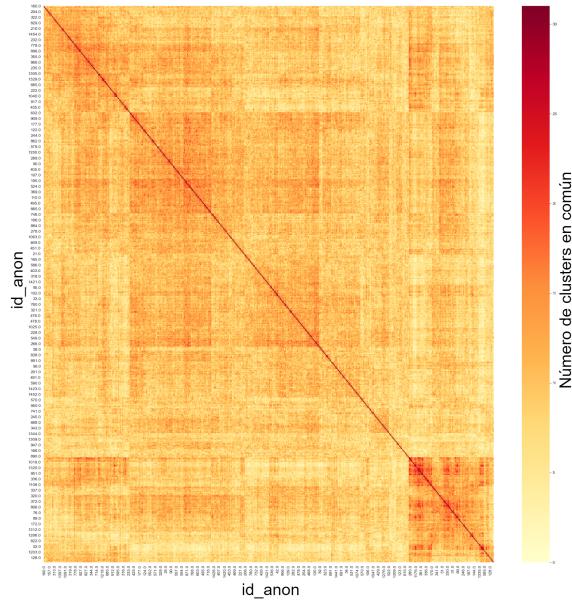


Fig. 3.8: Número de *clusters* en común de estudiantes que continuaron realizando entregas luego del primer parcial.

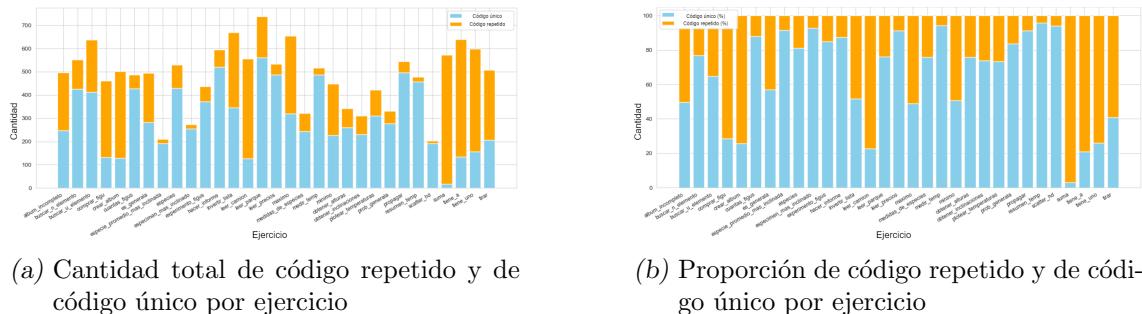
Siendo difícil distinguir por este medio, evidenciado en la la falta de agrupaciones observables en la figura anterior, surgió la necesidad de adoptar nuevas medidas para mejorar estos resultados.

3.4. Nuevos enfoques para mejorar los resultados

Para esta sección se intentó mejorar la separación entre aquellos alumnos que entregaron el primer examen parcial y aquellos que finalizaron el curso, entre quienes no fue posible distinguir en secciones anteriores.

3.4.1. Eliminación del código repetido

A partir del análisis anterior, se observó que, en algunos ejercicios, existía código idéntico entre distintos alumnos. Bajo la sospecha de que esto podía impactar negativamente en el desempeño general de la asignación de grupos, se identificó la proporción de entregas exactamente iguales por función. Para ello se analizaron tanto la cantidad absoluta de producciones de código idéntico en nuestra base, como el porcentaje que esta representaba dentro de cada ejercicio, como se observa en la Figura 3.9.



(a) Cantidad total de código repetido y de código único por ejercicio

(b) Proporción de código repetido y de código único por ejercicio

Fig. 3.9: Código repetido a lo largo de los 31 entregables, ordenados por orden cronológico.

Como es de esperarse, evidenciado en la Figura 3.9, en ejercicios más sencillos la proporción de código idéntico es mayor. Esto podría deberse a la simpleza del funcionamiento esperado de dichas funciones (como, por ejemplo, en el caso *suma*).

A continuación, se realizó un *mapeo* entre alumnos con código idéntico y el programa en cuestión. Una vez realizado esto, se generó un nuevo *dataset* sin dichas producciones. Se realizó entonces el procedimiento similar anterior: Reducción de la dimensionalidad de los datos con códigos únicos usando PCA → Aplicación de *Kmeans* → Re-inclusión de los alumnos que habían sido *mapeados* y eliminados anteriormente. Luego, se realizó una comparación ejercicio a ejercicio entre el *heatmap* original y aquél producto del procedimiento anterior. A modo de ejemplo, en la Figura 3.10 podemos observar el par de gráficos para el ejercicio *obtener_inclinaciones*.

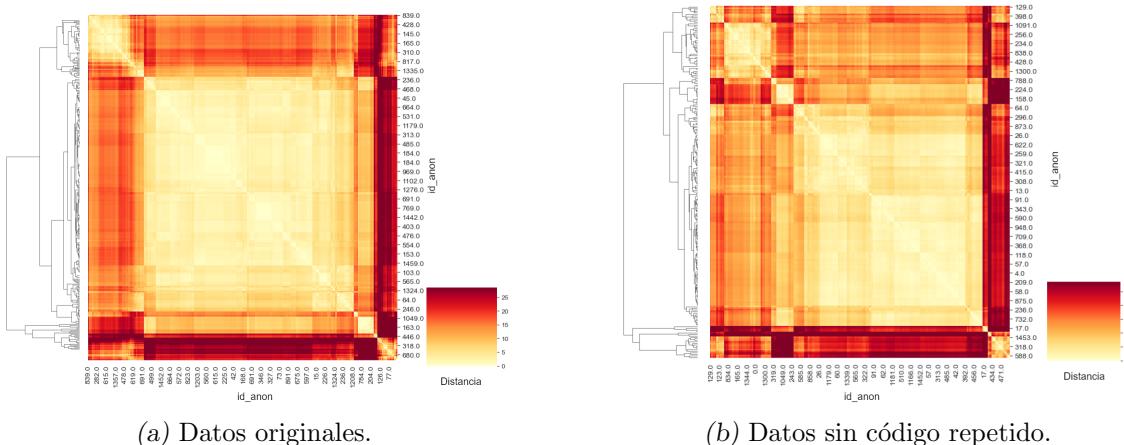


Fig. 3.10: Heatmaps para el ejercicio *obtener_inclinaciones* de la unidad tres con el agrupamiento realizado con los datos originales y con los datos sin el código repetido.

Como se observa en la Figura 3.10, a modo de ejemplo, no se observó una diferencia significativa entre el *clustering* original y aquél realizado contemplando lo mencionado anteriormente para ninguno de los 31 ejercicios.

Como última medida, se observó la proporción *cluster a cluster* de alumnos que habían finalizado el curso, esperando ver una diferencia significativa, como muestra la Figura .

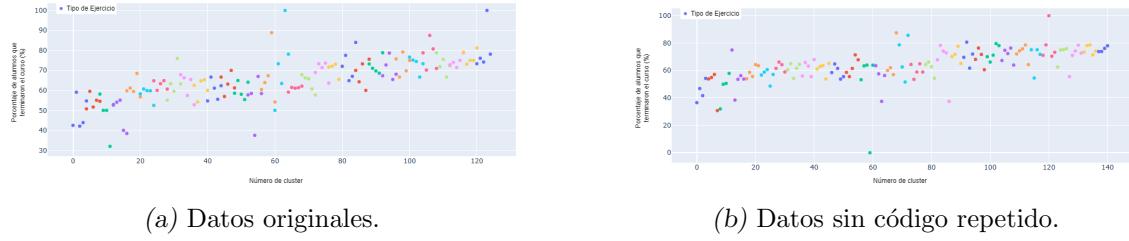


Fig. 3.11: Comparación entre el porcentaje de alumnos que finalizaron el curso *cluster a cluster* para los datos originales y los datos después de haber *mapeado* el código repetido.

Hubiera sido deseable observar que las diferencias entre dichas proporciones para cada ejercicio estuvieran más marcadas. Por el contrario, parecería haber una mejor separación de los dos grupos utilizando los datos originales.

Luego, no parecería haber un indicador de que el código idéntico incida sobre la *clusterización* de los alumnos.

3.4.2. Inserción de datos de interés

Para esta sección se le intentó hacer foco a ciertas características del código producido por los estudiantes, cuya extracción se comentó en la Sección 2.2.2. Se hizo un particular hincapié en el número de líneas, agregándolo como una nueva componente tanto en el vector de *embeddings* como luego de su reducción de la dimensionalidad. En ninguno de los dos casos se observaron cambios significativos en el agrupamiento. Una posible explicación es que el modelo ya contempla estas características inherentes al código en cuestión, por lo que no parecería estar incorporándose información adicional.

3.4.3. Usando *embeddings* normalizados

Se realizó nuevamente un análisis análogo al de la Figura 3.6, obteniéndose resultados análogos, como muestra la Figura 3.12.

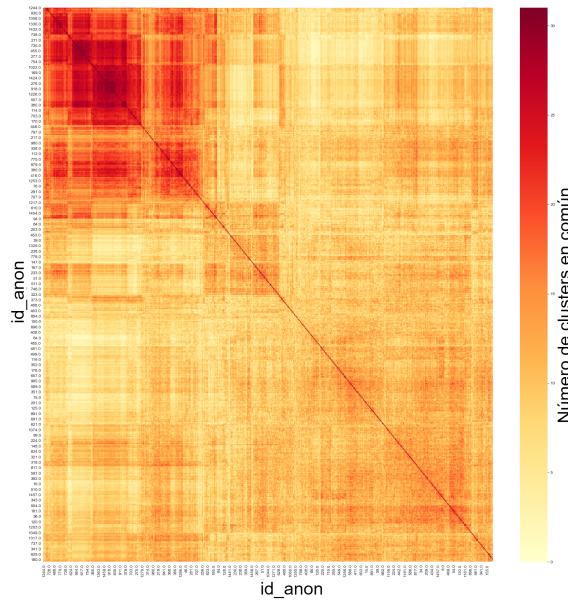


Fig. 3.12: Cantidad de *clusters* compartidos por estudiante con *embeddings* con normalizados.

Esto no agregó información adicional por sobre lo realizado anteriormente. Tras haber replicado el procedimiento de los *cliques*, estos retornaron resultados muy similares a los anteriores, teniendo como diferencia apenas un par de estudiantes por grupo.

3.4.4. Fuzzy-2-means

Una característica útil de *Fuzzy-2-means* es que este permite que algunos puntos pertenezcan *parcialmente* a un grupo. En pos de distinguir entre aquellos estudiantes que habían finalizado el curso y a los que habían continuado más allá del primer parcial pero sin terminar la materia, se aplicó *Fuzzy-2-means* a los datos originales como muestra la Figura 3.13.

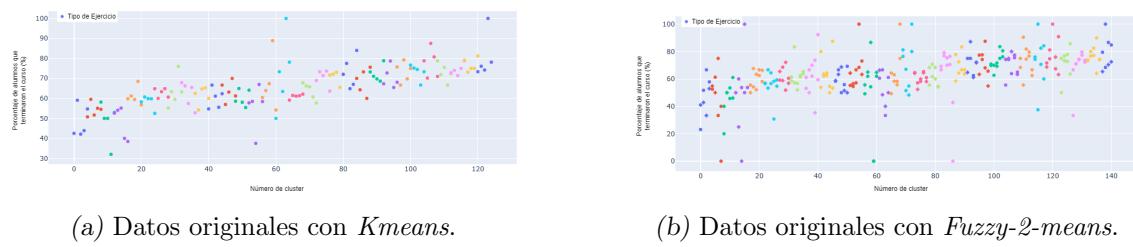


Fig. 3.13: Comparación entre el porcentaje de alumnos que finalizaron el curso *cluster* a *cluster* para los datos originales con distintos algoritmos de agrupación.

No se observaron alumnos a los cuales asignara parcialmente a un grupo, perteneciendo

todos en su totalidad a alguno de los *clusters* mostrados en la figura anterior. Si se observó un incremento en la cantidad de *clusters* que, si bien son más homogéneos, se debe a que estos son muy pequeños, contando con una cantidad de apenas algunos estudiantes. Luego, este algoritmo no proporcionó información adicional por sobre el *Kmeans* tradicional, para este caso particular.

3.5. Representando a los alumnos como un único vector

Basándonos en los resultados anteriores, se exploró un nuevo enfoque que consistió en utilizar los vectores generados por el modelo *CodeBERT*, concatenándolos según el orden cronológico del curso para obtener una única representación vectorial por estudiante. La hipótesis subyacente era que esta representación unificada permitiría una mejor discriminación entre los estudiantes que completaron el curso y aquellos que, si bien continuaron después del primer examen, no lo finalizaron.

3.5.1. Vectores de ceros

Para implementar técnicas de reducción de dimensionalidad y agrupamiento, es fundamental que todos los vectores de estudiantes tengan la misma dimensión. Esto presentó un desafío importante: ¿cómo manejar las entregas no realizadas por los alumnos?

En una primera aproximación, se construyó el vector de cada estudiante de la siguiente manera:

- **Para entregas realizadas:** se utilizó el *embedding* generado por *CodeBERT* correspondiente a dicha entrega.
- **Para entregas ausentes:** se insertó un vector de 768 ceros (coincidiendo con la dimensión del *output* de *CodeBERT*).

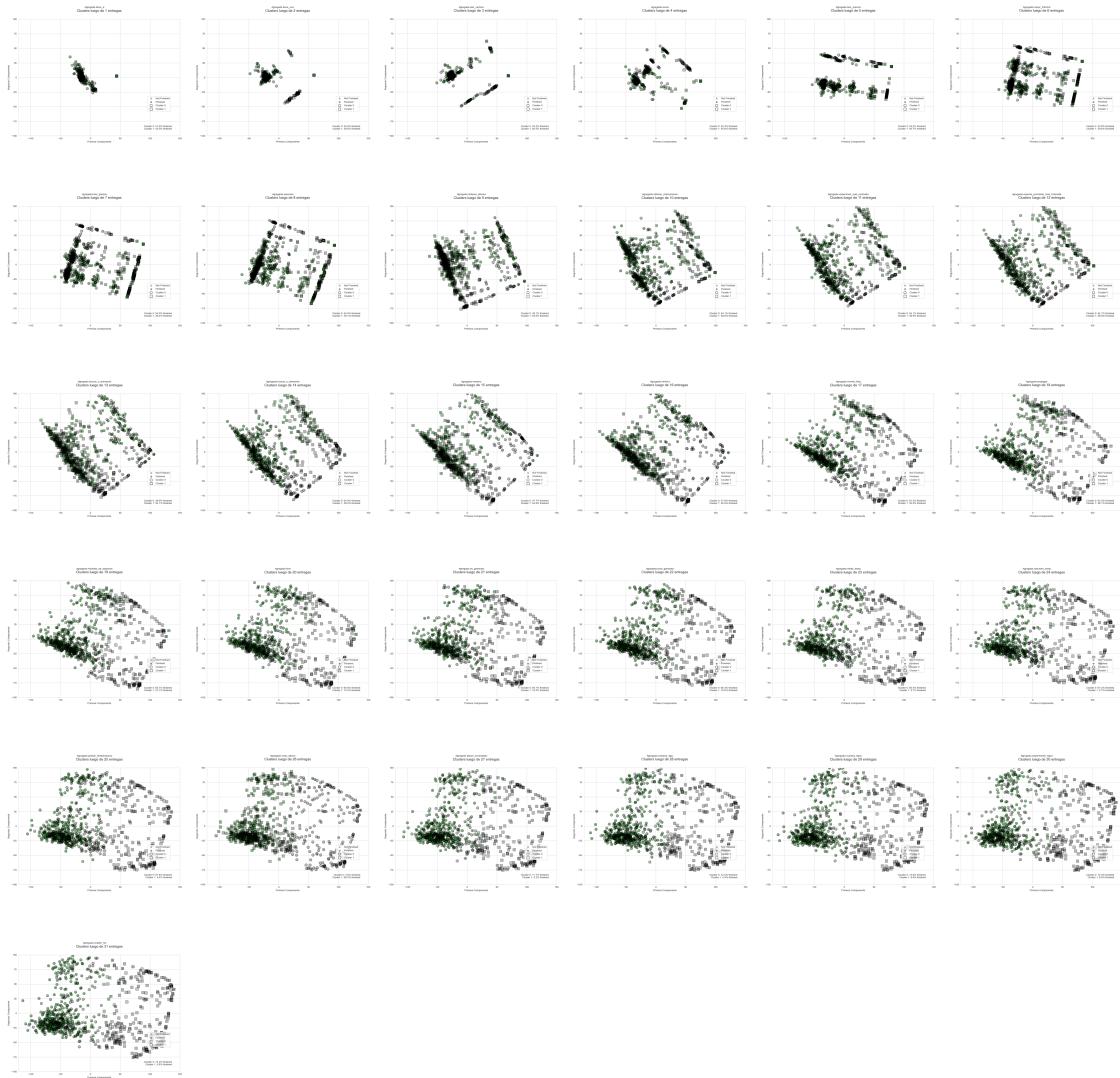


Fig. 3.14: Evolución del agrupamiento mediante PCA y *Kmeans* ($k = 2$) a lo largo del curso. Cada gráfico representa la adición secuencial de ejercicios, donde las entregas faltantes se completaron con vectores nulos.

Como se observa en la Figura 3.14, la separación entre grupos se vuelve más pronunciada a medida que se incorporan más ejercicios al análisis. Sin embargo, esta aparente mejora en la clasificación requiere un análisis más profundo, como se discutirá más adelante.

3.5.2. Función vacía

Considerando que un vector de ceros podría no tener un significado semántico claro en el espacio vectorial de *CodeBERT*, se implementó una segunda estrategia. En este caso, las entregas faltantes se representaron mediante el *embedding* de una “función vacía”

específica para cada ejercicio. El proceso de construcción fue el siguiente:

- **Para entregas realizadas:** se mantuvo el *embedding* original de *CodeBERT*.
- **Para entregas ausentes:** se generó el *embedding* de una función vacía contextualizada al ejercicio correspondiente.

Por ejemplo, para el ejercicio *buscar_u_elemento*, la función vacía se definió como el descrito en la Figura 3.5.2.

```
1 def buscar_u_elemento():
2     pass
```

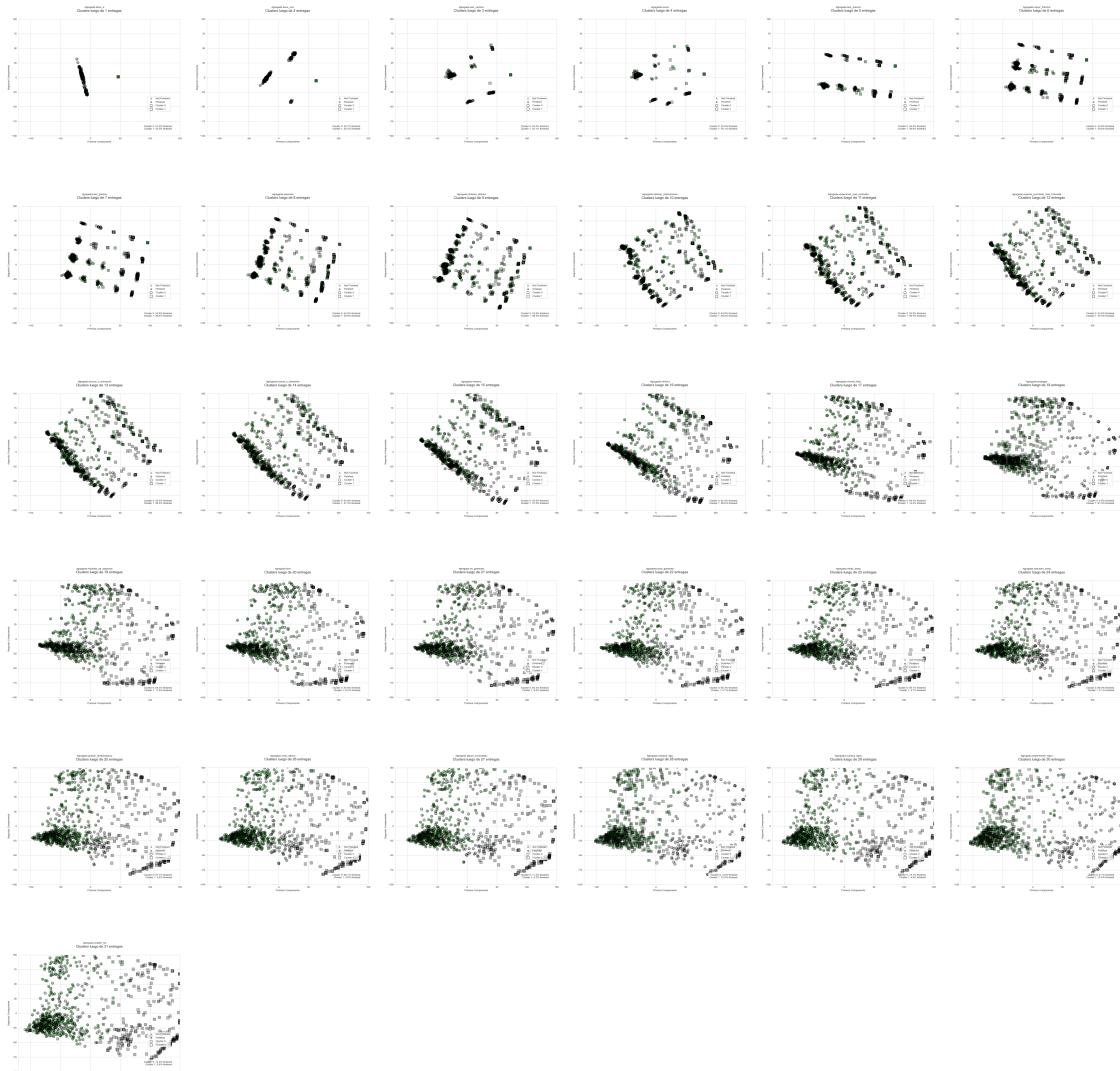


Fig. 3.15: Evolución del agrupamiento utilizando PCA y Kmeans ($k = 2$) a lo largo del curso. Los ejercicios no entregados se representaron mediante el *embedding* de una función vacía contextualizada.

En la Figura 3.15, se observa una tendencia similar a la del enfoque anterior: la separación entre grupos se acentúa conforme se incorporan más ejercicios al análisis. Las diferencias entre ambos métodos de completado no resultan sustanciales en términos de la calidad del agrupamiento.

3.5.3. Análisis de la proporción de entregas

Para comprender mejor la naturaleza de los agrupamientos observados, se realizó un análisis complementario examinando la proporción de entregas por ejercicio, como muestra la Figura 3.16.

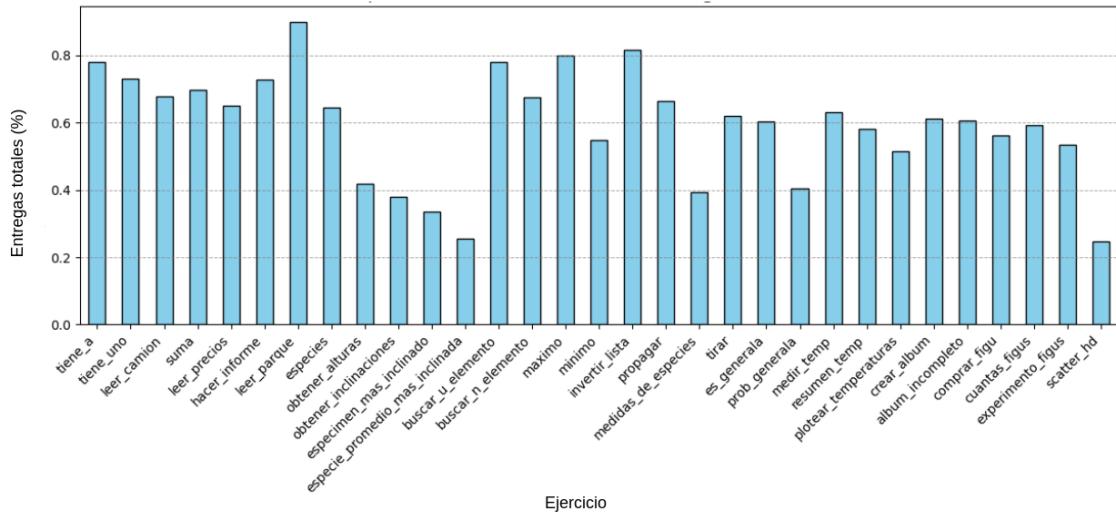


Fig. 3.16: Distribución temporal de las entregas realizadas por ejercicio, expresada como proporción del total de estudiantes.

Al contrastar esta información con los resultados de agrupamiento (Figuras 3.14 y 3.15), se reveló un patrón significativo: la aparente mejora en la clasificación está más relacionada con la tasa de participación que con las características intrínsecas de las soluciones. Por ejemplo, para el ejercicio *experimento_figus*, aunque el *cluster_0* contiene un 74 % de estudiantes que finalizaron el curso, este porcentaje aumenta al 80 % cuando se considera únicamente a quienes entregaron dicho ejercicio.

Es notable que en las etapas iniciales del curso, las proporciones entre grupos se mantienen cercanas al 50 %, sugiriendo una clasificación poco efectiva. Este análisis nos lleva a concluir que el algoritmo de agrupamiento está capturando principalmente patrones de participación (entregas realizadas vs. no realizadas) en lugar de características distintivas entre estudiantes que completarán o no el curso. Esta observación sugiere que la mera pre-

sencia o ausencia de entregas podría ser un predictor más simple y efectivo que el análisis del contenido de las soluciones para este propósito específico.

4. CONCLUSIÓN

Este trabajo abordó el análisis del comportamiento estudiantil en un curso Introductorio a Python en contexto de pandemia, centrándose exclusivamente en el análisis de las producciones de código de los alumnos. El objetivo principal fue identificar patrones que permitieran detectar grupos con mayor propensión a la deserción.

El *dataset* analizado comprende las entregas de 819 estudiantes a lo largo del curso, considerando únicamente la última versión de cada ejercicio como producción definitiva. El preprocessamiento de los datos incluyó la anonimización del código, para preservar la privacidad del alumnado, y la limpieza de los datos, que permitió eliminar entregas incorrectas y reducir la cantidad de código no procesable por el modelo. Para garantizar la seguridad de esta información sensible, se implementó una base de datos *local* que permitió el almacenamiento y procesamiento seguro de los datos.

En un primer análisis, se realizó un estudio detallado de las entregas examinando cada ejercicio de manera individual. Si bien se identificaron grupos claramente definidos para algunos ejercicios, no se encontró una correlación significativa entre estos patrones de agrupamiento y las variables objetivo de predicción. Esta falta de asociación sugiere que las características que definen los grupos no necesariamente están alineadas con los indicadores de rendimiento que se buscaba predecir.

En un segundo análisis, se realizó una comparación exhaustiva entre todos los *clusters* generados a lo largo del curso. Los resultados revelaron valores bajos tanto en los índices ARI (*Adjusted Rand Index*) como NMI (*Normalized Mutual Information*). De manera similar, los índices de Jaccard también mostraron valores reducidos. Estos resultados, en conjunto, sugieren una baja similitud estructural entre las diferentes particiones generadas, indicando que los patrones de agrupamiento varían significativamente entre las distintas etapas del curso.

En un tercer análisis, se examinó la distribución de *clusters* compartidos por cada estudiante. Durante esta exploración, se identificó un grupo de particular interés: aquellos estudiantes que compartían 13 o más *clusters* a lo largo del curso. Un análisis posterior reveló hallazgos significativos sobre este conjunto. No solo representaba más de la mitad de los estudiantes que eventualmente abandonarían la materia, sino que, más específicamente, englobaba a aquellos que la abandonarían después de la primera instancia de evaluación. Esta observación sugiere un patrón de comportamiento distintivo que podría servir como

indicador temprano de deserción.

Se implementaron diversos enfoques con el objetivo de mejorar la distinción entre dos grupos de estudiantes: aquellos que completaron el curso y aquellos que mantuvieron su participación, al menos, hasta la quinta unidad incluida en este *dataset*. Las estrategias exploradas incluyeron: (i) el análisis de la incidencia del código repetido en la *clusterización*, (ii) la incorporación de información adicional relevante, (iii) la normalización de los *embeddings*, y (iv) la implementación de una variante del algoritmo *K-means* para el agrupamiento. Sin embargo, ninguna de estas aproximaciones resultó en mejoras significativas respecto a los resultados previamente obtenidos.

Finalmente, se adoptó un enfoque alternativo donde cada estudiante fue representado como un único vector. Para abordar los ejercicios no entregados, se exploraron dos estrategias de completado de datos: inicialmente se utilizaron vectores nulos (de ceros) y posteriormente se implementó el *embedding* correspondiente a la función vacía. Ambos enfoques retornaron resultados similares; sin embargo, tanto la reducción de dimensionalidad como la *clusterización* resultante reflejaron principalmente patrones relacionados con la cantidad de entregas realizadas por cada estudiante, en lugar de identificar efectivamente a quienes completarían el curso. Esta observación sugiere que la representación vectorial propuesta capturó más la frecuencia de participación que los indicadores de éxito académico.

Los resultados obtenidos sugieren que, si bien el análisis del código por sí solo no fue suficiente para predecir con precisión la finalización del curso, permitió identificar patrones de comportamiento tempranos potencialmente indicativos de deserción, particularmente en estudiantes que comparten características de agrupamiento específicas después de la primera evaluación. Este hallazgo podría ser valioso para el desarrollo de estrategias de intervención temprana en futuros cursos.

5. TRABAJO FUTURO

Cerdeiro et al. destacan que durante la pandemia se implementaron diversos canales de comunicación, como Slack, lo cual facilitó que las *consultas estuvieran organizadas de manera que los estudiantes pudieran buscarlas, revisarlas y comentarlas fácilmente, evitando así su repetición* [11]. Por su parte, en el trabajo “Embedding navigation patterns for student performance prediction”, Loginova et al. desarrollaron un método para predecir el rendimiento académico estudiantil analizando patrones de navegación en plataformas educativas en línea, en lugar de basarse en calificaciones previas [30]. Su investigación reveló que el análisis de las secuencias de acciones de los estudiantes permitió identificar diferentes estrategias de aprendizaje entre alumnos de alto y bajo rendimiento, información valiosa para implementar sistemas de alerta temprana. Siguiendo esta línea, podríamos explorar la posibilidad de predecir qué estudiantes completarán el curso, considerando no solo el código que producen sino también sus patrones de interacción en la plataforma. Queda pendiente, además, indagar más sobre los comentarios dejados por los alumnos dentro del código. Esta información adicional podría servir de ayuda a la hora de realizar los agrupamientos futuros.

La estructura actual de nuestra base de datos facilitaría la incorporación de variables adicionales, como la cantidad de entregas por alumno. Un análisis más exhaustivo podría examinar la trayectoria individual de cada estudiante, contemplando tanto su progreso entre ejercicios como su evolución dentro de cada entrega.

Para investigaciones futuras, además de predecir la finalización del curso, podríamos estimar la presentación a los exámenes parciales. Específicamente, sería valioso analizar si los estudiantes que comparten más de 13 *clusters* llegaron a presentarse a la evaluación, más allá de haber dejado de entregar trabajos después de la primera evaluación. Esta información resultaría particularmente útil para la planificación docente y la gestión de los tiempos de corrección. También podríamos extender nuestro análisis a la segunda instancia de evaluación, enfocándonos especialmente en aquellos alumnos que, si bien no completaron el curso, tampoco lo abandonaron antes del primer parcial - un grupo que representa el 46 % de los estudiantes que no pudieron ser diferenciados en nuestro análisis inicial.

Bibliografía

- [1] Consejo Federal de Educación. Resolución nro. 263/15. <http://www.me.gov.ar/consejo/resoluciones/res15/263-15.pdf>, 2015.
- [2] UBA DOV. Datos y gráfico obtenidos de la dirección de orientación vocacional de la uba, 2024. Contacto: dov@de.fcen.uba.ar.
- [3] Javier Massa. Los primeros datos. <https://exactas.uba.ar/los-primeros-datos/>, 2022.
- [4] Nicolas Varaschin 187/08 Daniel Paz. *Análisis exploratorio y estadístico sobre el alumnado de Ciencias de la Computación en FCEN-UBA*. PhD thesis, Universidad de Buenos Aires, 2020.
- [5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.
- [6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, and Ming Zhou. Codebert - base. <https://huggingface.co/microsoft/codebert-base>, 2020.
- [7] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, 53(4):465–480, 2018.
- [8] Sonja Johnson-Yu, Nicholas Bowman, Mehran Sahami, and Chris Piech. Simgrade: Using code similarity measures for more accurate human grading, 2024.
- [9] Siegfried Nijssen Simone Brigante, Silvia Chiusano. *Evaluation of Students' Source Code Submissions Using Machine Learning*. PhD thesis, Politecnico di Torino, 2020.
- [10] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. Zero shot learning for code education: Rubric sampling with deep learning inference, 2018.
- [11] Manuela Cerdeiro, José Emilio Crespo, Oscar Filevich, Rafael Grimson, and Matías López y Rosenfeld. Escalando la enseñanza de programación en tiempos de pandemia: desafíos y oportunidades. *Cartografías del Sur. Revista de Ciencias, Artes y Tecnología*, 13(1), jun. 2021.

-
- [12] Jonathan Webster and Chunyu Kit. Tokenization as the initial phase in nlp. pages 1106–1110, 01 1992.
 - [13] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip Guo, and Robert Miller. Overcode: visualizing variation in student solutions to programming problems at scale. In *Adjunct Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST ’14 Adjunct, page 129–130, New York, NY, USA, 2014. Association for Computing Machinery.
 - [14] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, Rice University, Houston, Texas, 2011.
 - [15] Python Software Foundation. Abstract syntax trees — ast — python 3 documentation, 2024. Accessed: 2024-10-20.
 - [16] Hugging Face. Roberta model documentation, 2024. Accessed: 2024-10-20.
 - [17] sklearn.preprocessing.normalize. <https://scikit-learn.org/1.5/modules/generated/sklearn.preprocessing.normalize.html>. [Online]. Available: 2024-10-19.
 - [18] Mongodb documentation. <https://www.mongodb.com/docs/>. [Online].
 - [19] Pymongo. <https://pymongo.readthedocs.io/en/stable/>. [Online].
 - [20] Tomáš Musil. Examining structure of word embeddings with pca. *arXiv preprint arXiv:1906.00114*, 2019.
 - [21] Ali Basirat. Word embedding through pca. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.
 - [22] David Azcona, Piyush Arora, I-Han Hsiao, and Alan Smeaton. user2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, LAK19, page 86–95, New York, NY, USA, 2019. Association for Computing Machinery.
 - [23] Kristina P. Sinaga and Miin-Shen Yang. Unsupervised k-means clustering algorithm. *IEEE Access*, 2020.
 - [24] Abiodun Ikotun, Absalom Ezugwu, Laith Abualigah, Belal Abuhaija, and Jia Heming. K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*, 622, 12 2022.

- [25] Edy Umargono, Jatmiko Endro Suseno, and S.K Vincensius Gunawan. K-means clustering optimization using the elbow method and early centroid determination based on mean and median formula. In *Proceedings of the 2nd International Seminar on Science and Technology (ISSTEC 2019)*, pages 121–129. Atlantis Press, 2020.
- [26] K. P. Chan and Y. S. Cheung. Clustering of clusters. *Pattern Recognition*, 25(2):211–217, 1992.
- [27] Yun Yang. Chapter 3 - temporal data clustering. In Yun Yang, editor, *Temporal Data Mining Via Unsupervised Ensemble Learning*, pages 19–34. Elsevier, 2017.
- [28] Alessia Amelio and Clara Pizzuti. Correction for closeness: Adjusting normalized mutual information measure for clustering comparison: Correction for closeness: Adjusting nmi. *Computational Intelligence*, 33:8–9, 09 2016.
- [29] Vijay Kotu and Bala Deshpande. Chapter 4 - classification. In Vijay Kotu and Bala Deshpande, editors, *Data Science (Second Edition)*, pages 65–163. Morgan Kaufmann, second edition edition, 2019.
- [30] Ekaterina Loginova and Dries F. Benoit. Embedding navigation patterns for student performance prediction. In *Proceedings of The 14th International Conference on Educational Data Mining (EDM 2021)*, pages 391–399. International Educational Data Mining Society, 2021.
- [31] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, 2008.