

Generación de datos multi-ómicos sintéticos con Generative Adversarial Networks



Universitat Oberta
de Catalunya

Federico Lara Salas

STATISTICAL BIOINFORMATICS
AND MACHINE LEARNING

Máster en Bioinformática y Bioestadística

Nombre del tutor/a de TF:
Esteban Vegas

Nombre del/de la PRA:
Carles Ventura Royo

14 de enero de 2024



Esta obra esta sujeta a una licencia de Reconocimiento
<https://creativecommons.org/licenses/by-nc/3.0/es/>

Ficha Del Trabajo Final

Título del trabajo:	Generación de datos multi-ómicos sintéticos con Generative Adversarial Networks
Nombre del autor/a:	Federico Lara Salas
Nombre del tutor/a de TF:	Esteban Vegas
Nombre del/de la PRA:	Carles Ventura Royo
Fecha de entrega:	14 de enero de 2024
Titulación o programa:	Máster en Bioinformática y Bioestadística
Área del trabajo final:	STATISTICAL BIOINFORMATICS AND MACHINE LEARNING
Idioma del trabajo:	Castellano
Palabras clave:	Integración datos multi-ómicos , GAN, Generación datos sintéticos multi-ómicos, Transformer

Resumen del trabajo

Este trabajo se centra en la generación de datos sintéticos multi-ómicos con GANs. Se ha investigado y experimentado además con VAEs para compararlos con las GANs en la generación de datos sintéticos multi-ómicos. En un principio se quería utilizar la arquitectura Transformer para mejorar la generación de datos de las GANs pero al iniciar la experimentación se observó un rendimiento muy pobre y se cambió ese objetivo por el de investigar y experimentar con GANs y Transformers la conversión entre datos ómicos, en concreto de expresión génica a metilación y viceversa.

Este trabajo es relevante en campos como la genómica y la epigenética, donde la generación y conversión de datos ómicos sintéticos pueden ser herramientas valiosas para la investigación y el desarrollo de tratamientos personalizados.

Se han usado datos simulados con el paquete InterSIM y datos reales extraídos de <https://adex.genyo.es/> con la serie GSE117931 que contiene datos de expresión génica y metilación para la esclerosis sistémica.

Para la generación de datos sintéticos las GANs han demostrado ser ligeramente superiores en los datos simulados y notablemente superiores en los datos reales a los VAE. En el caso de la conversión de datos ómicos las GANs han sido ligeramente superiores en la conversión de expresión a metilación y los Transformers ligeramente superiores en la conversión de metilación a expresión. Los datos reales tenían muy pocas muestras para conseguir un rendimiento adecuado en la conversión de datos ómicos.

Como conclusión podemos afirmar que las GANs demuestran ser una herramienta valiosa para la generación de datos sintéticos multiómicos y la conversión de datos ómicos y los Transformers parecen también muy adecuados para la conversión de datos ómicos.

Abstract

This work focuses on the generation of synthetic multi-omic data using GANs. It also investigates and experiments with VAEs to compare them with GANs in generating synthetic multi-omic data. Initially, the Transformer architecture was intended to improve GANs' data generation, but poor performance led to a shift in focus to investigating and experimenting with GANs and Transformer in omic data conversion, specifically from gene expression to methylation and vice versa.

This work is relevant in fields like genomics and epigenetics, where generating and converting synthetic omic data can be valuable tools for research and developing personalized treatments.

Simulated data were used with the InterSIM package and real data from <https://adex.genyo.es/> with the GSE117931 series, which includes gene expression and methylation data for systemic sclerosis.

For synthetic data generation, GANs have proven slightly superior with simulated data and notably superior with real data in comparison to VAE. In omic data conversion, GANs were slightly better in converting expression to methylation, and Transformer slightly better in converting methylation to expression.

Real data had too few samples for adequate performance in omic data conversion. In conclusion, GANs prove to be a valuable tool for generating synthetic multi-omic data and omic data conversion and Transformer also seem very suitable for omic data conversion.

Índice general

1. Introducción	7
1.1. Contexto y justificación del trabajo	7
1.2. Descripción general	8
1.3. Objetivos del trabajo	9
1.3.1. Objetivos generales	9
1.3.2. Objetivos específicos	9
1.4. Impacto en sostenibilidad, ético-social y de diversidad	10
1.4.1. Sostenibilidad	10
1.4.2. Comportamiento ético y responsabilidad social	10
1.4.3. Diversidad y derechos humanos	10
1.5. Enfoque y método seguido	11
1.6. Planificación del trabajo	11
1.6.1. Tareas	12
1.6.2. Calendario	13
1.6.3. Hitos	13
1.7. Análisis de riesgos	13
1.8. Desviaciones del plan	14
1.9. Breve sumario de productos obtenidos	14
1.10. Breve descripción de los otros capítulos de la memoria	14
2. Materiales y Métodos	16
2.1. Investigación Teórica y Revisión Bibliográfica	16
2.2. Introducción a las arquitecturas usadas	18
2.2.1. Generative Adversarial Network	18
2.2.2. Variational AutoEncoder	23
2.2.3. Transformer	25
2.3. Entorno y Datos	29
2.3.1. Herramientas de Software y Entorno de Trabajo	29
2.3.2. Generación de datos simulados	30
2.3.3. Preprocesamiento de datos simulados	31
2.3.4. Obtención de datos reales	32
2.3.5. Preprocesamiento de datos reales	32
2.4. Generación de datos sintéticos multiómicos	34
2.4.1. GAN	34

2.4.2. VAE	39
2.4.3. Integración de arquitectura Transformer en GAN y exploración de modelos	43
2.5. Conversión entre datos ómicos	44
2.5.1. Exploración modelos GAN y TRANSFORMER Conversión Expresión <->Metilación	45
2.5.2. Modelo final TRANSFORMER Conversión Expresión→Metilación	46
2.5.3. Modelo final GAN Conversión Expresión→Metilación	47
2.5.4. Modelo final TRANSFORMER Conversión Metilación→Expresión	48
2.5.5. Modelo final GAN Conversión Metilación→Expresión	50
2.5.6. Validación Conversión Expresión <->Metilación	51
3. Resultados	53
3.1. Generación datos multi-ómicos	53
3.1.1. Modelo GAN Final con datos simulados	53
3.1.2. Modelo GAN Final con datos reales	60
3.1.3. Modelo VAE Final con datos simulados	67
3.1.4. Modelo VAE Final con datos reales	74
3.1.5. Comparación modelos GAN y VAE	80
3.2. Conversión datos ómicos	80
3.2.1. Modelos TRANSFORMER Conversión Expresión→Metilación	81
3.2.2. Modelos GAN Conversión Expresión→Metilación	83
3.2.3. Modelos TRANSFORMER Conversión Metilación→Expresión	85
3.2.4. Modelos GAN Conversión Metilación→Expresión	86
3.2.5. Comparación modelos Transformer y GAN	88
4. Conclusiones y trabajos futuros	89
5. Anexos	96
5.1. Anexo A. Requisitos entorno Python	96
5.2. Anexo B. Análisis estadísticos de los datos	100
5.2.1. Análisis estadístico de los datos simulados	100
5.2.2. Análisis estadístico de los datos reales	117
5.3. Anexo C. Código de preprocesamiento y validación	134
5.3.1. Preprocesamiento datos simulados	134
5.3.2. Validación modelo GAN	136
5.3.3. Validación modelo VAE	140
5.3.4. Procesamiento datos conversión E <->M	143
5.3.5. Validación modelo TRANSFORMER E <->M	144
5.3.6. Validación modelo GAN E <->M	146

Capítulo 1

Introducción

1.1. Contexto y justificación del trabajo

En la era moderna de la biología y la medicina, el estudio de los datos multi-ómicos se ha convertido en un pilar fundamental para entender la complejidad subyacente en los sistemas biológicos. Estos datos representan distintos niveles de información molecular, desde genómica hasta proteómica y metabolómica, y su integración proporciona una imagen más holística y precisa del funcionamiento y la dinámica celular. [23][34][8][2][31]

Sin embargo, a pesar de la riqueza de la información que pueden proporcionar, enfrentamos desafíos fundamentales en la adquisición, integración y análisis de datos multi-ómicos.[35]

El primer desafío es la necesidad de generar un volumen suficiente de datos multi-ómicos de alta calidad. La adquisición de tales datos es costosa, consume mucho tiempo y, en ocasiones, puede ser inviable debido a limitaciones técnicas o de muestra. Además, dado que los datos multi-ómicos provienen de diversas fuentes y plataformas, su integración se ve obstaculizada por la heterogeneidad, las diferencias de escala y la falta de estándares uniformes.

La relevancia del tema radica en el potencial de los datos multi-ómicos para revolucionar nuestra comprensión de enfermedades complejas, mecanismos biológicos y para desarrollar terapias más efectivas. Actualmente, los enfoques tradicionales para abordar estos desafíos incluyen métodos estadísticos, algoritmos de integración y herramientas computacionales, pero aún enfrentan limitaciones en cuanto a eficiencia, escala y resolución.[23][34][8][17][20]

En este Trabajo de Final de Máster, proponemos implementar una innovadora solución basada en las redes generativas antagónicas (GANs, por sus siglas en inglés). Las GANs han demostrado ser herramientas poderosas en diversos campos para generar datos sintéticos que imitan distribuciones reales. Nuestra hipótesis es que al usar GANs, podemos generar datos multi-ómicos sintéticos de alta calidad que superen las barreras de adquisición y heterogeneidad.[12][1][22][37]

El resultado que buscamos obtener es un modelo GAN robusto y versátil que pueda integrar efectivamente múltiples ómicas, generando datos sintéticos que no solo sean consistentes

dentro de cada dominio ómicos, sino también entre dominios. Esperamos que este modelo facilite la investigación en biomedicina y biología, proporcionando un recurso valioso para simulaciones, validaciones y descubrimientos.

Además, pretendemos mejorar el modelo con la introducción de una arquitectura Transformer en el generador que mejora el manejo de secuencias largas y añade el mecanismo de atención permitiendo aprender relaciones más complejas entre los datos. [29][11][9][38][24][19]

Con este trabajo, aspiramos a dar un paso más para cerrar la brecha entre la necesidad de datos multi-ómicos integrados y la capacidad actual de adquirir y procesar tales datos, ofreciendo una solución basada en la inteligencia artificial para avanzar en la investigación biomédica.

1.2. Descripción general

Para abordar el desafío de integrar y generar datos multi-ómicos, nuestra metodología se basa en una serie de etapas que permiten una transición fluida desde la teoría hasta la práctica.

Comenzamos con una investigación teórica y una revisión bibliográfica exhaustiva. Esta etapa es esencial, ya que nos sumergimos en la literatura científica, explorando los avances actuales y desafíos en la generación e integración de datos ómicos. Además, nos familiarizamos con los entresijos de las GANs y la arquitectura Transformer. El objetivo principal es establecer un marco teórico sólido, identificar las mejores prácticas y comprender las limitaciones de los enfoques actuales.

Una vez acabado esto, procederemos a la obtención y preparación de los datos. En primera instancia usaremos datos simulados que nos permiten tener de forma fácil y rápida datos para iniciar el desarrollo. Posteriormente se usarán datos reales de alguna base de datos para la validación del modelo.

Dado que el alcance del trabajo es bastante ambicioso, hemos decidido separar el trabajo en dos partes diferenciadas. Una en la que se diseña e implementa el modelo con GANs y otra en la que se mejora ese modelo con la arquitectura Transformer.

Tras tener una base teórica firme, pasamos al diseño de nuestro modelo usando GANs. En este proceso, delineamos la arquitectura, definimos los parámetros esenciales y decidimos las características específicas que nuestro modelo deberá tener. Es crucial en este punto contemplar cómo integraremos eficazmente los Transformers con las GANs, previendo posibles obstáculos y soluciones para superarlos.

Habiendo desarrollado un diseño riguroso, iniciamos la fase de implementación.. Aquí es donde las ideas y especificaciones cobran vida, transformándose en una herramienta práctica. Es natural que surjan desafíos técnicos durante esta fase, y es probable que se realicen ajustes basados en pruebas y descubrimientos iniciales.

Finalmente, llegamos a la validación. Es fundamental asegurarnos de que nuestro modelo no solo cumpla con las expectativas teóricas, sino que también se destaque en la práctica.

Comparamos su rendimiento con otros modelos existentes, analizamos la calidad de los datos generados y ajustamos cualquier imperfección que pueda surgir.

Después de esto pasaremos a la fase de mejorar el modelo integrando el Transformer, optimizarlo y validararlo. De este modo, queremos mitigar el riesgo de no obtener ningún software funcional por tener demasiado alcance.

En resumen, nuestra metodología se centra en construir sobre una base teórica sólida, diseñar cuidadosamente teniendo en cuenta las particularidades de nuestro tema, implementar con precisión y validar rigurosamente, todo con el objetivo de innovar y lograr resultados robustos y confiables.

1.3. Objetivos del trabajo

1.3.1. Objetivos generales

1. Desarrollar un entendimiento profundo de la integración y generación de datos multi-ómicos y su relación con GANs.
2. Implementar y validar un modelo que utilice GANs para la generación e integración de datos multi-ómicos.
3. Mejorar la generación de datos multi-ómicos utilizando Transformer, adaptando esta arquitectura avanzada para potenciar el rendimiento del modelo.

1.3.2. Objetivos específicos

- Para el objetivo general 1:
 - Realizar una revisión bibliográfica exhaustiva sobre la generación de datos multi-ómicos y las técnicas actuales de integración.
 - Analizar las propiedades y características de los datos multi-ómicos y cómo se pueden relacionar con la arquitectura de GANs.
 - Estudiar las ventajas y limitaciones de GANs en el contexto de datos multi-ómicos. Comparar con otras arquitecturas, como por ejemplo VAEs.
- Para el objetivo general 2:
 - Diseñar un modelo basado en GANs adecuado para la generación e integración de datos multi-ómicos.
 - Desarrollar un protocolo de validación para evaluar la calidad, autenticidad y coherencia biológica de los datos generados.
 - Comparar los resultados del modelo propuesto con otras técnicas de generación e integración en términos de precisión, velocidad y calidad.
- Para el objetivo general 3:

- Investigar la aplicabilidad de la arquitectura Transformer en la generación y integración de datos multi-ómicos.
- Adaptar los Transformers para trabajar en conjunto con GANs, potenciando las características distintivas de ambos métodos.
- Evaluar el impacto de la integración de Transformer en el rendimiento del modelo, considerando métricas como precisión, velocidad y calidad de datos generados.
- Analizar la interpretación de los resultados generados por el modelo mejorado, identificando las áreas donde los Transformer proporcionan ventajas adicionales.

1.4. Impacto en sostenibilidad, ético-social y de diversidad

1.4.1. Sostenibilidad

Dado que el TFM se centra en la integración y generación de datos multi-ómicos mediante GANs, es principalmente un trabajo computacional y teórico. En términos de consumo energético directo, la principal preocupación sería el uso de servidores y computadoras de alto rendimiento necesarias para el entrenamiento de modelos complejos, lo que podría llevar a un aumento en el consumo de energía eléctrica.

Uno de los impulsores detrás de este TFM es la promoción de la investigación eficiente y sostenible en el campo de la biomedicina. Al poder generar datos sintéticos de alta calidad, se reduce la necesidad de experimentos costosos, consumidores de recursos y contaminantes, alineándose así con los principios de sostenibilidad y conservación de recursos.

1.4.2. Comportamiento ético y responsabilidad social

La generación de datos sintéticos, especialmente en el campo biomédico, tiene implicaciones éticas y sociales. Por ejemplo, si los datos generados se utilizan en investigaciones biomédicas y no se diferencian adecuadamente de los datos reales, podría haber consecuencias en la integridad científica y en decisiones médicas basadas en estos datos.

A su vez, la generación automática de datos sintéticos podría reducir la necesidad de ciertas actividades manuales, como la curación y generación de datos. Esto impactaría en los trabajos y podría verse como una oportunidad para que los profesionales se centren en tareas más avanzadas.

1.4.3. Diversidad y derechos humanos

La generación de datos sintéticos puede llevar implícito el sesgo de los datos generados dependiendo del sesgo de los datos usados en el entrenamiento del modelo. Por ello, habría que poner especial atención en que los datos que se usan para entrenar el modelo tengan los menos sesgos por raza, género, etc...

1.5. Enfoque y método seguido

Entre las posibles estrategias a seguir, nos encontramos principalmente con las opciones de intentar llevar a cabo el modelo GAN con el Transformer integrado desde un principio, y, la opción más conservadora de primero conseguir un modelo con GAN y posteriormente modificarlo para introducir el Transformer en la red Generadora. Dado el alcance, el conocimiento y el tiempo disponible se ha optado por elegir esta segunda opción.

Las principales ventajas de elegir esta opción serían:

1. **Enfoque Incremental:** Adoptar un enfoque gradual permite construir sobre una base sólida y estable. Al comenzar con una GAN básica, se puede garantizar que se tiene un punto de partida funcional antes de introducir la complejidad adicional del Transformer. De esta manera, si surgieran problemas en etapas posteriores, sería más sencillo identificar si se deben a la implementación de la GAN en sí o a la incorporación del Transformer.
2. **Evaluación y Validación:** Trabajando inicialmente con una GAN básica, es posible realizar pruebas y validaciones en etapas tempranas, asegurando que el modelo satisface las expectativas en términos de generación de datos multi-ómicos sintéticos. Una vez establecido este punto de referencia, se puede proceder a integrar el Transformer y, posteriormente, comparar y contrastar los resultados obtenidos con ambas arquitecturas.
3. **Mitigación de Riesgos:** Al dividir el proceso en dos etapas claramente definidas, se reduce el riesgo de enfrentar problemas insuperables a medio camino. Si la integración del Transformer resultara ser particularmente desafiante o llevara a problemas inesperados, siempre se tendría el modelo GAN básico como un resultado tangible y valioso del proyecto.

El método que seguiremos consistirá en los siguientes pasos:

1. Investigación Teórica y Revisión Bibliográfica
2. Obtención y Preparación de Datos
3. Diseño del Modelo con GANs
4. Implementación del Modelo con GANs
5. Validación del Modelo GAN Básico
6. Integración y Optimización del Transformer
7. Validación Posterior con el Transformer Integrado
8. Documentación y Conclusión

1.6. Planificación del trabajo

1.6.1. Tareas

- Investigación preliminar y revisión bibliográfica (2 semanas)
 - Estudio del contexto y relevancia de los datos multi-ómicos.
 - Investigación sobre GANs y Transformer.
 - Establecimiento de un marco teórico.
- Análisis y preparación de datos (1 semana)
 - Identificación y recolección de conjuntos de datos multi-ómicos.
 - Pre-procesamiento y normalización.
- Diseño e implementación del modelo GAN para datos multi-ómicos (3 semanas)
 - Elección y diseño de la arquitectura de GAN.
 - Codificación y pruebas iniciales.
- Validación y comparación del modelo GAN (1 semana)
 - Establecimiento de protocolo de validación.
 - Comparación con otros métodos existentes.
- Estudio en profundidad del Transformer y adaptación al contexto (2 semanas)
 - Investigación sobre cómo los Transformers pueden mejorar el modelo GAN.
 - Adaptación para el contexto multi-ómicos.
- Integración del Transformer con el modelo GAN (2 semanas)
 - Diseño del modelo híbrido GAN-Transformer.
 - Codificación y pruebas iniciales del modelo combinado.
- Evaluación y optimización del modelo mejorado (2 semanas)
 - Evaluación del modelo híbrido.
 - Análisis y optimización basada en los resultados.
- Redacción de la memoria y preparación de otros entregables (2 semanas)
 - Redacción de la memoria del TFM.
 - Preparación de herramientas o productos resultantes.
- Revisión y finalización (1 semana)
 - Revisión general y ajustes finales del proyecto.
 - Preparación para la presentación.

1.6.2. Calendario

①	Name	Duration	Start	Finish	Predecessors
1	Investigación y revisión	10 days	9/27/23, 8:00 AM	10/10/23, 5:00 PM	
2	Analisis y preparación	5 days	10/11/23, 8:00 AM	10/17/23, 5:00 PM	1
3	Diseño e implementación	15 days	10/18/23, 8:00 AM	11/7/23, 5:00 PM	2
4	Validación y comparación	5 days	11/8/23, 8:00 AM	11/14/23, 5:00 PM	3
5	Estudio y adaptación	10 days	11/15/23, 8:00 AM	11/28/23, 5:00 PM	4
6	Integración Transform	10 days	11/29/23, 8:00 AM	12/1/23, 5:00 PM	5
7	Evaluación y optimización	10 days	12/13/23, 8:00 AM	12/26/23, 5:00 PM	6
8	Redacción y preparación	10 days	12/27/23, 8:00 AM	1/9/24, 5:00 PM	7
9	Revisión y finalización	5 days	1/10/24, 8:00 AM	1/16/24, 5:00 PM	8

Figura 1.1: Tareas

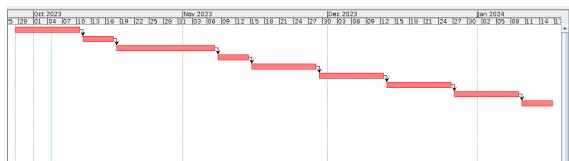


Figura 1.2: Gráfico Gantt

1.6.3. Hitos

- Finalización de la revisión bibliográfica - 10/10/2023
- Compleción de la preparación de datos - 17/10/2023
- Diseño e implementación modelo - 7/11/2023
- Validación inicial del modelo GAN - 14/11/2023
- Implementación completa del modelo híbrido GAN-Transformer - 12/12/2023
- Optimización y ajustes finales del modelo - 26/12/2023
- Redacción completa de la memoria del TFM - 9/1/2024
- Presentación y defensa del proyecto - 16/1/2024

1.7. Análisis de riesgos

- Complejidad técnica: La integración de GANs y Transformer podría presentar desafíos técnicos inesperados.
- Calidad de datos: Los datos multi-ómicos pueden requerir más pre-procesamiento o limpieza de lo esperado, lo cual puede retrasar la implementación del modelo.
- Tiempo insuficiente: Dada la amplitud del proyecto, podría haber riesgos de no cumplir con los plazos, especialmente si surgen problemas técnicos.
- Resultados insatisfactorios: Es posible que los primeros modelos no den los resultados esperados, requiriendo más tiempo de optimización.

- Cambios en el alcance: La adaptación o integración de nuevas técnicas o datos puede desviar el proyecto de su objetivo original.

1.8. Desviaciones del plan

Debido a limitaciones en la capacidad de cómputo, los modelos con datos reales no han podido ser explorados en profundidad. Es decir, se han usado los modelos con datos simulados y se les han aplicado las mejoras exclusivamente necesarias para hacerlos funcionar con los datos reales. Posiblemente podríamos tener mejores resultados si hubiéramos podido escalar y adecuar los modelos a la dimensionalidad de los datos reales.

El modelo de Transformer integrado en el generador de una GAN no dio los resultados esperados así que se optó por explorar otras opciones. Entre ellas, la seleccionada fue la de explorar la conversión de datos de expresión génica a metilación y viceversa usando las GANs y compararlo con los Transformers.

1.9. Breve sumario de productos obtenidos

- Plan de trabajo:

Un documento estructurado que detalla la planificación, metodología y enfoque del proyecto desde su inicio hasta su finalización.

- Memoria:

Un informe exhaustivo que abarca la revisión bibliográfica, la metodología adoptada, los experimentos realizados, los resultados obtenidos, las conclusiones y las recomendaciones para trabajos futuros.

- Producto:

Software desarrollado para la generación, integración y conversión de datos multi-ómicos.

- Presentación virtual:

Una presentación multimedia que resume el proyecto, sus objetivos, metodología, resultados y conclusiones, diseñada para ser presentada virtualmente ante un panel de expertos o interesados.

1.10. Breve descripción de los otros capítulos de la memoria

En el capítulo de Materiales y Métodos explicamos los enfoques y procedimientos utilizados en el trabajo. Sus principales apartados son:

- Investigación Teórica y Revisión Bibliográfica: En esta parte repasamos toda la investigación que se ha encontrado y lo que aporta al campo de la generación de datos multiómicos y de las GANs y Transformers
- Entorno y Datos: En esta parte explicamos el entorno usado, los datos usados y el procesamiento de los datos realizado
- Generación de datos sintéticos multi-ómicos: En esta parte explicamos el método usado para la exploración, selección y validación de los modelos usados para la generación de datos sintéticos.
- Conversión entre datos ómicos: En esta parte explicamos el método usado para la exploración, selección y validación de los modelos usados para la conversión entre datos ómicos.

El capítulo de Resultados presenta los hallazgos y resultados de los modelos. Sus principales apartados son:

- Generación datos multi-ómicos: En esta parte se presentan los resultados de los modelos usados para la generación de datos.
- Conversión datos ómicos: En esta parte se presentan los resultados de los modelos usados para la conversión entre datos ómicos.

En el capítulo de Conclusiones y trabajos futuros resumimos los resultados, sacamos conclusiones sobre ellos, analizamos las limitaciones del trabajo y se sugieren direcciones para trabajos futuros basados en este.

Capítulo 2

Materiales y Métodos

2.1. Investigación Teórica y Revisión Bibliográfica

La relevancia de las Redes Generativas Antagónicas (GANs) en el campo de la biología computacional y la medicina de precisión ha sido significativamente reforzada por una serie de desarrollos y mejoras en su tecnología. El concepto fundacional de las GANs fue introducido por Goodfellow et al. [13], marcando un hito en el aprendizaje generativo profundo. Este trabajo inicial abrió la puerta a una nueva era de generación de datos sintéticos, proporcionando una metodología robusta para la síntesis de datos que es especialmente pertinente en campos donde los datos son escasos o sensibles, como en estudios genómicos y de expresión génica.

Posteriormente, Arjovsky et al. [5] y Arjovsky y Bottou [4] introdujeron mejoras significativas en la estabilidad del entrenamiento de las GANs a través del concepto de GANs de Wasserstein, abordando desafíos críticos como el colapso de modos y proporcionando una métrica más significativa para la convergencia del modelo.

Gulrajani et al. [16] mejoraron la calidad de la generación y de la convergencia introduciendo el concepto de la penalización de gradiente de gradiente en las GANs de Wasserstein.

La contribución de Karras et al. [21] con las GANs de crecimiento progresivo y la de Brock et al. [7] con GANs a gran escala han demostrado cómo la calidad y variabilidad de la generación de imágenes puede ser mejorada significativamente, lo que tiene implicaciones directas en la generación de datos biomédicos complejos y detallados.

La normalización espectral, propuesta por Miyato et al. [28], representó otro avance en la estabilización del entrenamiento de las GANs, lo que es crucial para asegurar la fiabilidad de los datos generados en aplicaciones sensibles como el diseño de fármacos y el modelado de enfermedades. Lucic et al. [25] proporcionaron un análisis comparativo exhaustivo de diferentes arquitecturas de GANs, ofreciendo una guía valiosa para seleccionar la arquitectura más adecuada dependiendo de la aplicación específica en biología computacional.

Las técnicas para mejorar la formación de GANs, como las propuestas por Salimans et al.

[32] y Mescheder et al. [27], han permitido avances en la generación de datos más consistente y diversa, un aspecto crítico cuando se trata de simular variaciones biológicas reales. Heusel et al. [18] proporcionaron una regla de actualización de dos escalas temporales para el entrenamiento de GANs, lo que facilita la convergencia a un equilibrio de Nash local, un paso crucial para la generación de datos sintéticos en investigación biomédica.

Los trabajos de Chong y Forsyth [10] y Borji [6] han contribuido al entendimiento de las aplicaciones prácticas y las limitaciones de las GANs, proporcionando una perspectiva crítica sobre sus capacidades y desafíos en contextos de vida real. Finalmente, Guan y Loew [15] propusieron una nueva medida para evaluar GANs basada en el análisis directo de imágenes generadas, un enfoque que podría adaptarse para evaluar la calidad y utilidad de datos sintéticos ómicos generados mediante GANs.

La integración de datos multi-ómicos mediante Redes Generativas Antagónicas (GANs) representa un avance significativo en biología computacional y medicina de precisión. Ahmed et al. [1] exploraron el uso de GANs para integrar datos ómicos y redes de interacción, demostrando su capacidad para mejorar la clasificación y predicción en cáncer. De manera similar, Moon y Lee [29] propusieron un algoritmo de atención multitarea para la interpretación de datos multi-ómicos, destacando la importancia de procesos biológicos clave para el rendimiento diagnóstico.

Leng et al. [23] realizaron un estudio comparativo sobre métodos de aprendizaje profundo para la fusión de datos multi-ómicos en cáncer, enfatizando la importancia de combinar múltiples tipos de datos para comprender procesos biológicos complejos. Shi et al. [34] presentaron una GAN basada en módulos de atención para la predicción de resultados pronósticos en cáncer, destacando la efectividad de las GANs en la descripción de información heterogénea de múltiples modalidades de datos.

Brombacher et al. [8] investigaron el rendimiento de modelos generativos profundos en el aprendizaje de incrustaciones conjuntas de datos ómicos de células individuales, ilustrando la utilidad de los modelos generativos en la superación de limitaciones éticas y logísticas en la recolección de datos de expresión génica. Lee [22] proporcionó una revisión exhaustiva de los avances recientes en GANs aplicadas a datos de expresión génica, enfatizando su capacidad para generar datos sintéticos y abordar limitaciones de los datos disponibles.

Dubey y Singh [11] realizaron una encuesta exhaustiva sobre GANs basadas en transformadores en visión por computadora, proporcionando una perspectiva sobre las aplicaciones potenciales de estas técnicas en el análisis de datos ómicos. Choi y Lee [9] revisaron exhaustivamente la aplicación de arquitecturas basadas en transformadores y mecanismos de atención en el análisis de datos genómicos, resaltando su aplicabilidad y potencial en bioinformática.

Subramanian et al. [35] discutieron la integración e interpretación de datos multi-ómicos y su aplicación en biología, resaltando la importancia de las herramientas y métodos integrativos en este campo. La importancia de la cooperación entre biólogos y científicos de la computación en el trabajo con datos ómicos fue destacada por Poinsignon et al. [31].

Figueira y Vaz [12] proporcionaron una visión general de la generación de datos sintéticos y métodos de evaluación de GANs, lo que es fundamental para entender la aplicación de estas

técnicas en datos ómicos. Xu et al. [38] realizaron un estudio empírico sobre la adopción de transformadores en GANs, lo que podría informar sobre su uso en el análisis de datos ómicos.

Li et al. [24] presentaron una GAN basada en transformadores para la generación de datos de series temporales biomédicas, un enfoque relevante para el análisis de datos ómicos. Hess et al. [17] exploraron el uso de modelos log-lineales en combinación con técnicas de aprendizaje profundo generativo para extraer patrones de datos ómicos.

Jiang et al. [20] desarrollaron TLSurv, una red super híbrida para la integración de múltiples tipos de datos ómicos para la predicción de la supervivencia en cáncer, demostrando la utilidad de la integración de datos multi-ómicos. Viñas et al. [37] desarrollaron un método basado en GAN para generar datos de expresión génica realistas, subrayando la importancia de generar datos sintéticos que conserven propiedades específicas del tejido y el tipo de cáncer.

Hudson y Zitnick [19] introdujeron el GANformer, un tipo eficiente de Transformer aplicado al modelado generativo visual, que podría inspirar enfoques similares en el análisis de datos ómicos. La relevancia de las GANs en el campo de la biología computacional y la medicina de precisión fue resaltada por Goodfellow et al. [13], Arjovsky et al. [5], Karras et al. [21], Brock et al. [7], Miyato et al. [28], Lucic et al. [25], Arjovsky y Bottou [4], Salimans et al. [32], Mescheder et al. [27], Heusel et al. [18], Chong y Forsyth [10], Borji [6] y Guan y Loew [15], cuyos trabajos han contribuido significativamente al desarrollo y comprensión de las GANs.

En conjunto, estos estudios proporcionan una amplia perspectiva sobre el desarrollo y la aplicación de las GANs y otros enfoques de aprendizaje profundo en la integración y análisis de datos multi-ómicos. Estas investigaciones ofrecen un marco teórico y práctico adecuado para la realización de este trabajo.

2.2. Introducción a las arquitecturas usadas

2.2.1. Generative Adversarial Network

Una Red Generativa Antagónica (GAN) es un tipo de modelo de aprendizaje automático compuesto por dos redes neuronales que se entrena simultáneamente mediante un proceso competitivo. Las dos redes neuronales son:

- Generador: es la red neuronal que aprende a crear datos que imitan algún conjunto de datos real. No ve los datos reales durante el entrenamiento, sino que aprende a generar datos a partir de ruido aleatorio que se le proporciona como entrada. Su objetivo es producir datos falsos que sean indistinguibles de los verdaderos.
- Discriminador: es la red neuronal que se entrena para diferenciar entre los datos reales y los datos generados por el Generador. Recibe tanto muestras reales del conjunto de datos como muestras falsas del Generador y debe aprender a identificar correctamente cuáles son reales y cuáles son falsas.

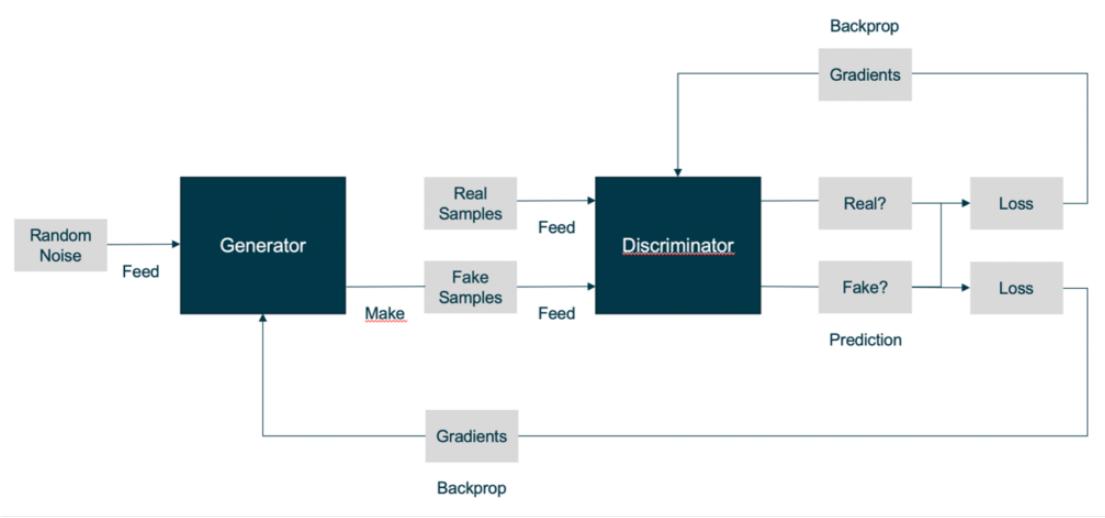


Figura 2.1: Descripción de la Arquitectura de la GAN. Imagen obtenida de [3]

El proceso de entrenamiento es el siguiente:

- Entrenar el Discriminador:
 - Se toma un lote de muestras reales del conjunto de datos de entrenamiento
 - Se genera un lote de muestras falsas utilizando el Generador, que toma ruido aleatorio como entrada
 - Se alimenta al Discriminador con ambos lotes (reales y falsos)
 - Se calcula la perdida del Discriminador basándose en qué tan bien clasifica las muestras reales como reales y las generadas por el Generador como falsas.
- Entrenar el Generador:
 - Se genera otro lote de muestras falsas utilizando el Generador.
 - Estas muestras generadas se pasan al Discriminador
 - Se calcula la pérdida del Generador basándose en qué tan bien las muestras falsas engañaron al Discriminador

Una vez finalizado el entrenamiento, se puede usar el Generador solamente para, a partir de ruido aleatorio como entrada, generar los datos que haya aprendido durante el entrenamiento.

Las ventajas y desventajas de esta clase de modelos son:

Ventajas:

1. Calidad de Generación: Capacidad para generar datos altamente realistas, especialmente imágenes.
2. Aprendizaje Sin Supervisión: Capaces de aprender a generar datos sin etiquetado.

3. Innovación en Aplicaciones Creativas: Impulsan innovaciones en áreas como arte, moda y diseño.

Desventajas:

1. Inestabilidad de Entrenamiento: Difíciles de entrenar y propensos a la falta de convergencia.
2. Colapso de Modos: Riesgo de producir un rango limitado de salidas, reduciendo la diversidad.
3. Requisito de Recursos: Necesitan una cantidad significativa de recursos computacionales y datos.

El colapso de modos es uno de los problemas con el que nos encontramos durante la realización de este trabajo.

Este fenómeno ocurre cuando el generador comienza a producir un rango limitado de salidas, ignorando gran parte de la variabilidad de los datos reales. Esto se debe a la naturaleza del entrenamiento de las GANs, donde el generador puede encontrar un "punto dulce" para engañar al Discriminador sin necesidad de explorar toda la distribución de los datos reales.

Por ello, procedimos a usar tipos de GAN que evitaran este colapso. El primer tipo fue la WGAN.

La WGAN introduce una nueva función de pérdida basada en la distancia de Wasserstein (también conocida como distancia de la tierra), que proporciona un entrenamiento más estable y evita el problema del colapso de modo.

Distancia de Wasserstein

La distancia de Wasserstein es una medida que cuantifica la "distancia" entre dos distribuciones de probabilidad. Su fórmula es:

$$W_p(\mu, \nu) = \left(\inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} d(x, y)^p d\gamma(x, y) \right)^{\frac{1}{p}}$$

Donde:

- $W_p(\mu, \nu)$ es la distancia de Wasserstein de orden p entre μ y ν .
- $\Gamma(\mu, \nu)$ denota el conjunto de todas las medidas de transporte (o medidas de acoplamiento) entre μ y ν , es decir, todas las medidas de probabilidad en $M \times M$ cuyos márgenes son μ y ν respectivamente.
- $d(x, y)$ es la distancia en el espacio métrico M entre los elementos x y y .
- El operador \inf representa el ínfimo (o el mínimo) sobre el conjunto $\Gamma(\mu, \nu)$.

Matemáticamente, la distancia de Wasserstein se define como el costo mínimo de transporte de la masa de una distribución a otra, considerando una cierta función de costo de transporte (como la distancia euclíadiana).

Función de pérdida de Wasserstein

Esta función de pérdida depende de una modificación del esquema GAN (llamado “GAN de Wasserstein” o “WGAN”) en el que el discriminador no clasifica realmente las instancias. Para cada instancia, devuelve un número. Este número no tiene por qué ser menor que uno ni mayor que cero, por lo que no podemos usar 0.5 como umbral para decidir si una instancia es real o falsa. El entrenamiento del discriminador solo intenta hacer que la salida sea mayor para las instancias reales que para las falsas.

Debido a que realmente no puede discriminar entre lo real y lo falso, al discriminador de WGAN se le llama “crítico” en lugar de “discriminador”. Esta distinción tiene importancia teórica, pero para fines prácticos podemos tratarlo como un reconocimiento de que las entradas a las funciones de pérdida no tienen que ser probabilidades.

Las funciones de pérdida en sí mismas son engañosamente simples:

Pérdida del Crítico: $D(x) - D(G(z))$

El discriminador intenta maximizar esta función. En otras palabras, intenta maximizar la diferencia entre su salida en instancias reales y su salida en instancias falsas.

Pérdida del Generador: $D(G(z))$

El generador intenta maximizar esta función. En otras palabras, intenta maximizar la salida del discriminador para sus instancias falsas.

En estas funciones:

- $D(x)$ es la salida del crítico para una instancia real.
- $G(z)$ es la salida del generador cuando se le da ruido z .
- $D(G(z))$ es la salida del crítico para una instancia falsa.
- La salida del crítico D no tiene por qué estar entre 1 y 0.
- Las fórmulas se derivan de la distancia del movimiento de tierras entre las distribuciones reales y generadas.

[14]

La función de perdida de Wasserstein es una medida más efectiva para calcular la diferencia entre la distribución de los datos generados y los reales, lo que conduce a un aprendizaje más coherente y estable del generador.

En la WGAN, además, se requiere que el Discriminador (llamado crítico en este contexto) sea 1-Lipschitz.

Condición de Lipschitz

Una función 1-Lipschitz es un tipo especial de función que satisface una condición específica sobre la forma en que controla sus variaciones. Esta condición, conocida como la condición de Lipschitz, es una forma de cuantificar cuánto puede cambiar una función en respuesta a cambios en sus entradas.

Una función $f : X \rightarrow Y$ entre dos espacios métricos X y Y se dice que es 1-Lipschitz si existe una constante L (en este caso $L = 1$) tal que para todos los puntos x_1 y x_2 en X , la siguiente desigualdad se mantiene:

$$|f(x_1) - f(x_2)| \leq L \cdot |x_1 - x_2|$$

Cuando $L = 1$, esto se simplifica a:

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|$$

Esto significa que el cambio absoluto en la función f entre dos puntos cualesquiera no es mayor que el cambio absoluto entre esos puntos en el dominio de la función.

Una función 1-Lipschitz puede considerarse como una función que “no se mueve demasiado rápido”. No importa qué dos puntos elijas en su dominio, el ritmo al que la función cambia entre esos puntos está limitado. Esto es análogo a decir que la “pendiente” (en términos de valor absoluto) de la función nunca excede 1.

Para conseguir que el Discriminador sea 1-Lipschitz se logra con el recorte de pesos del Discriminador, lo que puede llevar a problemas de convergencia y a un entrenamiento inestable.

Recorte de pesos

El recorte de pesos consiste en que después de cada actualización de los parámetros del Discriminador durante el entrenamiento, los pesos se ”recortan” o se limitan a permanecer dentro de un rango predefinido, como $[-c, c]$, donde c es una constante elegida.

Dado que la WGAN nos seguía dando el colapso de modos el siguiente tipo de GAN que probamos fue la WGAN-GP.

La WGAN-GP mejora aún más la WGAN original al introducir una penalización de gradiente (GP, por sus siglas en inglés) para asegurar la restricción 1-Lipschitz de una manera más efectiva y estable.

Esta técnica funciona penalizando el modelo si el gradiente del Discriminador con respecto a sus entradas no cumple con la condición 1-Lipschitz.

Funcionamiento de la penalización de gradiente:

1. Se interpolan muestras entre datos reales y generados.
2. Se calcula el gradiente del Discriminador con respecto a estas muestras interpoladas.
3. Se aplica una penalización de gradiente si la norma euclídea del gradiente se desvía de 1, lo que indica una violación de la condición 1-Lipschitz.

La fórmula para la penalización de gradiente es:

$$\mathcal{L}_{GP} = \lambda \cdot \mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]$$

donde

- \mathcal{L}_{GP} es la pérdida por penalización de gradiente
- λ es un hiperparámetro que controla la intensidad de la penalización de gradiente
- \mathbb{E} representa el valor esperado sobre las muestras interpoladas
- $\hat{x}, \mathbb{P}_{\hat{x}}$ es la distribución de estas muestras
- $\nabla_{\hat{x}}D(\hat{x})$ es el gradiente del Discriminador con respecto a \hat{x}
- $\|\cdot\|_2$ denota la norma euclidiana

Esta técnica asegura que el crítico cumpla la restricción de Lipschitz de una manera más efectiva y suave, lo que lleva a un entrenamiento más estable y a una mejora en la calidad de los datos generados.

Para que esta técnica funcione eficazmente, es común entrenar más veces al discriminador por cada actualización del generador. Esta práctica, conocida como 'training ratio' o 'critic-to-generator ratio', ayuda a asegurar que el discriminador sea lo suficientemente bueno para guiar al generador hacia una mejor generación de datos.

2.2.2. Variational AutoEncoder

Un VAE, o Autoencoder Variacional, es un tipo de modelo de aprendizaje automático que se utiliza principalmente para el aprendizaje no supervisado de representaciones complejas. A diferencia de un autoencoder tradicional, un VAE introduce aleatoriedad en el proceso. En lugar de codificar una entrada como un único punto en el espacio latente, el VAE codifica la entrada como una distribución de probabilidad sobre el espacio latente.

Consta de dos partes:

- Codificador: Toma datos de entrada y los transforma en una representación de menor dimensión llamada espacio latente.
- Decodificador: Toma la representación comprimida en el espacio latente y trata de reconstruir los datos originales lo más fielmente posible.

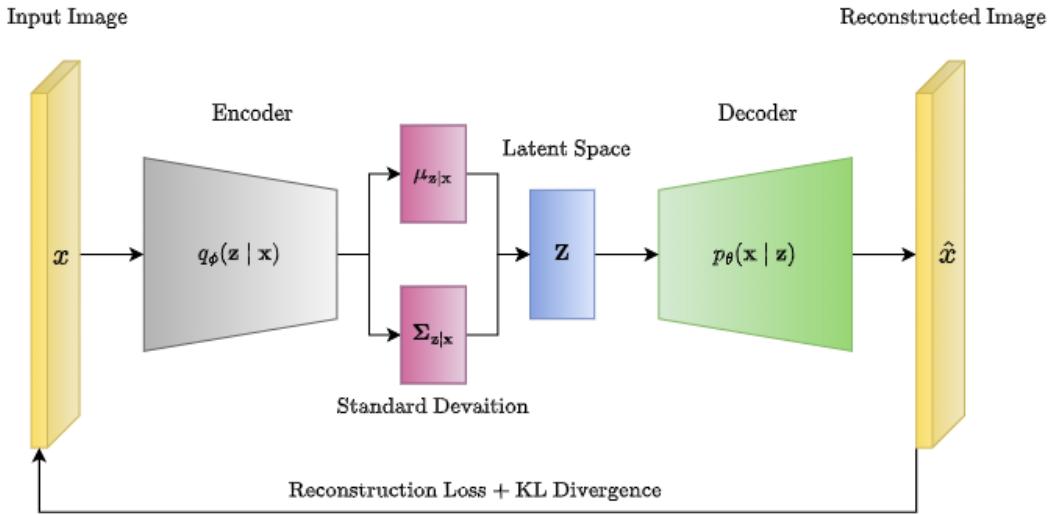


Figura 2.2: Descripción de la Arquitectura del VAE. Imagen obtenida de [33]

Durante la codificación, el VAE aprende dos cosas para cada dato de entrada:

- Una media (μ)
- Una desviación estándar (σ)

Estos parámetros definen una distribución gaussiana en el espacio latente desde la cual se puede muestrear para generar nuevas entradas.

Durante el entrenamiento, el VAE intenta minimizar la diferencia entre los datos de entrada originales y sus reconstrucciones, conocida como pérdida de reconstrucción.

En nuestro caso concreto se usa la entropía cruzada binaria que está definida como:

$$\text{Reconstruction Loss} = - \sum_i [x_i \log(\hat{x}_i) + (1 - x_i) \log(1 - \hat{x}_i)]$$

Además, para asegurarse de que el espacio latente tenga buenas propiedades (continuidad, por ejemplo), se utiliza una pérdida adicional conocida como pérdida de Kullback-Leibler (KL) para medir cuán eficientemente se está utilizando el espacio latente.

La pérdida de Kullback-Leibler está definida como:

$$\text{KL Divergence} = -\frac{1}{2} \sum_j (1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2)$$

Una vez finalizado el entrenamiento se puede usar solamente el Decoder para generar los datos del espacio latente.

Las ventajas y desventajas de esta clase de modelos son:

Ventajas:

1. Estructura del Espacio latente: Aprenden espacios latentes bien estructurados y continuos.
2. Estabilidad de Entrenamiento: Más estables y fáciles de entrenar que las GANs.
3. Aplicaciones en Aprendizaje No Supervisado: Efectivos en aprendizaje no supervisado y reducción de dimensionalidad.

Desventajas:

1. Calidad de la Generación: Menos nítidos y realistas en comparación con GANs.
2. Aproximación Variacional: Puede no capturar completamente la verdadera distribución de los datos.
3. Limitaciones en la Capacidad de Modelado: Limitaciones en modelar distribuciones de datos altamente complejas.

2.2.3. Transformer

El **Transformer** es un modelo de aprendizaje profundo introducido por Vaswani et al. en su influyente trabajo “Attention Is All You Need” [36].

Esta arquitectura ha sido fundamental en el avance del procesamiento del lenguaje natural (NLP), superando a los modelos anteriores basados en redes neuronales recurrentes (RNN) y convolucionales (CNN).

Su característica distintiva es el uso exclusivo de mecanismos de atención, lo que elimina la necesidad de secuencialidad en el procesamiento de los datos, permitiendo un procesamiento paralelo más eficiente y una captura más efectiva de dependencias a largo plazo en los datos.

Funcionamiento del Mecanismo de Atención

El mecanismo de atención se basa en los conceptos de ”query” (consulta), ”key” (clave) y ”value” (valor). Estos términos se utilizan para describir cómo se extrae y se presta atención a la información relevante de una secuencia de entrada.

Query representa el elemento actual que estamos tratando de analizar o para el cual estamos tratando de determinar la relevancia de otros elementos en la secuencia.

Por ejemplo, en una oración, si estamos tratando de entender el contexto de una palabra específica, esa palabra sería la "query".

La Key representa los elementos en la secuencia a los que queremos prestar atención.

En el ejemplo de la oración, cada palabra en la oración (o en un contexto más amplio) podría ser una "key".

El Value representa la relevancia de las Keys con respecto a las Queries.

El proceso de atención funciona de la siguiente manera:

Para cada "query", se calcula un puntaje de atención comparándola con todas las "keys". Este puntaje determina cuánta atención se debe prestar a cada "key" en relación con la "query".

Los puntajes de atención se normalizan (generalmente usando la función softmax) para que sumen 1, convirtiéndolos en una especie de pesos.

Estos pesos se utilizan luego para crear una combinación ponderada de los "values". Esta combinación ponderada es esencialmente la salida del mecanismo de atención para la "query" dada y representa una agregación de la información relevante de toda la secuencia.

Mecanismo de Atención de Múltiples Cabezas

El mecanismo de atención de múltiples cabezas es un componente clave en la arquitectura Transformer.

La idea detrás de la atención de múltiples cabezas es que, en lugar de tener una sola cabeza de atención que procesa la secuencia completa, el modelo tiene múltiples cabezas de atención que trabajan en paralelo.

Cada cabeza de atención realiza el mismo proceso de atención básico, pero con diferentes conjuntos de parámetros. Esto permite que cada cabeza se enfoque en diferentes aspectos o partes de la secuencia de entrada.

Para su funcionamiento primero la consulta, clave y valor (Q , K , V) se dividen en múltiples conjuntos más pequeños. Cada conjunto corresponde a una cabeza de atención.

Cada cabeza calcula su propio conjunto de pesos de atención y una combinación ponderada de values, independientemente de las otras cabezas.

Los resultados de todas las cabezas se concatenan y luego se transforman mediante una operación lineal para producir el resultado final de la capa de atención de múltiples cabezas.

Tipos de Atención

Hay dos tipos principales de atención en el modelo Transformer:

- **Self-Attention:** las queries, keys y values provienen de la misma secuencia de entrada.
- **Encoder-Decoder Attention:** las queries provienen del decoder (la parte del modelo que genera la salida), mientras que las keys y values provienen del encoder (la parte del modelo que procesa la entrada).

Arquitectura del Transformer

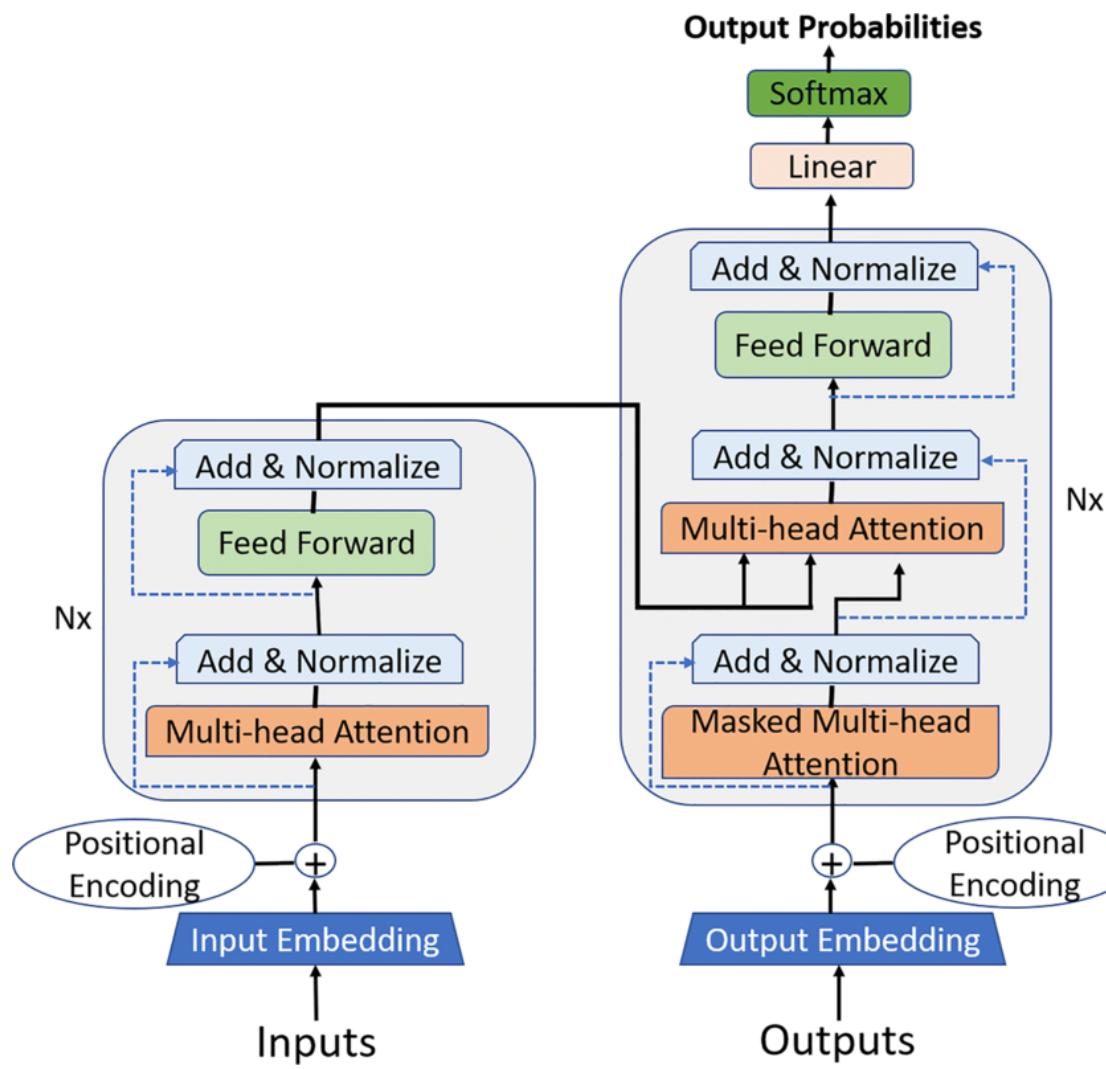


Figura 2.3: Descripción de la Arquitectura del Transformer. Imagen obtenida de [30]

El modelo Transformer consta de dos componentes principales: el **Encoder** y el **Decoder**.

Encoder

El encoder del Transformer tiene una serie de capas idénticas. Cada capa contiene dos subcapas:

- **Atención de Múltiples Cabezas:** Permite al modelo enfocarse en diferentes partes de la secuencia de entrada. Esto es crucial para capturar distintos aspectos contextuales y semánticos.
- **Red Neuronal Feed-Forward:** Una red completamente conectada que se aplica de manera idéntica a cada posición de la secuencia, proporcionando una transformación adicional.

Cada subcapa en el encoder, así como en el decoder, está seguida de una conexión residual y

una normalización de capa. Estas conexiones residuales ayudan a evitar el desvanecimiento del gradiente en redes profundas.

Decoder

El decoder también está compuesto por capas idénticas, cada una con tres subcapas:

- **Atención de Múltiples Cabezas:** Similar a la del encoder, pero se aplica a la secuencia de salida.
- **Atención Encoder-Decoder:** Permite a cada posición en el decoder enfocarse en todas las posiciones del texto de entrada, utilizando la salida del encoder como guía. Esto es especialmente relevante para tareas como la traducción automática.
- **Red Neuronal Feed-Forward:** Al igual que en el encoder, transforma las representaciones secuenciales.

Input Embedding

Los modelos Transformer no pueden procesar directamente texto crudo. Primero, el texto se divide en unidades más pequeñas llamadas "tokens" (como palabras o subpalabras).

Cada token se mapea a un vector numérico fijo mediante una tabla de embedding. Esta tabla contiene un vector único para cada token posible en el vocabulario del modelo.

Los vectores resultantes son representaciones densas. A diferencia de las representaciones dispersas (como one-hot encoding), donde la mayoría de los elementos son cero excepto uno, las representaciones densas utilizan todos los elementos del vector para codificar información.

Estos vectores de embedding están diseñados para capturar significados semánticos y relaciones sintácticas. Palabras o tokens con significados o usos similares tienden a tener vectores de embedding que son cercanos en el espacio vectorial.

Durante el entrenamiento del modelo, estos embeddings se ajustan para reflejar mejor las relaciones entre palabras en el contexto del corpus de entrenamiento.

Positional Encoding

Además, los Transformers agregan un "positional encoding"^a los embeddings de entrada para incorporar información sobre la posición relativa de los tokens en la secuencia.

Este paso es crucial porque los modelos Transformer, a diferencia de las RNN o LSTM, no tienen un sentido intrínseco del orden secuencial debido a su naturaleza paralela.

En el Transformer, el positional encoding se implementa mediante una fórmula matemática con senos y cosenos específicos que genera un vector único para cada posición posible en una secuencia.

Las frecuencias de las ondas seno y coseno varían según la dimensión del embedding. Esto permite que cada dimensión del vector de positional encoding capture información de posición en diferentes escalas temporales.

Los vectores de positional encoding se suman a los vectores de embedding de tokens. Esto significa que la representación final de cada token contiene tanto información semántica (del embedding) como información posicional.

Las ventajas y desventajas de esta clase de modelos son:

Ventajas:

1. Eficiencia en el Procesamiento Paralelo: Permite un procesamiento altamente paralelo.
2. Capacidad para Capturar Contexto a Larga Distancia: Manejan dependencias a larga distancia eficazmente.
3. Versatilidad y Escalabilidad: Base de modelos de lenguaje de última generación como BERT y GPT.

Desventajas:

1. Requisitos de Datos y Recursos: Necesitan grandes cantidades de datos y recursos computacionales.
2. Complejidad y Comprensión: Difícil interpretación y comprensión de sus mecanismos de atención.
3. Sobreadaptación a Datos de Entrenamiento: Riesgo de sobreadaptación a conjuntos de datos de entrenamiento.

En este trabajo hemos optado por usar una arquitectura de Transformer Encoder-Only porque daba mejores resultados con nuestros datos que no son secuenciales. Además no ha sido necesario el uso de Input Embedding ni Positional Encoding.

2.3. Entorno y Datos

2.3.1. Herramientas de Software y Entorno de Trabajo

El trabajo se llevó a cabo utilizando el lenguaje de programación Python, con librerías especializadas como *PyTorch*, *Pandas*, *NumPy*, *Matplotlib*, *Seaborn*, *Scikit-learn* y *SciPy*. Se empleó un entorno de cómputo con soporte para CUDA(versión 11.8), facilitando el entrenamiento de modelos en GPU.

La lista de requisitos de las versiones del entorno Python se puede encontrar en el Anexo B [5.1](#)

El proyecto se realizó en Jupyter Notebook en un portátil con las siguientes especificaciones:

- Procesador Intel i7-11800H
- Memoria RAM 64GB
- GPU Nvidia Geforce RTX 3070 Laptop 8GB VRAM
- Sistema Operativo Debian 12

2.3.2. Generación de datos simulados

En este estudio, se utilizó el paquete **InterSIM** en R para la simulación de conjuntos de datos genómicos interrelacionados.

InterSIM es un paquete de R que genera tres conjuntos de datos interrelacionados con relaciones inter e intrarrealistas basadas en la metilación del ADN, la expresión del ARNm y la expresión de proteínas del estudio de cáncer de ovario TCGA.

Se puede acceder al manual del paquete en la siguiente dirección: <https://cran.r-project.org/web/packages/InterSIM/InterSIM.pdf>

Al generar los datos simulados nos interesaba tener dos grupos, en el que uno fueran los pacientes sanos y el otro los enfermos. Se usó un desplazamiento medio de 5 entre los grupos y queríamos conservar las estructuras de covarianza de los datos del estudio original a partir del que se generaron.

Los datos fueron generados utilizando la función `InterSIM()` del paquete, configurada con los siguientes parámetros:

- Número de muestras (`n.sample`): Se generaron conjuntos con 500, 3000, 10000 y 20000 muestras.
- Proporción de muestras en los clústeres (`cluster.sample.prop`): `c(0.8, 0.2)`.
- Desplazamiento medio del clúster para los datos de metilación, expresión génica y expresión proteica (`delta.methyl`, `delta.expr`, `delta.protein`): 5.
- Proporción de CpGs diferencialmente expresados (`p.DMP`): 0.2.
- Las estructuras de covarianza para los datos de metilación, expresión génica y expresión proteica se dejaron los valores predeterminados que tiene la función.
- No se generaron mapas de calor (`do.plot`): FALSE.

```
1 library(InterSIM)
2 prop <- c(0.8, 0.2)
3 effect <- 5
4 sim.data <- InterSIM(n.sample = 20000, cluster.sample.prop = prop,
5 delta.methyl = effect, delta.expr = effect, delta.protein = effect,
6 p.DMP = 0.2, p.DEG = NULL, p.DEP = NULL,
7 sigma.methyl = NULL, sigma.expr = NULL, sigma.protein = NULL,
8 cor.methyl.expr = NULL, cor.expr.protein = NULL,
9 do.plot = FALSE)
10 sim.methyl <- sim.data$dat.methyl
11 sim.expr <- sim.data$dat.expr
12 #sim.protein <- sim.data$dat.protein
13 sim.assignment <- sim.data$clustering.assignment
14 write.csv(sim.methyl, "/home/fede/Documents/BIOINFO/UOC/TFM/methyl20K.
15   csv")
16 write.csv(sim.expr, "/home/fede/Documents/BIOINFO/UOC/TFM/expr20K.csv")
```

```
16     write.csv(sim.assignment, "/home/fede/Documents/BIOINFO/UOC/TFM/assign20K  
 .csv")
```

Listing 2.1: Código para la generación de datos usando InterSIM

Cada dataset consiste en un fichero expr.csv (con los datos de expresión génica, con 131 variables), un fichero methyl.csv (con los datos de metilación, 367 variables) y un fichero assig.csv (con las asignaciones de cada sample a un cluster)

Este numero de variables es fijo y proviene del estudio original a partir del cual se diseñó la herramienta.

2.3.3. Preprocesamiento de datos simulados

Los pasos para el preprocesamiento fueron:

- **Lectura y Limpieza:** Los datos fueron leídos usando la biblioteca *pandas* en Python, eliminando las primeras columnas no relevantes.
- **Conversión a float:** Todas las columnas se convirtieron a tipo flotante.
- **Normalización:** Se aplicó una normalización usando *MinMaxScaler* de *sklearn*, escalaendo todos los valores al rango de 0 a 1.
- **Conversión a tensor:** Se convirtieron los dataframes a tensores y se pasaran a la gpu.

El código usado para la lectura y preprocesamiento se puede consultar en [5.3](#)

Debajo podemos ver las matrices de matrices de correlación de los datos

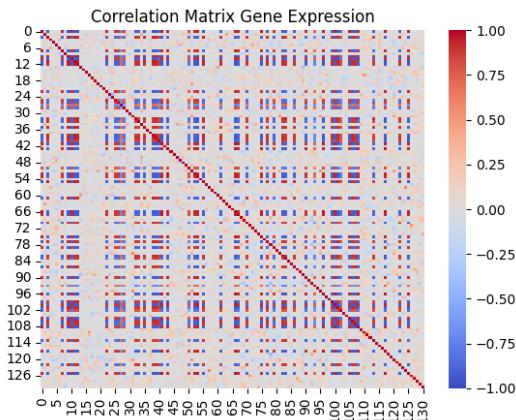


Figura 2.4: Matriz de correlación de expresión génica

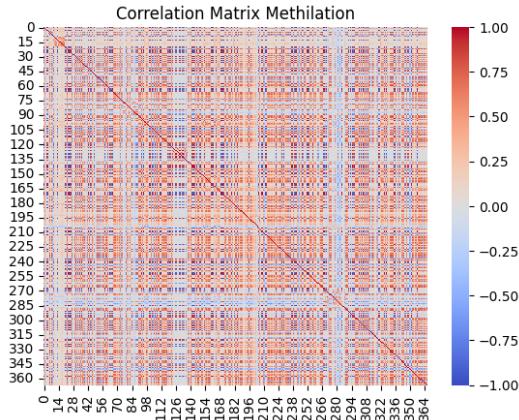


Figura 2.5: Matriz de correlación de metilación

Se puede consultar un análisis estadístico más exhaustivo en [5.2](#)

2.3.4. Obtención de datos reales

Los datos reales los hemos obtenido de <https://adex.genyo.es/>.

Es una base de datos que integra 82 estudios curados de transcriptómica y metilación, abarcando 5609 muestras para algunas de las enfermedades autoinmunes más comunes. La base de datos proporciona, en un entorno fácil de usar, métodos avanzados de análisis de datos y estadísticos para explorar conjuntos de datos ómicos, incluyendo metaanálisis, expresión diferencial o análisis de vías. Martorell-Marugán et al. [26]

De esta base de datos hemos descargado los datos de expresión y metilación de la serie GSE117931 <https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE117931>

En este estudio la integración del metiloma completo del ADN y la transcripción reveló genes y vías regulados por metilación anómalos en las células mononucleares de sangre periférica de la esclerosis sistémica.

El dataset tiene 37 muestras y consiste en 3 archivos:

- GSE117931_GPL14951.tsv: Los datos de expresión con 14760 variables
- GSE117931_GPL13534.tsv: Los datos de metilación con 420643 variables
- metadata.tsv: Con los metadatos de las muestras

De las 37 muestras, 19 corresponden a individuos sanos y 18 a individuos con esclerosis sistémica.

2.3.5. Preprocesamiento de datos reales

Los pasos para el preprocesamiento fueron:

- **Creación de fichero de asignación a grupos:** Se creó el fichero de asignación de los samples a grupos a partir del metadata.tsv
- **Lectura y Limpieza:** Los datos fueron leídos usando la biblioteca *pandas* en Python, eliminando las primeras columnas no relevantes.
- **Tratamiento de NAs:** Se quitaron todas las columnas de los datos de metilación que contenían NAs
- **Conversión a float:** Todas las columnas se convirtieron a tipo flotante.
- **Normalización:** Se aplicó una normalización usando *MinMaxScaler* de *sklearn*, escalando todos los valores al rango de 0 a 1.
- **Conversión a tensor:** Se convirtieron los dataframes a tensores y se pasaran a la gpu.

Tras el procesamiento los datos quedan así:

- 37 muestras, 19 correspondientes a individuos sanos, 18 a enfermos.
- Datos de expresión génica con 14760 variables
- Datos de metilación con 416660 variables

El código usado para la lectura y preprocesamiento se puede consultar en [5.3](#)

Deabajo podemos ver las matrices de correlación de los datos

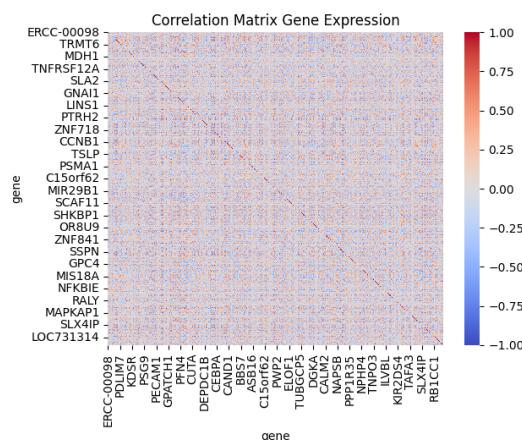


Figura 2.6: Matriz de correlación de expresión génica(solo se muestran etiquetas de algunas variables)

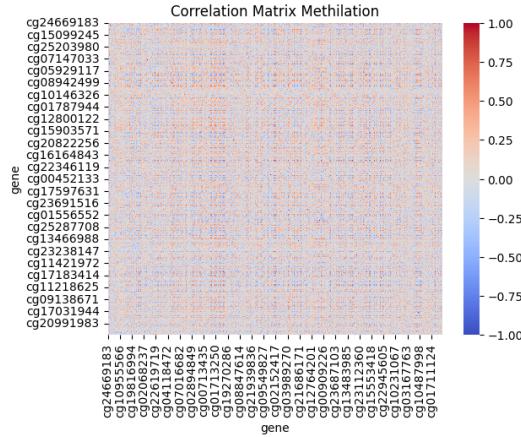


Figura 2.7: Matriz de correlación de metilación(solo se muestran etiquetas de algunas variables)

Se puede consultar un análisis estadístico más exhaustivo en [5.2](#)

También se han usado los datos reales seleccionando solo las variables cuya desviación estándar se encontraba en el percentil 90 % o superior.

Con este filtrado tenemos:

- 37 muestras, 19 correspondientes a individuos sanos, 18 a enfermos.
- Datos de expresión génica con 1476 variables
- Datos de metilación con 41666 variables

2.4. Generación de datos sintéticos multiómicos

En este apartado exploramos los modelos que generan datos sintéticos multiómicos. En concreto, durante el entrenamiento se les entrena con datos de expresión génica, de metilación y la asignación a un grupo(enfermo/sano) y una vez entrenados tienen que generar datos de expresión génica, metilación y la asignación al grupo correspondiente.

2.4.1. GAN

Exploración de modelos preliminares

En esta parte nos centramos en la exploración rápida de distintos modelos e hiperparámetros para seleccionar los mejores candidatos. Para ello usamos el conjunto de datos generado de 500 muestras y procedimos a probar distintas arquitecturas e hiperparámetros.

Entre las variables que se exploraron estaban:

- Número de capas ocultas: Se probaron 0,1,2,3,4 y 5
- Número de neuronas por capa: Se probaron 128,256,512,1024,2048,4096 y 8192

- Tasa de aprendizaje: Se probaron 0.2, 0.002, 0.0002, 0.00002 y 0.000002
- Optimizador: Se probó Adam y RMSprop
- Función de perdida: Se probó BCELoss, BCEWithLogitsLoss y MSELoss
- Tamaño de lote: Se probó 32, 64 y 128
- Número de épocas de entrenamiento: Se probó 5000, 10000, 15000 y 30000
- Tipo de normalización: Se probó sin normalización, StandardScaler y MinMaxScaler
- Coeficiente de penalización de gradiente: Se probó 5, 10 y 15
- Número de veces de actualización de Discriminador: Se probó 1, 2, 5 y 8

Dado la limitación en capacidad de cómputo y en tiempo de desarrollo se decidió usar una exploración aleatoria e intuitiva no completa. Para ello se usó el análisis de la evolución de los valores de las funciones de pérdida de Generador y Discriminador, la exploración visual de los datos generados y el conocimiento adquirido mediante el aprendizaje de las variaciones que sufrían los datos dependiendo de las variables exploradas.

Se generaron cerca de 200 modelos. Se puede acceder a algunos de ellos en(dentro de la carpeta oldmodels):

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfal3utRY1si?usp=sharing>

Para evaluar el rendimiento de los modelos, inicialmente se uso un clasificador SVC con los datos generados y se esperaba que la clasificación siguiera la misma proporción que en la clasificación de un SVC con los datos reales(simulados).

Un Support Vector Classifier (SVC), es una variante de las Support Vector Machines (SVM) enfocada en la clasificación. Es un modelo de aprendizaje supervisado utilizado para clasificar datos en dos o más grupos distintos. En un espacio N-dimensional, un SVC busca encontrar un hiperplano (una línea en 2D, un plano en 3D, etc.) que separe de la mejor manera posible las clases de datos.

En esta fase, rápidamente nos encontramos con el problema del colapso de modo. Haciendo que fuera realmente difícil encontrar la arquitectura correcta para que no ocurriera.

Dado esto, decidimos usar una arquitectura WGAN-GP (Wasserstein GAN con penalización de gradiente). Con esta arquitectura pudimos explorar con más facilidad modelos más potentes.

Selección de modelo final

Tras decidir usar la WGAN-GP, el optimizador RMSprop, la tasa de aprendizaje de 0.0002, el tamaño de lote de 64 y el número de épocas 10.000.

Procedimos a realizar diversos modelos WGAN-GP con distintos números de capas y neuronas. En este caso usamos los conjuntos de datos de 3000 y 10.000 muestras para entrenarlos y compararlos.

Para comparar estos modelos usamos distintas métricas, entre ellas:

- Distancia de Wasserstein: mide el costo mínimo para transformar una distribución probabilística en otra, basado en la distancia geodésica.
- Test Kolmogorov-Smirnov: es un procedimiento no paramétrico para evaluar la igualdad de distribuciones continuas basado en sus funciones de distribución acumulativa.
- Distancia euclíadiana promedio: calcula el promedio de todas las distancias euclidianas calculadas entre los pares de puntos reales y generados.
- Matrices de correlación de datos reales y generados y matriz de diferencia de correlación: medimos las relaciones entre las variables, buscamos que las matrices sean parecidas y que la diferencia sea mínima.
- Clasificación con SVC de datos reales y generados: medimos la capacidad del clasificador de distinguir entre datos reales y generados.
- Clasificación con SVC de datos generados: buscamos tener una distribución en clasificación parecida a la de clasificar los datos reales con el mismo clasificador.
- Dendogramas jerárquicos de datos reales y generados: buscamos la similitud en la estructura de los clústeres ya que indica que la GAN está aprendiendo correctamente la distribución de los datos reales.
- Clasificación con k-NN(K-Nearest Neighbours): Comparamos la clasificación de los datos reales y los generados con un clasificador k-NN
- t-SNE(t-Distributed Stochastic Neighbor Embedding): reduce la dimensionalidad manteniendo la similitud local entre los puntos. Ayuda a ver si la GAN está capturando la estructura de los datos reales y si existe colapso de modo.
- UMAP(Uniform Manifold Approximation and Projection): es similar a t-SNE en que busca preservar las estructuras locales, pero también tiene en cuenta la estructura global de los datos. Ayuda a ver si la GAN está capturando la estructura de los datos reales y si existe colapso de modo.
- PCA(Principal Components Analysis): reduce la dimensionalidad identificando las "direcciones"(componentes principales) que maximizan la varianza en el conjunto de datos. Útil para obtener una visión rápida y general de la distribución de los datos generados por la GAN en comparación con los datos reales.
- KDE(Kernel Density Estimation) de distintas variables al azar: es una técnica no paramétrica para estimar la función de densidad de probabilidad de una variable aleatoria. Utilizado para visualizar la distribución de los datos generados por una GAN y compararla con la distribución de los datos reales.

Al final, el modelo seleccionado fue el siguiente:

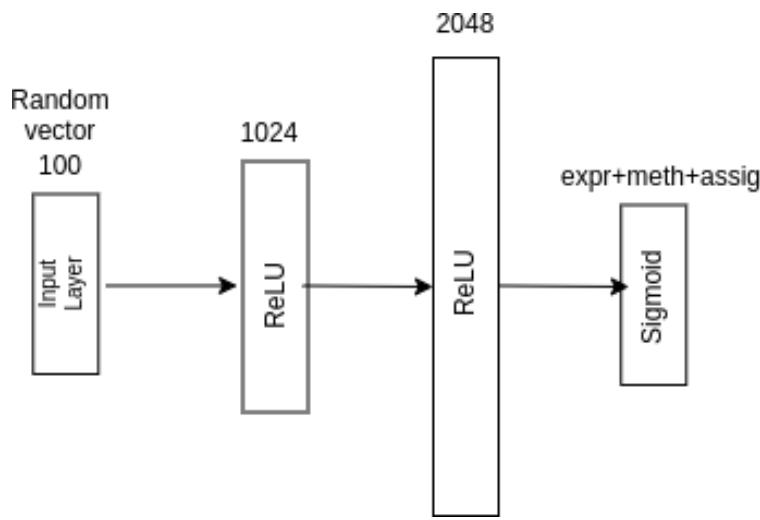


Figura 2.8: Generador Modelo Final GAN

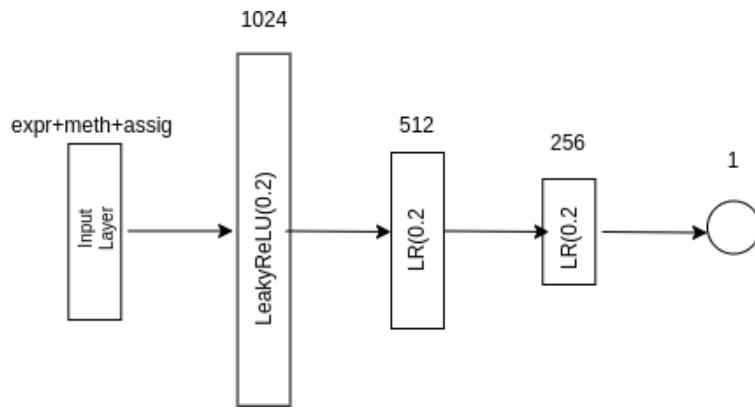


Figura 2.9: Discriminador Modelo Final GAN

Como podemos ver el Generador consta de 2 capas ocultas(una con 1024 neuronas y la otra con 2048). La capa de entrada es el vector de ruido aleatorio de tamaño 100 y la capa de salida tiene el tamaño del numero de variables total(expresión, metilación y asignación).

El Discriminador consta de 3 capas ocultas(con 1024, 512 y 256 neuronas). La capa de entrada tiene el tamaño del numero de variables total(expresión, metilación y asignación) y la capa de salida tiene 1 neurona que devuelve la puntuación de Wasserstein.

Los parámetros de entrenamiento fueron:

- Optimizador: RMSprop para Generador y Discriminador
- Learning rate = 0.0002 para ambos
- n_critic = 5 (número de veces que se actualiza el Discriminador)
- lambda_gp = 10 (Coeficiente de penalización de gradiente)

- n_epochs=10000 (número de épocas de entrenamiento)

Se puede acceder al código y resultados del modelo en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfal3utRY1si?usp=sharing>

Nombre de archivo: ModelFullWGANGPFINAL.ipynb

Ejemplo de salida del entrenamiento:

```
El modelo Generator tiene 3225075 parámetros entrenables.
El modelo Discriminator tiene 1168385 parámetros entrenables.
tensor([[0.2223, 0.2898, 0.7237, ..., 0.1499, 0.0492, 1.0000],
[0.8888, 0.2219, 0.2198, ..., 0.1576, 0.0782, 1.0000],
[0.2538, 0.3784, 0.6752, ..., 0.1828, 0.0454, 1.0000],
...,
[0.8658, 0.5372, 0.1708, ..., 0.3736, 0.3168, 1.0000],
[0.2013, 0.5704, 0.7373, ..., 0.0159, 0.0922, 1.0000],
[0.1293, 0.7239, 0.8890, ..., 0.0124, 0.2554, 1.0000]],
device='cuda:0', grad_fn=<SigmoidBackward0>)
Epoch 100/10000 | Disc Loss: -0.18918289244174957 | Gen Loss:
-0.2872646152973175
El código tardó 522.16918 segundos en ejecutarse.

...
tensor([[0.1694, 0.5401, 0.8082, ..., 0.1434, 0.0806, 0.9998],
[0.1580, 0.6170, 0.7817, ..., 0.1022, 0.1935, 0.9989],
[0.1696, 0.4390, 0.8565, ..., 0.1017, 0.2471, 1.0000],
...,
[0.1431, 0.5972, 0.6722, ..., 0.0817, 0.0616, 1.0000],
[0.7032, 0.6903, 0.1088, ..., 0.0877, 0.1314, 1.0000],
[0.1956, 0.4110, 0.7686, ..., 0.1011, 0.1397, 1.0000]],
device='cuda:0', grad_fn=<SigmoidBackward0>)
Epoch 10000/10000 | Disc Loss: -1.8283095359802246 | Gen Loss:
-0.4242134094238281
El código tardó 642.30941 segundos en ejecutarse.
```

Ajustes finales del modelo GAN

Tras la selección del modelo final procedimos a ajustar los últimos parámetros. Se probaron las siguientes variantes y se compararon con el modelo anterior:

- Normalización con StandardScaler
- Normalizar la asignación a los clústeres: En el fichero de asignación original los clusters están como 1 y 2, se probó con 0 y 1
- Número de veces que se actualiza el Discriminador: Se probaron los valores 2 y 8
- lambda_gp(coeficiente de penalización de gradiente): Se probaron los valores 5 y 15
- Tamaño de lote: Se probaron los valores 32 y 128
- Número de épocas de entrenamiento: Se probaron los valores 5000 y 15000

Al final, ninguna de estas variantes demostró mejorar el modelo y se continuó con el modelo anterior que ya estaba correctamente ajustado.

Validación GAN

Las métricas usadas para validar el modelo son las siguientes:

- Distancia de Wasserstein
- Test Kolmogorov-Smirnov
- Distancia euclidiana promedio
- Matrices de correlación de datos reales y generados
- Clasificación con SVC de datos reales y generados
- Clasificación con SVC de datos generados
- Dendogramas jerárquicos de datos reales y generados
- Clasificación con k-NN
- t-SNE
- UMAP
- PCA
- KDE de distintas variables al azar

El código usado para la validación del modelo se puede consultar en [5.3](#)

Ajustes del modelo GAN para datos reales

- Se pasó el entrenamiento a CPU ya que el modelo ocupaba unos 18 GB de memoria.
- Se ha cambiado el batch_size a 8 debido a que los datos reales tienen menos samples.
- Se mejoró el sistema de guardado de modelo durante el entrenamiento ya que el entrenamiento duró 25 días
- Se entrenó solo 6500 épocas por falta de tiempo. El entreno de esas épocas llevó alrededor de 25 días completos de cómputo.

2.4.2. VAE

Exploración y selección de modelo VAE

Dado que el objetivo principal del trabajo estaba centrado en los modelos GANs, la exploración de los modelos VAE no fue tan exhaustiva. Seguramente por ello el modelo seleccionado no expresa toda la potencialidad de la arquitectura VAE.

Igual que antes, para reducir tiempo de entrenamiento, parte de la exploración inicial se realizó con el conjunto de datos de 500 muestras y el ajuste del modelo final con los conjuntos de datos de 3.000 y 10.000 muestras.

Al final el modelo seleccionado fue el siguiente:

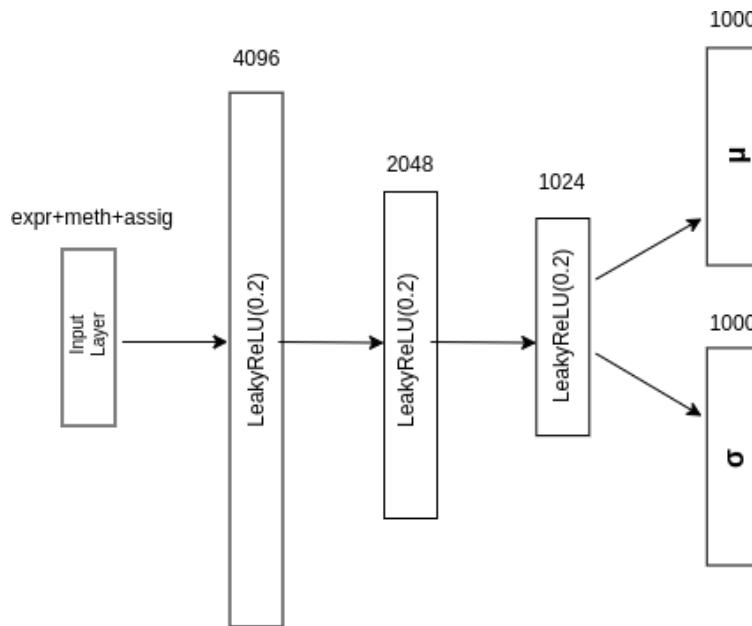


Figura 2.10: Encoder Modelo Final VAE

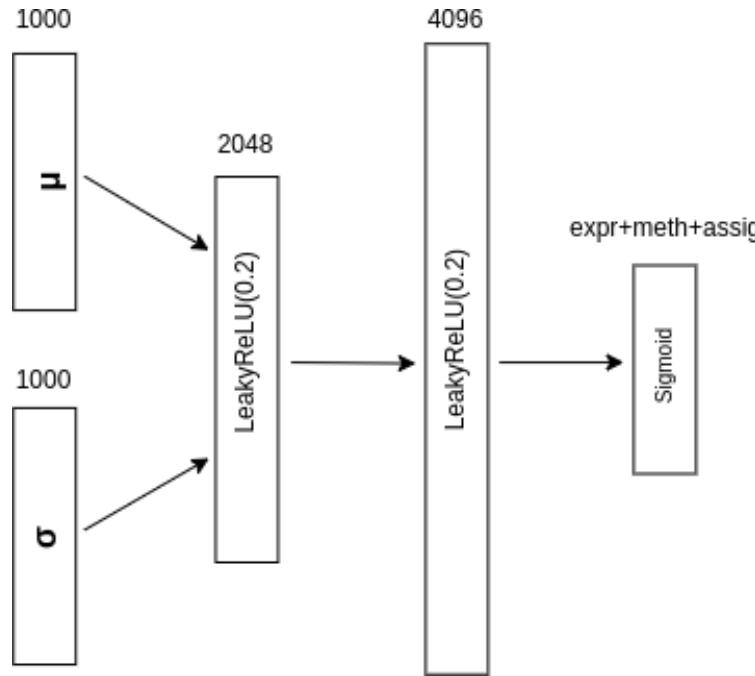


Figura 2.11: Decoder Modelo Final VAE

Como podemos ver el Encoder consta de 3 capas ocultas(con 4096, 2048 y 1024 neuronas). La capa de entrada tiene el tamaño del numero de variables total(expresión, metilación y asignación) y la capa de salida tiene el tamaño del espacio latente, 2000 en este caso(1000 para las medias y 1000 para las desviaciones).

El Decoder consta de 2 capas ocultas(con 2048 y 4096 neuronas). La capa de entrada tiene el tamaño del espacio latente y la capa de salida tiene el tamaño del numero de variables total(expresión, metilación y asignación).

Los parámetros de entrenamiento fueron:

- Optimizador: RMSprop
- Learning rate = 0.0002
- Función de perdida de reconstrucción : BCELoss
- Pérdida total = $5 * \text{recon_loss} + \text{kl_divergence}$
- n_epochs=10000 (número de épocas de entrenamiento)

Además se realiza un recorte de pesos para estabilizar el entrenamiento.

Se puede acceder al código y resultados del modelo en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfal3utRY1si?usp=sharing>

Nombre de archivo: ModelVAEFINAL.ipynb

Ejemplo de salida del entrenamiento:

```
El modelo VAE tiene 27073987 parámetros entrenables.
tensor([[0.1990, 0.6487, 0.7670, ..., 0.1787, 0.3488, 0.0000],
[0.2804, 0.4655, 0.8138, ..., 0.0449, 0.3330, 0.0000],
[0.2711, 0.4546, 0.5484, ..., 0.2769, 0.2343, 0.0000],
...,
[0.2880, 0.2426, 0.6411, ..., 0.3764, 0.9219, 0.0000],
[0.2210, 0.4209, 0.7431, ..., 0.1352, 0.2251, 0.0000],
[0.1787, 0.5634, 0.6677, ..., 0.2881, 0.1554, 0.0000]],
device='cuda:0')
tensor([[0.2066, 0.5413, 0.7733, ..., 0.2736, 0.2686, 0.0000],
[0.2211, 0.4943, 0.7454, ..., 0.1324, 0.1340, 0.0000],
[0.2107, 0.5277, 0.7586, ..., 0.1769, 0.2191, 0.0000],
...,
[0.2107, 0.4996, 0.7907, ..., 0.4777, 0.5830, 0.0000],
[0.2202, 0.4864, 0.7408, ..., 0.2465, 0.2731, 0.0000],
[0.1897, 0.5525, 0.7688, ..., 0.1298, 0.1346, 0.0000]],
device='cuda:0')
Epoch 100/10000 | Total Loss: 17999.310546875 | Recon Loss: 228.28055703125 | KL
Div: 7.405713665008545
El código tardó 228.94472 segundos en ejecutarse.
Saved model
...
```

```

tensor([[0.1990, 0.6487, 0.7670, ..., 0.1787, 0.3488, 0.0000],
[0.2804, 0.4655, 0.8138, ..., 0.0449, 0.3330, 0.0000],
[0.2711, 0.4546, 0.5484, ..., 0.2769, 0.2343, 0.0000],
...,
[0.2880, 0.2426, 0.6411, ..., 0.3764, 0.9219, 0.0000],
[0.2210, 0.4209, 0.7431, ..., 0.1352, 0.2251, 0.0000],
[0.1787, 0.5634, 0.6677, ..., 0.2881, 0.1554, 0.0000]],
device='cuda:0')
tensor([[0.2124, 0.5730, 0.7242, ..., 0.1880, 0.3372, 0.0000],
[0.2844, 0.4407, 0.8316, ..., 0.0850, 0.3321, 0.0000],
[0.2498, 0.5228, 0.5679, ..., 0.2118, 0.2097, 0.0000],
...,
[0.2717, 0.1692, 0.6251, ..., 0.3391, 0.8777, 0.0000],
[0.2720, 0.4973, 0.7533, ..., 0.1527, 0.2429, 0.0000],
[0.1645, 0.5822, 0.7001, ..., 0.2459, 0.1554, 0.0000]],
device='cuda:0')
Epoch 10000/10000 | Total Loss: 17574.78515625 | Recon Loss: 220.0576673828125 |
KL Div: 21.422170651245118
El código tardó 229.19173 segundos en ejecutarse.

```

Validación VAE

Las métricas usadas para validar el modelo son las siguientes:

- Distancia de Wasserstein
- Test Kolmogorov-Smirnov
- Distancia euclidiana promedio
- Matrices de correlación de datos reales y generados
- Clasificación con SVC de datos reales y generados
- Clasificación con SVC de datos generados
- Dendogramas jerárquicos de datos reales y generados
- Clasificación con k-NN
- t-SNE
- UMAP
- PCA
- KDE de distintas variables al azar

El código usado para la validación del VAE se puede consultar en [5.3](#)

Ajustes del modelo VAE para datos reales

- Se pasó el entrenamiento a CPU ya que el modelo ocupaba unas 45 GB de memoria.
- Se ha cambiado el batch_size a 8 debido a que los datos reales tienen menos samples.

- Se añadió Batch Normalization al Encoder y al Decoder para evitar el problema de desvanecimiento de gradiente.
- Se mejoró el sistema de guardado de modelo durante el entrenamiento ya que el entrenamiento duró 20 días
- Se entrenó solo 6500 épocas por falta de tiempo. El entreno de esas épocas llevo alrededor de 20 días completos de cómputo.

2.4.3. Integración de arquitectura Transformer en GAN y exploración de modelos

En este apartado exploramos el objetivo inicial de mejorar la GAN usando un Transformer. Después de la investigación, concluimos que lo mejor era integrar un Transformer Encoder en el Generador.

El uso del Transformer completo(Encoder y Decoder) no daba buenos resultados, creemos que es porque los datos no son secuenciales y la arquitectura Transformer está especialmente diseñada para tratar datos secuenciales.

Tras integrar el Transformer en el Generador se procedió a una exploración rápida de distintos modelos e hiperparámetros para seleccionar los mejores candidatos. Para ello usamos el conjunto de datos generado de 500 muestras y procedimos a probar distintas arquitecturas e hiperparámetros. Entre las variables que se exploraron estaban:

- Número de cabezas de atención: Se probaron 10,20 y 50, entre otros
- Número de neuronas en dimensión oculta: Se probaron 1024, 2048 y 4096, entre otros
- Numero de bloques de Transformer: Se probaron 3,6,12 y 18, entre otros

Dado la limitación en capacidad de cómputo y en tiempo de desarrollo se decidió usar una exploración aleatoria e intuitiva no completa. Para ello se usó el análisis de la evolución de los valores de las funciones de pérdida de Generador y Discriminador, la exploración visual de los datos generados y el conocimiento adquirido mediante el aprendizaje de las variaciones que sufrían los datos dependiendo de las variables exploradas.

Se generaron cerca de 25 modelos. Podemos ver el código de un ejemplo completo en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfaI3utRY1si?usp=sharing>

Nombre de fichero: oldmodels\MFWGANPTR5.ipynb

Debajo tenemos el diagrama de uno de los modelos probados.

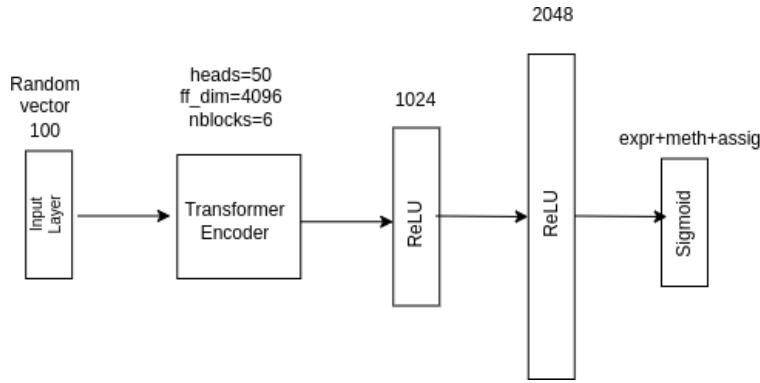


Figura 2.12: Generador Modelo GANFORMER

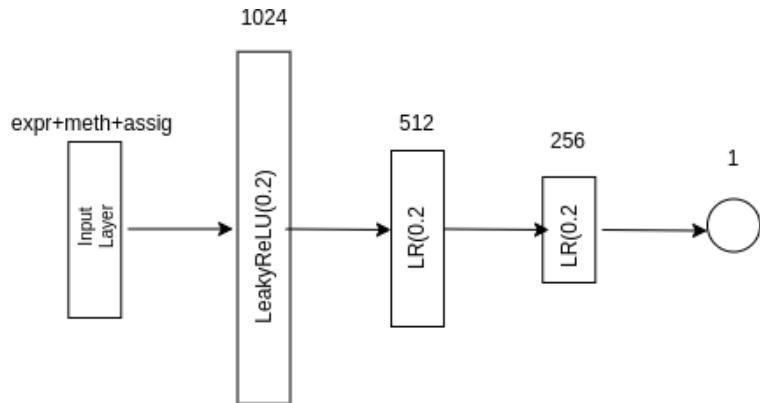


Figura 2.13: Discriminador Modelo GANFORMER

En este modelo mostrado podemos ver como se usa un Transformer Encoder antes de las capas normales del Generador con 50 cabezas de atención, una capa densa con 4096 neuronas y 6 bloques de Encoder.

Los resultados obtenidos no fueron satisfactorios y se decidió explorar los modelos GAN y Transformer en la conversión de datos ómicos.

2.5. Conversión entre datos ómicos

En este apartado se exploran modelos que convierten datos ómicos de expresión a metilación y viceversa.

A diferencia de los modelos anteriores de generación de datos sintéticos, estos modelos reciben datos ómicos de un dominio y devuelven datos ómicos de otro dominio.

2.5.1. Exploración modelos GAN y TRANSFORMER Conversión Expresión <->Metilación

Del mismo modo que en las exploraciones de modelos anteriores, se procedió a explorar los modelos GAN y Transformer .

Para ello se exploraron cuatro tipos de modelos:

- Modelo GAN de conversión Expresión→Metilación
- Modelo GAN de conversión Metilación→Expresión
- Modelo Transformer de conversión Expresión→Metilación
- Modelo Transformer de conversión Metilación→Expresión

Para cada tipo de modelo se procedió a una exploración específica usando la misma metodología anterior y explorando los mismo parámetros. En concreto, en los modelos GAN se incidió en la exploración de cómo incidía la arquitectura(número de capas y número de neuronas) en los resultados. En los modelos Transformer, se incidió en la exploración de los parámetros específicos de los Transformer(numero de cabezas de atención, numero de neuronas de la dimensión oculta y números de bloques Transformer)

La estructura de un modelo Transformer, que incluye tanto un codificador (encoder) como un decodificador (decoder), es típica de tareas de secuencia a secuencia donde tanto la entrada como la salida durante el entrenamiento son conocidas (como en la traducción automática). Sin embargo, en este caso la estructura habitual del modelo debe adaptarse.

Por ello se ha optado usar una arquitectura de Transformer Encoder-Only

Para comparar estos modelos usamos distintas métricas, entre ellas:

- MSE y RMSE: EL MSE es el promedio de los cuadrados de las diferencias entre los valores predichos y los valores reales. Proporciona una medida de la calidad de un estimador; en este caso, cuán bien un modelo puede predecir datos de la ómica objetivo a partir de datos de la ómica origen. El RMSE es simplemente la raíz cuadrada del MSE. Tiene la ventaja de estar en las mismas unidades que los datos de salida, lo que facilita la interpretación de su magnitud.
- PCC: Mide la correlación lineal entre dos conjuntos de datos. En este contexto, se utiliza para evaluar la relación lineal entre los valores reales y los predichos por el modelo
- t-SNE: Entre los datos ómicos objetivos reales y los convertidos por el modelo. Ayuda a ver que los datos que ofrece el modelo tienen la misma distribución y agrupación.
- UMAP: ídem anterior
- PCA: ídem anterior

En total se generaron cerca de 40 modelos. Podemos ver los modelos finales seleccionados en los siguientes apartados.

2.5.2. Modelo final TRANSFORMER Conversión Expresión→Metilación

Debajo tenemos el modelo seleccionado de Transformer para la conversión de Expresión a Metilación

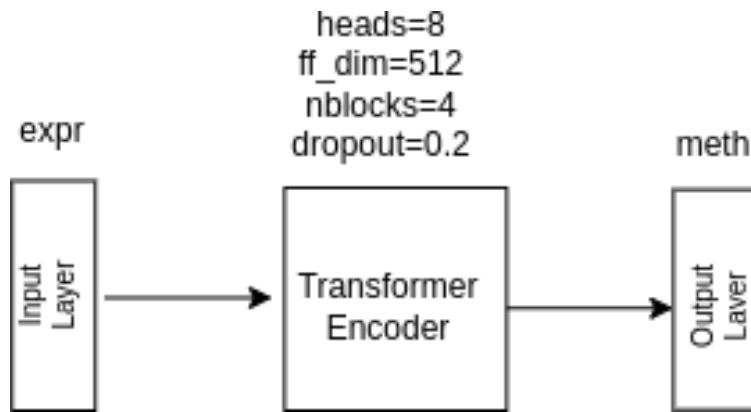


Figura 2.14: Modelo Final Transformer Expresión→Metilación

Como podemos ver el modelo consta de 1 Transformer Encoder con 8 cabezas de atención, una capa densa de 512 neuronas con dropout de 0.2 y 4 bloques de Encoder. La capa de entrada tiene el tamaño del numero de variables total de expresión y la capa de salida tiene el tamaño del número de variables total de metilación.

Los parámetros de entrenamiento fueron:

- Optimizador: RMSprop
- Learning rate = 0.0002
- Función de perdida: MSELoss
- n_epochs=100 (número de épocas de entrenamiento)

Se puede acceder al código y resultados del modelo en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfa3utRY1si?usp=sharing>

Nombre de archivo: ModelETMTRFINAL.ipynb

Aquí tenemos un extracto de la salida del entrenamiento:

```
Epoch 1, Train Loss: 0.0510
Saved model with Train Loss: 0.051003

...
Epoch 99, Train Loss: 0.0117
Saved model with Train Loss: 0.011742
Epoch 100, Train Loss: 0.0117
Saved model with Train Loss: 0.011700
```

2.5.3. Modelo final GAN Conversión Expresión→Metilación

Debajo tenemos el modelo seleccionado de GAN para la conversión de Expresión a Metilación

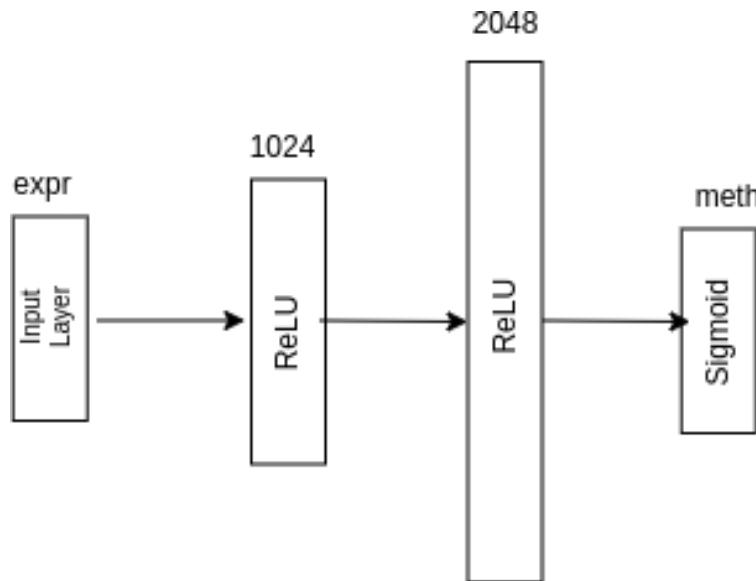


Figura 2.15: Generador Modelo Final GAN Expresión a Metilación

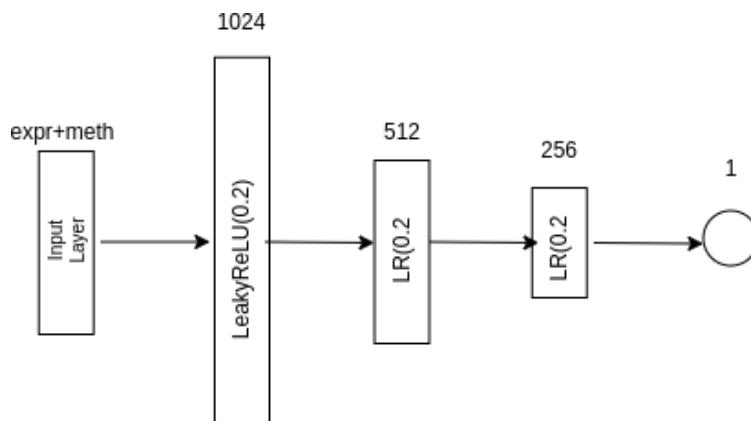


Figura 2.16: Discriminador Modelo Final GAN Expresión a Metilación

Como podemos ver el Generador consta de 2 capas ocultas(unas con 1024 neuronas y la otra con 2048). La capa de entrada tiene el tamaño del número total de variables de expresión y la capa de salida tiene el tamaño del numero de variables total de metilación.

El Discriminador consta de 3 capas ocultas(con 1024, 512 y 256 neuronas). La capa de entrada tiene el tamaño del numero de variables total(expresión y metilación) y la capa de salida tiene 1 neurona que devuelve la puntuación de Wasserstein.

Los parámetros de entrenamiento fueron:

- Optimizador: RMSprop para Generador y Discriminador
- Learning rate = 0.0002 para ambos
- n_critic = 5 (número de veces que se actualiza el Discriminador)
- lambda_gp = 10 (Coeficiente de penalización de gradiente)
- n_epochs=10000 (número de épocas de entrenamiento)

Se puede acceder al código y resultados del modelo en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfaI3utRY1si?usp=sharing>

Nombre de archivo: ModelETMGFINAL.ipynb

Aquí tenemos un extracto de la salida del entrenamiento:

```
El modelo Generator tiene 2986351 parámetros entrenables.
El modelo Discriminator tiene 1167361 parámetros entrenables.
tensor([[0.7145, 0.4985, 0.0985, ..., 0.2153, 0.3050, 0.2752],
[0.8106, 0.4922, 0.2808, ..., 0.1933, 0.1596, 0.1642],
[0.1727, 0.5079, 0.7407, ..., 0.1385, 0.1602, 0.2016],
...,
[0.1415, 0.7692, 0.7691, ..., 0.1475, 0.1963, 0.1506],
[0.1504, 0.7744, 0.7458, ..., 0.1999, 0.0708, 0.2026],
[0.7504, 0.3782, 0.1947, ..., 0.1890, 0.1143, 0.1801]],
device='cuda:0', grad_fn=<CatBackward0>
Epoch 100/10000 | Disc Loss: -0.8761624097824097 | Gen Loss: 0.11704029142856598
El código tardó 299.71799 segundos en ejecutarse.

...
tensor([[0.2081, 0.5523, 0.7298, ..., 0.2217, 0.1578, 0.2121],
[0.7747, 0.4977, 0.2258, ..., 0.1799, 0.3047, 0.2045],
[0.1973, 0.5197, 0.7091, ..., 0.1564, 0.0452, 0.1732],
...,
[0.2565, 0.5387, 0.7886, ..., 0.1775, 0.2337, 0.1884],
[0.7860, 0.5795, 0.2049, ..., 0.1880, 0.1810, 0.1464],
[0.8153, 0.7501, 0.2820, ..., 0.2355, 0.1042, 0.2080]],
device='cuda:0', grad_fn=<CatBackward0>
Epoch 10000/10000 | Disc Loss: -0.02598470076918602 | Gen Loss:
1.2188583612442017
El código tardó 400.16688 segundos en ejecutarse.
```

2.5.4. Modelo final TRANSFORMER Conversión Metilación→Expresión

Debajo tenemos el modelo seleccionado de Transformer para la conversión de Metilación a Expresión.

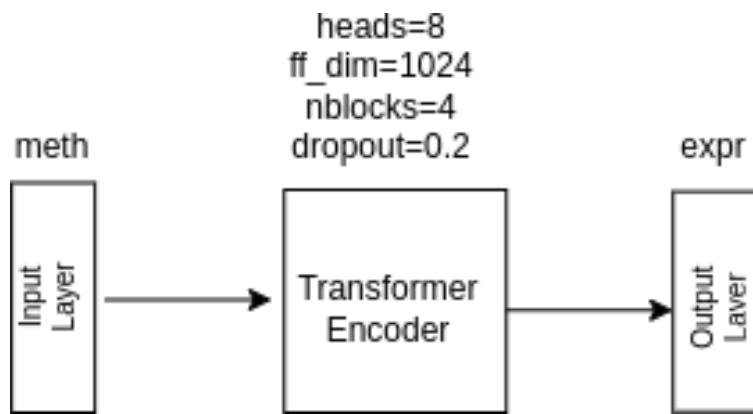


Figura 2.17: Modelo Final Transformer Metilación→Expresión

Como podemos ver el modelo consta de 1 Transformer Encoder con 8 cabezas de atención, una capa densa de 1024 neuronas con dropout de 0.2 y 4 bloques de Encoder. La capa de entrada tiene el tamaño del numero de variables total de metilación y la capa de salida tiene el tamaño del número de variables total de expresión.

Los parámetros de entrenamiento fueron:

- Optimizador: RMSprop
- Learning rate = 0.0002
- Función de perdida: MSELoss
- n_epochs=100 (número de épocas de entrenamiento)

Se puede acceder al código y resultados del modelo en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfaI3utRY1si?usp=sharing>

Nombre de archivo: ModelMTETRFINAL.ipynb

Aquí tenemos un extracto de la salida del entrenamiento:

```

Epoch 1, Train Loss: 0.0933
Saved model with Train Loss: 0.093329

...
Epoch 96, Train Loss: 0.0126
Saved model with Train Loss: 0.012615
Epoch 97, Train Loss: 0.0126
Epoch 98, Train Loss: 0.0127
Epoch 99, Train Loss: 0.0126
Epoch 100, Train Loss: 0.0126

```

2.5.5. Modelo final GAN Conversión Metilación→Expresión

Debajo tenemos el modelo seleccionado de GAN para la conversión de Metilación a Expresión.

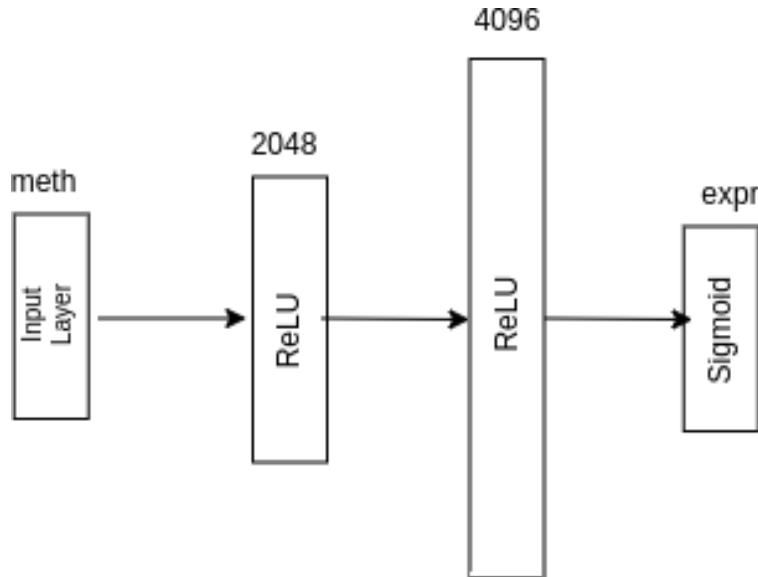


Figura 2.18: Generador Modelo Final GAN Metilación a Expresión

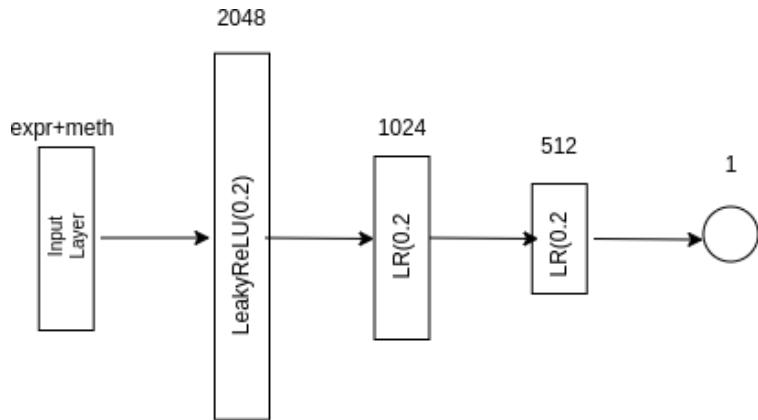


Figura 2.19: Discriminador Modelo Final GAN Metilación a Expresión

Como podemos ver el Generador consta de 2 capas ocultas(unas con 2048 neuronas y la otra con 4096). La capa de entrada tiene el tamaño del número total de variables de metilación y la capa de salida tiene el tamaño del numero de variables total de expresión.

El Discriminador consta de 3 capas ocultas(con 2048, 1024 y 512 neuronas). La capa de entrada tiene el tamaño del numero de variables total(expresión y metilación) y la capa de salida tiene 1 neurona que devuelve la puntuación de Wasserstein.

Los parámetros de entrenamiento fueron:

- Optimizador: RMSprop para Generador y Discriminador
- Learning rate = 0.0002 para ambos
- n_critic = 5 (número de veces que se actualiza el Discriminador)
- lambda_gp = 10 (Coeficiente de penalización de gradiente)
- n_epochs=10000 (número de épocas de entrenamiento)

Se puede acceder al código y resultados del modelo en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfa13utRY1si?usp=sharing>

Nombre de archivo: ModelMTETGFFINAL.ipynb

Aquí tenemos un extracto de la salida del entrenamiento:

```
El modelo Generator tiene 9683075 parámetros entrenables.
El modelo Discriminator tiene 3645441 parámetros entrenables.
tensor([[0.2179, 0.4674, 0.7808, ..., 0.4340, 0.1132, 0.2723],
[0.1969, 0.4333, 0.7711, ..., 0.1162, 0.0648, 0.0573],
[0.2095, 0.4862, 0.8035, ..., 0.1200, 0.1396, 0.0883],
...,
[0.2311, 0.6196, 0.8016, ..., 0.1685, 0.0093, 0.0753],
[0.2288, 0.4081, 0.7106, ..., 0.3108, 0.1574, 0.2948],
[0.7748, 0.5227, 0.1816, ..., 0.0910, 0.0064, 0.0435]],
device='cuda:0', grad_fn=<CatBackward0>
Epoch 100/10000 | Disc Loss: -0.28299927711486816 | Gen Loss:
-1.5862239599227905

...
El código tardó 536.70104 segundos en ejecutarse.
tensor([[0.2832, 0.6746, 0.8086, ..., 0.1202, 0.2343, 0.1171],
[0.2509, 0.6460, 0.7817, ..., 0.1872, 0.0238, 0.0706],
[0.1989, 0.4894, 0.7668, ..., 0.1795, 0.0504, 0.0840],
...,
[0.2488, 0.6827, 0.8149, ..., 0.2576, 0.1283, 0.2404],
[0.3141, 0.6440, 0.6669, ..., 0.0809, 0.0538, 0.0913],
[0.1871, 0.7521, 0.7329, ..., 0.3783, 0.3854, 0.4703]],
device='cuda:0', grad_fn=<CatBackward0>
Epoch 10000/10000 | Disc Loss: -0.08675849437713623 | Gen Loss:
1.6604948043823242
El código tardó 1572.67191 segundos en ejecutarse.
```

2.5.6. Validación Conversión Expresión <->Metilación

Para la validación de los modelos se usaron las siguientes métricas

- MSE y RMSE
- PCC
- t-SNE

- UMAP
- PCA

Se puede consultar el código usado para la validación en [5.3](#)

Capítulo 3

Resultados

3.1. Generación datos multi-ómicos

En este apartado se muestran los resultados de los modelos diseñados para generar datos sintéticos multi-ómicos.

3.1.1. Modelo GAN Final con datos simulados

A continuación veremos los valores del modelo final de las métricas usadas para validar y comparar los modelos.

```
Wasserstein Distance: 0.002620999323856315  
KS Statistic: 0.012775100401606454, P-Value: 0.0  
Distancia Euclidiana Promedio: 5.430746748843026
```

La distancia de Wasserstein es muy baja lo que indica que hay pocas diferencias entre las dos distribuciones.

El estadístico del test Kolmogorov-Smirnov es muy bajo, lo que sugiere que las diferencias entre las dos distribuciones son pequeñas. Sin embargo, el P-Valor es 0.0, lo que indica que estas pequeñas diferencias son estadísticamente significativas.

La distancia euclíadiana promedio es de las más bajas que hemos obtenido con los diferentes modelos.

En la figura a continuación podemos ver como las matrices de correlación de los datos reales y los generados son muy parecidas. Debajo tenemos también la matriz de diferencias de correlación.

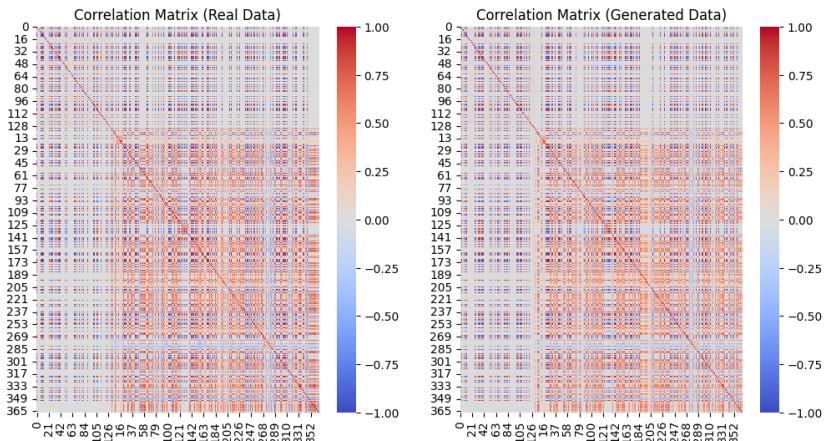


Figura 3.1: Matrices de correlación de datos reales(izq) y generados(der)

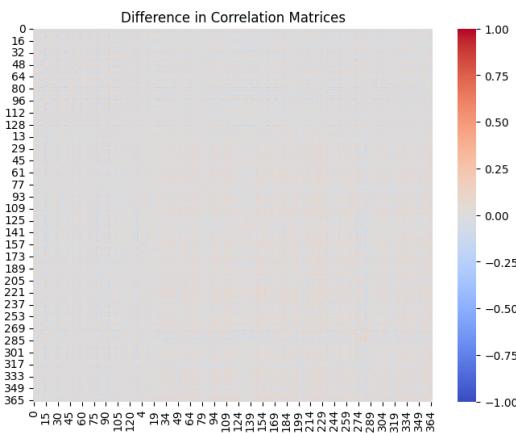


Figura 3.2: Matriz de diferencia de correlación

A continuación podemos ver el informe y el mapa de clasificación del SVC que se ha usado para clasificar los datos reales y generados. En este mapa los datos generados son los de clase 0 y los reales de clase 1.

Cómo se ha explicado anteriormente esto nos da una medida de la calidad de los datos generados ya que usamos otro clasificador para medir cuánto se parecen los datos generados a los reales. Idealmente, el SVC no debería de poder distinguir entre datos reales y generados.

Cómo podemos ver la precisión es del 0.78. Lo ideal sería del 0.5 dado que indicaría que el SVC no puede distinguir entre reales y generados.

	Precision	Recall	F1-score	Support
0	0.79	0.75	0.77	2983
1	0.77	0.80	0.78	3017
Accuracy			0.78	6000
Macro avg	0.78	0.78	0.78	6000
Weighted avg	0.78	0.78	0.78	6000

Cuadro 3.1: Clasificación datos reales y generados

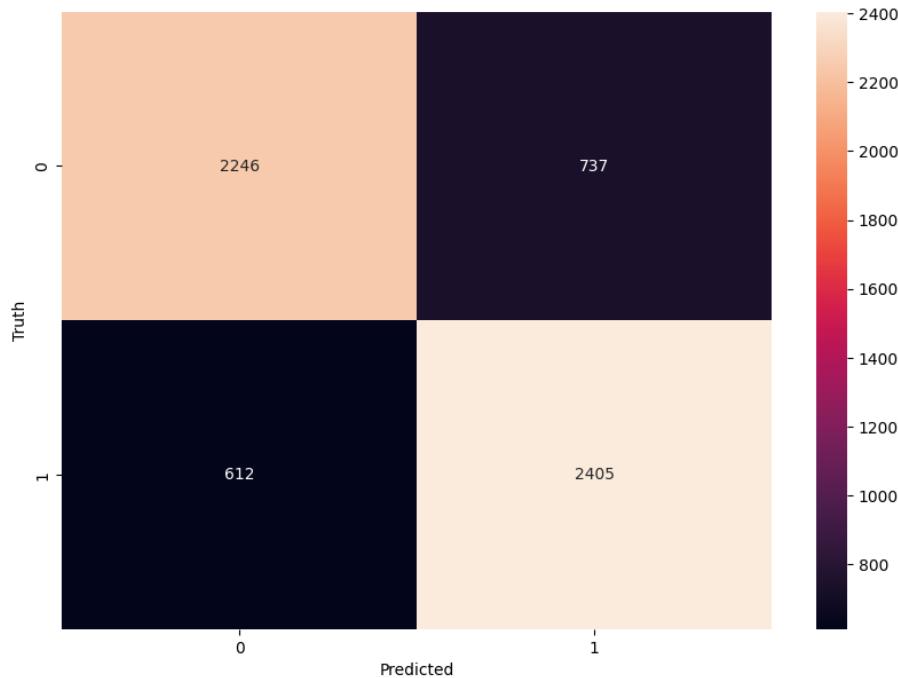


Figura 3.3: Mapa de calor de la clasificación con SVC de datos reales(1) y generados(0)

A continuación podemos ver el informe y el mapa de clasificación del SVC que se ha usado para clasificar los datos generados. Con esto se buscaba que la clasificación en los clusters de los datos generados tuvieran una distribución parecida a la clasificación de los datos reales.

	Precision	Recall	F1-score	Support
1	1.00	1.00	1.00	2370
2	1.00	1.00	1.00	630
Accuracy			1.00	3000
Macro avg	1.00	1.00	1.00	3000
Weighted avg	1.00	1.00	1.00	3000

Cuadro 3.2: Clasificación datos reales

	Precision	Recall	F1-score	Support
0	0.83	1.00	0.91	1925
1	1.00	0.63	0.78	1075
Accuracy			0.87	3000
Macro avg	0.91	0.82	0.84	3000
Weighted avg	0.89	0.87	0.86	3000

Cuadro 3.3: Clasificación datos generados

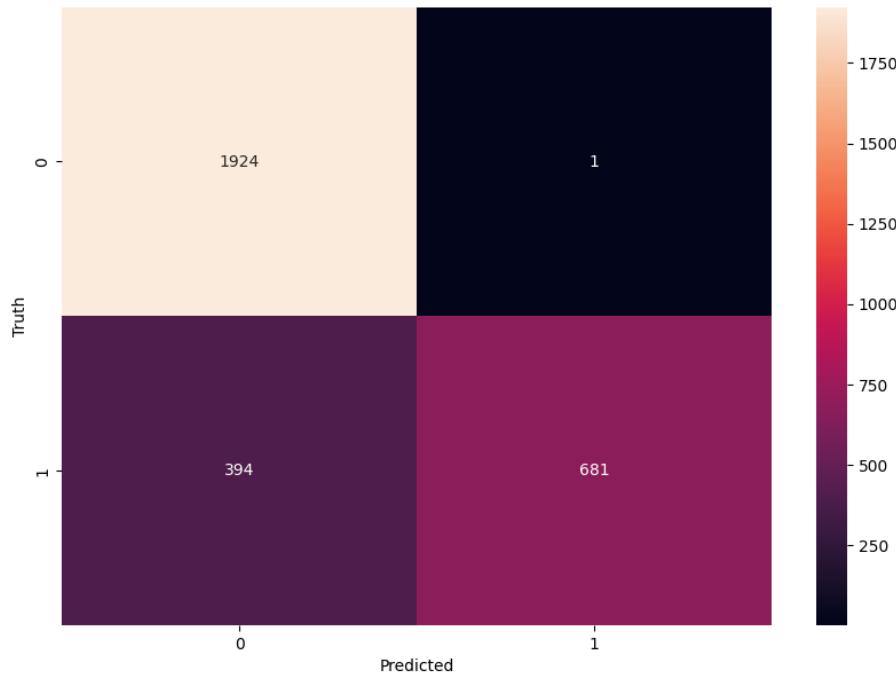


Figura 3.4: Mapa de calor de clasificación de datos generados en los clusters

Podemos observar cómo la distribución de los datos generados es parecida pero no idéntica a los datos reales. En los datos reales tenemos 2370 muestras de clase 1 y 630 muestras de clase 2.

En los datos generados tenemos 1925 muestras de clase 0 (clase 1 en real) y 1075 muestras de clase 1 (clase 2 en real). Además, de las muestras de clase 1 hay 394 que el clasificador clasifica como tipo 0.

Como podemos ver en los siguientes dendogramas jerárquicos, la clasificación y jerarquía de los datos reales y generados es muy parecida. Esto indica que el modelo ha aprendido bien la estructura subyacente de los datos.

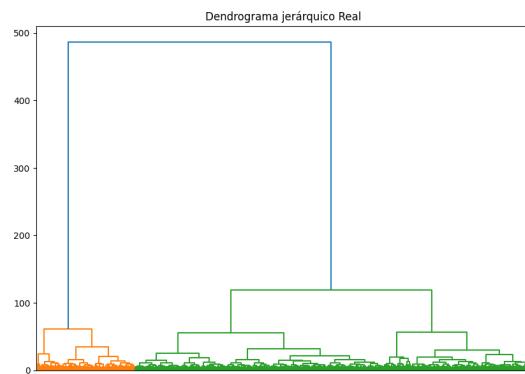


Figura 3.5: Dendrograma jerarquico datos reales.

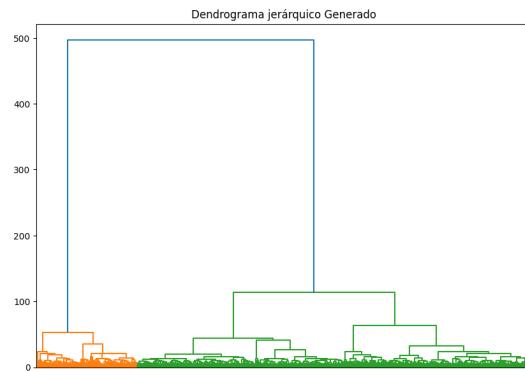


Figura 3.6: Dendrograma jerarquico datos generados.

Deabajo podemos ver cómo usando un clasificador k-NN los datos reales se separan en dos clusters y los generados también.

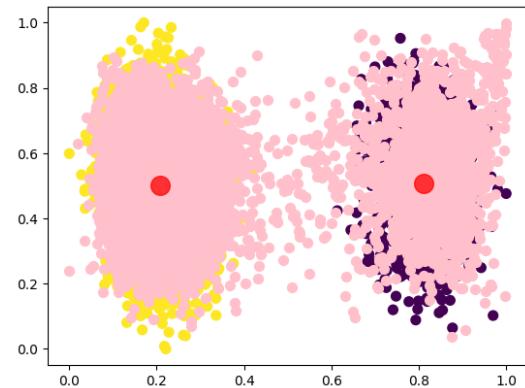


Figura 3.7: Clasificación usando K-NN. Amarillo y Azul datos reales. Rosa datos generados. Rojo centroides grupos

Como podemos ver en el siguiente diagrama del t-SNE los datos reales y generados también

se separan en los dos clusters y los generados siguen una distribución muy parecida a la de los reales. Tenemos un pequeño conjunto de datos que forma un cluster pequeño aislado.

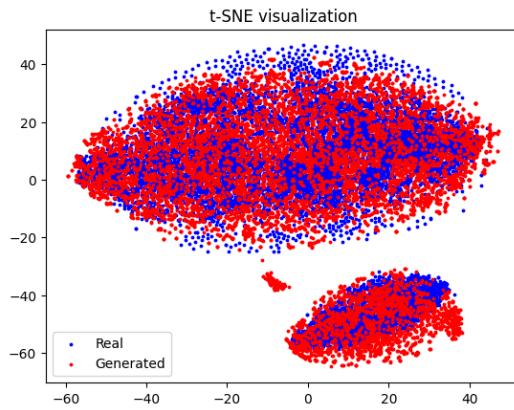


Figura 3.8: t-SNE

De igual forma el gráfico UMAP nos muestra algo parecido.

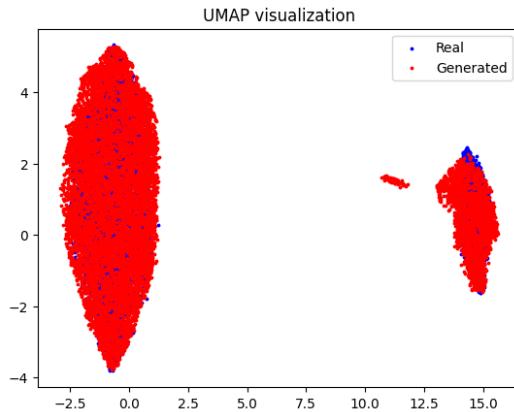


Figura 3.9: UMAP

En el PCA, los datos reales y generados también se separan en dos clusters para las dos componentes más importantes y los datos generados siguen una distribución bastante parecida. Aquí podemos observar como en uno de los clusters hay una diferencia y como algunos datos generados se quedan a medio camino entre los dos clusters.

Probablemente, esos datos que se quedan en medio son los que formen el cluster aislado que podíamos ver en el t-SNE y el UMAP.

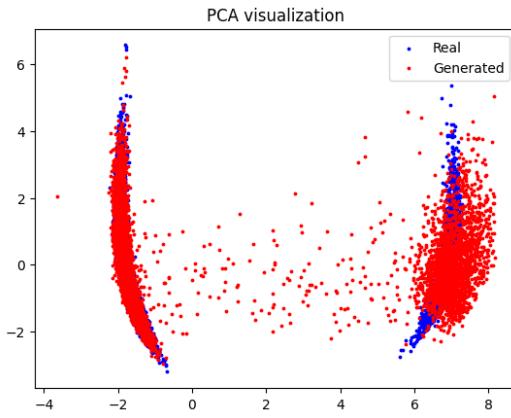


Figura 3.10: Principal Components Analysis

A continuación podemos ver los graficos KDE para algunas variables al azar. Podemos observar como las distribuciones de los datos generados se asemejan mucho a la de los reales en esas variables.

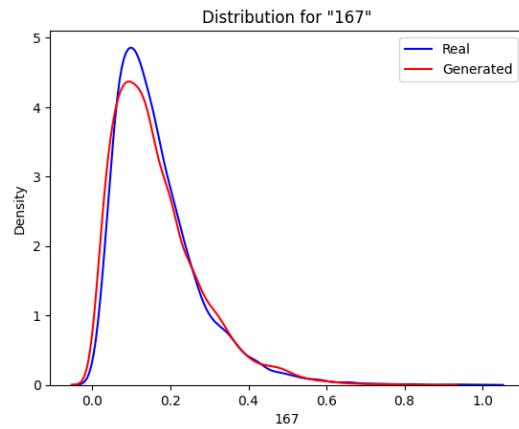


Figura 3.11: KDE

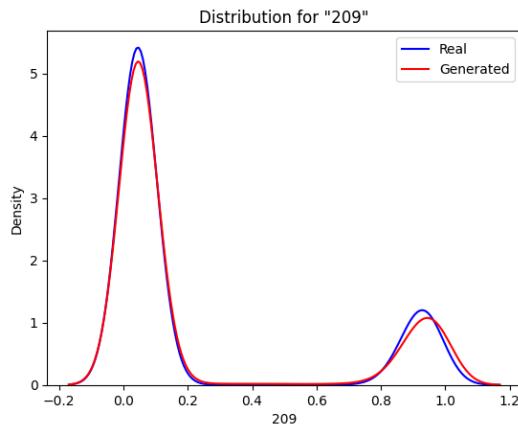


Figura 3.12: KDE

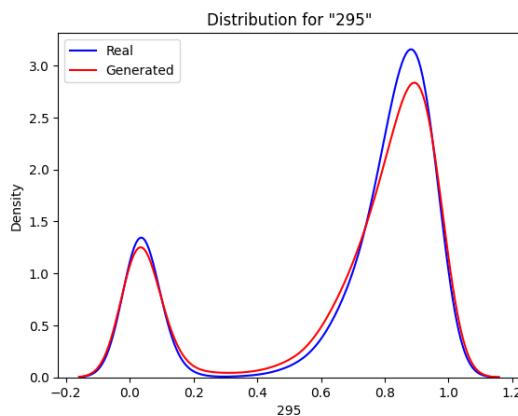


Figura 3.13: KDE

3.1.2. Modelo GAN Final con datos reales

En el modelo GAN Final con datos reales y todas las variables solo se pudo entrenar el modelo durante 6500 épocas, a diferencia de las 10000 de los datos simulados, por una cuestión de tiempo.

Se entrenaron también modelos con datos reales filtrando todas las variables con desviaciones estándar mayores al percentil 90 % durante 10000 épocas.

Se puede acceder al código y resultados de dichos modelos en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfal3utRY1si?usp=sharing>

Nombre de fichero: ModelWGANGPREALstd90.ipynb

En esta sección podemos ver los resultados del modelo entrenado durante 6500 épocas con los datos reales y todas las variables.

A continuación veremos los valores del modelo final de las métricas usadas para validar y comparar los modelos.

Wasserstein Distance: 0.03826670645535537
KS Statistic: 0.07646583814355362, P-Value: 0.0
Distancia Euclíadiana Promedio: 214.92647250940098

La distancia de Wasserstein es muy baja lo que indica que hay pocas diferencias entre las dos distribuciones.

El estadístico del test Kolmogorov-Smirnov es muy bajo, lo que sugiere que las diferencias entre las dos distribuciones son pequeñas. Sin embargo, el P-Valor es 0.0, lo que indica que estas pequeñas diferencias son estadísticamente significativas.

Podemos observar que ambos valores son más altos que con los datos simulados.

La distancia euclíadiana promedio no es comparable con la de los datos simulados por tener un mayor número de variables.

No se han podido calcular las matrices de correlación debido al gran número de variables

A continuación podemos ver el informe y el mapa de clasificación de la SVC que se ha usado para clasificar los datos reales y generados. En este mapa los datos generados son los de clase 0 y los reales de clase 1.

Cómo se ha explicado anteriormente esto nos da una medida de la calidad de los datos generados ya que usamos otro clasificador para medir cuánto se parecen los datos generados a los reales. Idealmente, el SVC no debería de poder distinguir entre datos reales y generados.

Cómo podemos ver la precisión es del 0.48, lo cual está muy bien. Lo ideal sería del 0.5 dado que indicaría que el SVC no puede distinguir entre reales y generados.

	Precision	Recall	F1-score	Support
0	0.36	0.44	0.40	9
1	0.58	0.50	0.54	14
Accuracy			0.48	23
Macro avg	0.47	0.47	0.47	23
Weighted avg	0.50	0.48	0.48	23

Cuadro 3.4: Clasificación datos generados y reales

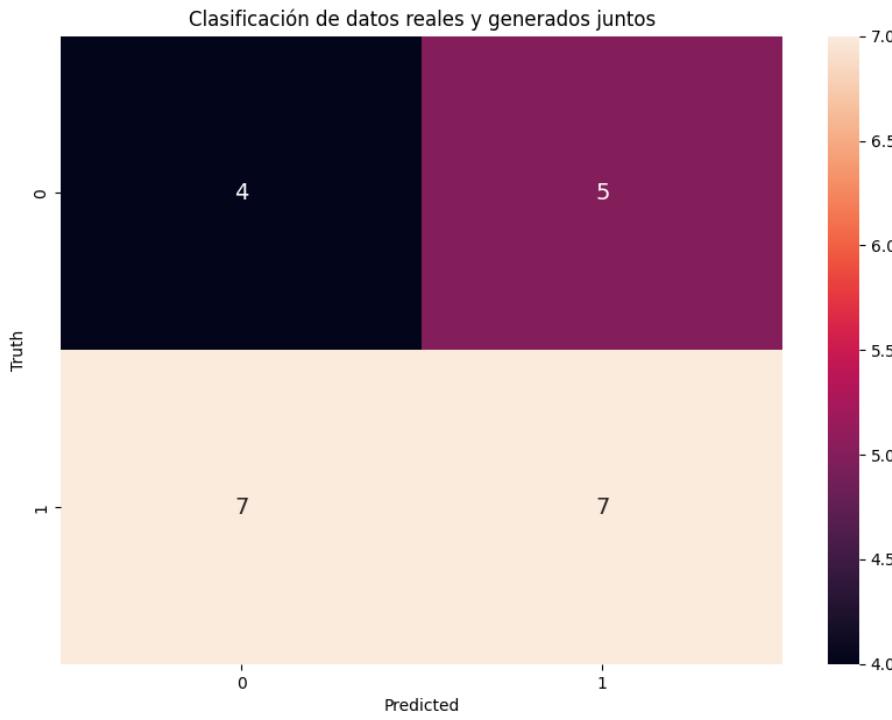


Figura 3.14: Mapa de calor de la clasificación con SVC de datos reales(1) y generados(0)

A continuación podemos ver el informe y el mapa de clasificación de la SVC que se ha usado para clasificar los datos generados. Con esto se buscaba que la clasificación en los clusters de los datos generados tuvieran una distribución parecida a la clasificación de los datos reales.

	Precision	Recall	F1-score	Support
0	1.00	1.00	1.00	12
Accuracy			1.00	12
Macro avg	1.00	1.00	1.00	12
Weighted avg	1.00	1.00	1.00	12

Cuadro 3.5: Clasificación datos generados

	Precision	Recall	F1-score	Support
0	0.88	1.00	0.93	7
1	1.00	0.80	0.89	5
Accuracy			0.92	12
Macro avg	0.94	0.90	0.91	12
Weighted avg	0.93	0.92	0.91	12

Cuadro 3.6: Clasificación datos reales

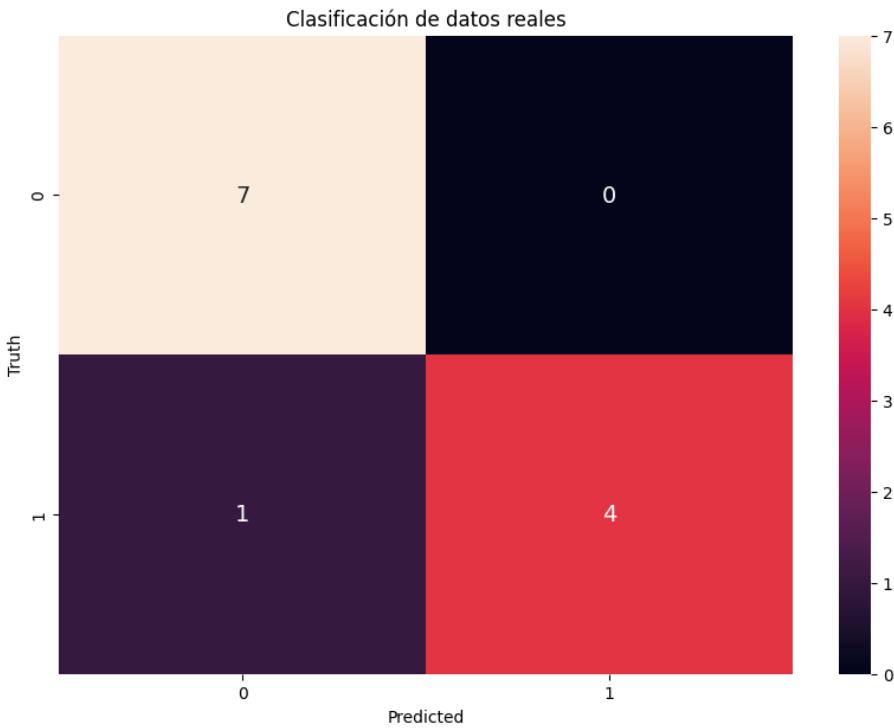


Figura 3.15: Mapa de calor de clasificación de datos reales

Podemos observar cómo la distribución de los datos generados es totalmente diferente a los datos reales. En los datos reales tenemos 7 muestras de clase 0 y 5 muestras de clase 1. En los datos generados tenemos 12 muestras de clase 0.

Como podemos ver en los siguientes dendogramas jerárquicos, la clasificación y jerarquía de los datos reales y generados no es demasiado parecida. Esto indica que el modelo no ha aprendido bien la estructura subyacente de los datos.

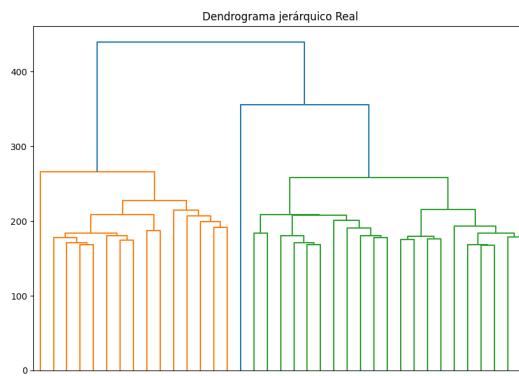


Figura 3.16: Dendograma jerárquico datos reales.

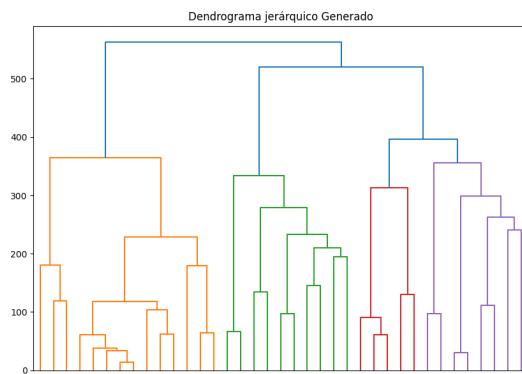


Figura 3.17: Dendrograma jerarquico datos generados.

Deabajo podemos ver cómo usando a un clasificador k-NN le cuesta agrupar los datos reales por la poca cantidad de muestras y como los datos generados están cercanos a los reales.

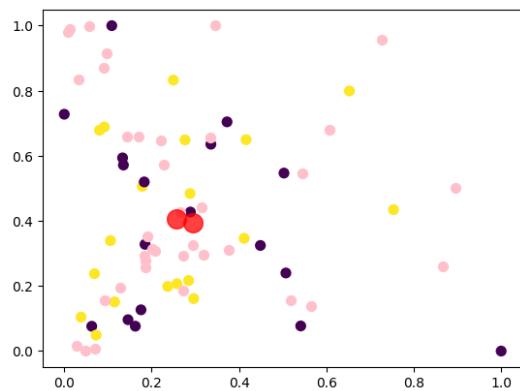


Figura 3.18: Clasificación usando K-NN. Amarillo y Azul datos reales. Rosa datos generados. Rojo centroides grupos

Mientras que en el resto de las gráficas y en el resto de modelos enseñados anteriormente se generaban la misma cantidad de datos generados que reales para mostrar, en este caso se han generado 500 datos para poder visualizar cómo la generación de datos se agrupa alrededor de los datos reales.

Como podemos ver en el siguiente diagrama del t-SNE los datos generados se agrupan alrededor de los datos reales y en una zona intermedia.

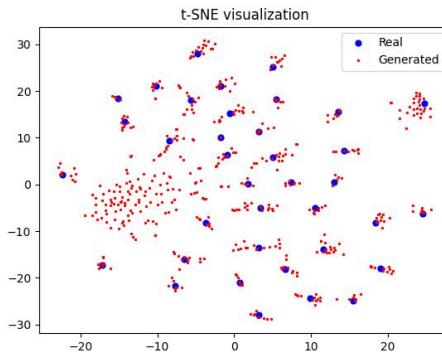


Figura 3.19: t-SNE

De igual forma el gráfico UMAP nos muestra algo parecido.

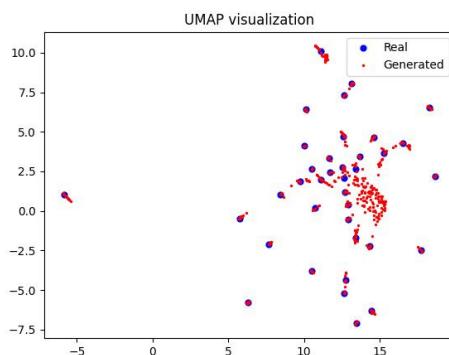


Figura 3.20: UMAP

En el PCA podemos ver cómo esa zona intermedia toma mayor protagonismo y una mayor dispersión de los datos generados.

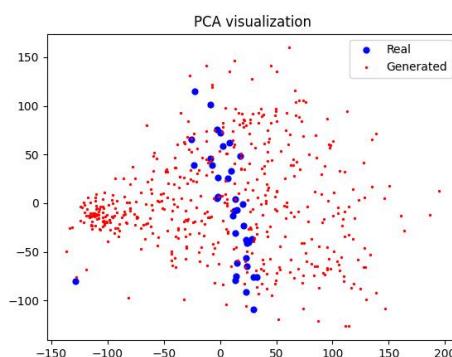


Figura 3.21: Principal Components Analysis

A continuación podemos ver los gráficos KDE para algunas variables al azar. Podemos observar como las distribuciones de los datos generados se asemejan pero no son idénticas a la de los reales en esas variables.

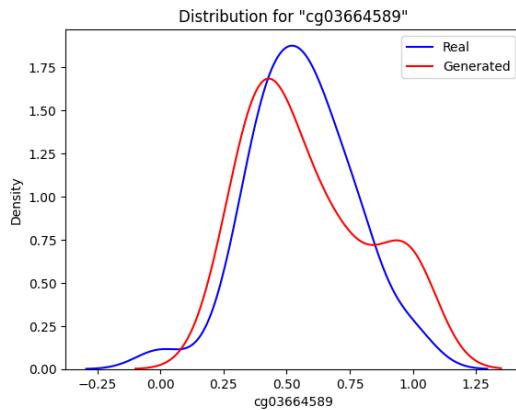


Figura 3.22: KDE

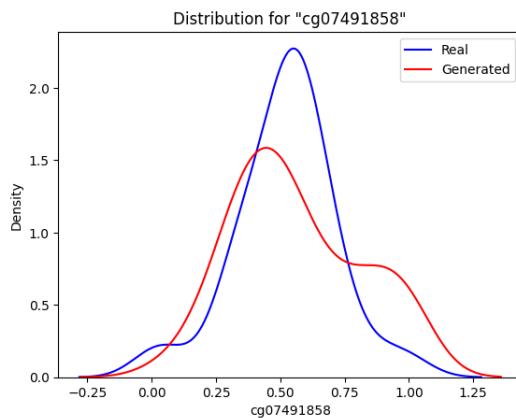


Figura 3.23: KDE

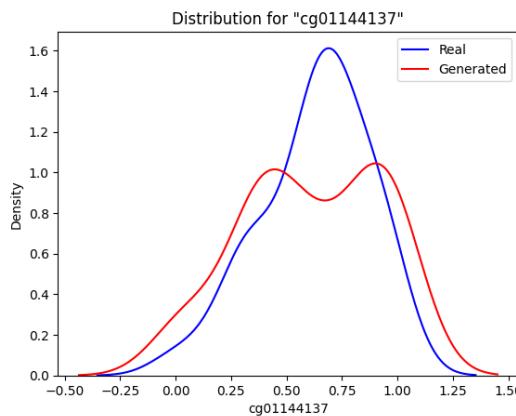


Figura 3.24: KDE

3.1.3. Modelo VAE Final con datos simulados

A continuación veremos los valores del modelo final de las métricas usadas para validar y comparar los modelos.

Wasserstein Distance: 0.008742039096039872

KS Statistic: 0.01700783132530123, P-Value: 0.0

Distancia Euclíadiana Promedio: 5.008909290363482

La distancia de Wasserstein es muy baja lo que indica que hay pocas diferencias entre las dos distribuciones.

El estadístico del test Kolmogorov-Smirnov es muy bajo, lo que sugiere que las diferencias entre las dos distribuciones son pequeñas. Sin embargo, el P-Valor es 0.0, lo que indica que estas pequeñas diferencias son estadísticamente significativas.

La distancia euclíadiana promedio es de las más bajas que hemos obtenido con los diferentes modelos.

En la figura a continuación podemos ver como las matrices de correlación de los datos reales y los generados son parecidas. Y debajo tenemos la matriz de diferencia de correlación.

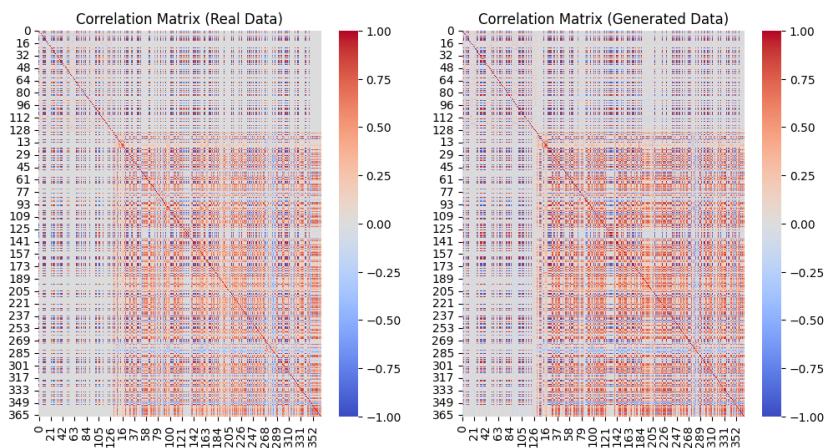


Figura 3.25: Matrices de correlación de datos reales(izq) y generados(der)

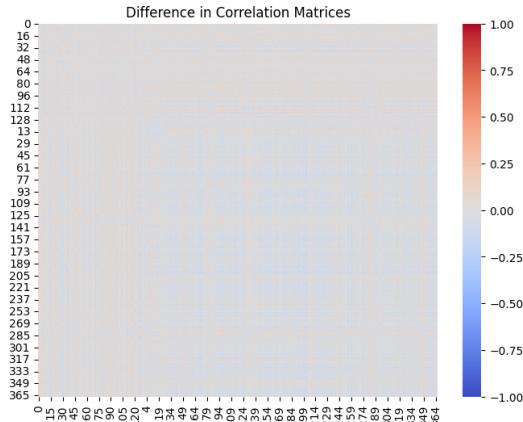


Figura 3.26: Matriz de diferencia de correlación

A continuación podemos ver el informe y el mapa de clasificación de la SVC que se ha usado para clasificar los datos reales y generados. En este mapa los datos generados son los de clase 0 y lo reales de clase 1.

Cómo se ha explicado anteriormente esto nos da una medida de la calidad de los datos generados ya que usamos otro clasificador para medir cuanto se parecen los datos generados a los reales. Idealmente, el SVC no debería de poder distinguir entre datos reales y generados.

Cómo podemos ver la precisión es del 0.96. Lo ideal sería del 0.5 dado que indicaría que el SVC no puede distinguir entre reales y generados.

	Precision	Recall	F1-score	Support
0	0.96	0.97	0.96	2983
1	0.97	0.96	0.96	3017
Accuracy			0.96	6000
Macro avg	0.96	0.96	0.96	6000
Weighted avg	0.96	0.96	0.96	6000

Cuadro 3.7: Clasificación datos reales y generados

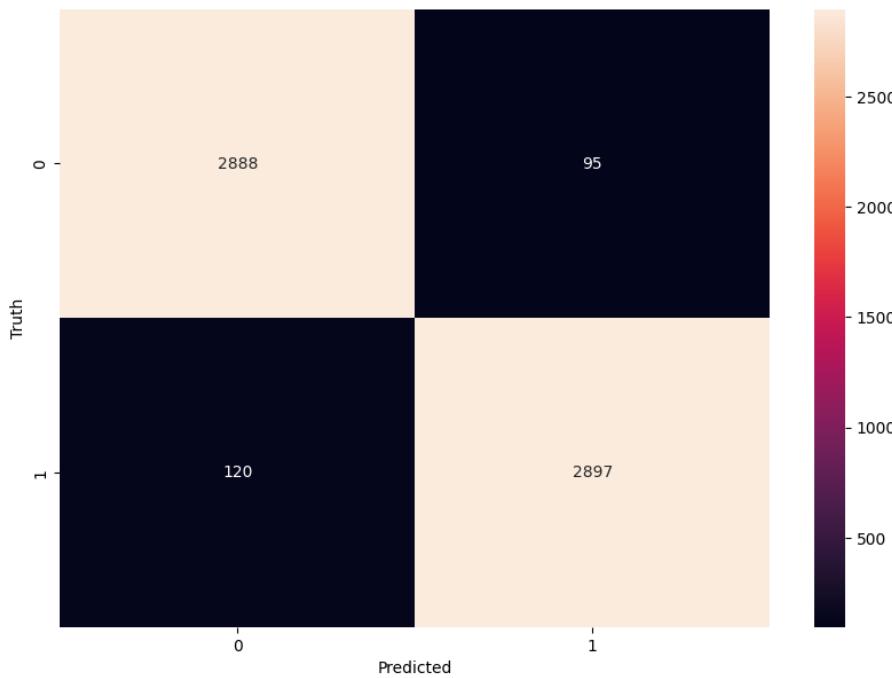


Figura 3.27: Mapa de calor de la clasificación con SVC de datos reales(1) y generados(0)

A continuación podemos ver el informe y el mapa de clasificación de la SVC que se ha usado para clasificar los datos generados. Con esto se buscaba que la clasificación en los clusters de los datos generados tuvieran una distribución parecida a la clasificación de los datos reales.

	Precision	Recall	F1-score	Support
1	1.00	1.00	1.00	2370
2	1.00	1.00	1.00	630
Accuracy			1.00	3000
Macro avg	1.00	1.00	1.00	3000
Weighted avg	1.00	1.00	1.00	3000

Cuadro 3.8: Clasificación datos reales

	Precision	Recall	F1-score	Support
0	1.00	1.00	1.00	2556
1	0.98	1.00	0.99	444
Accuracy			1.00	3000
Macro avg	0.99	1.00	0.99	3000
Weighted avg	1.00	1.00	1.00	3000

Cuadro 3.9: Clasificación datos generados

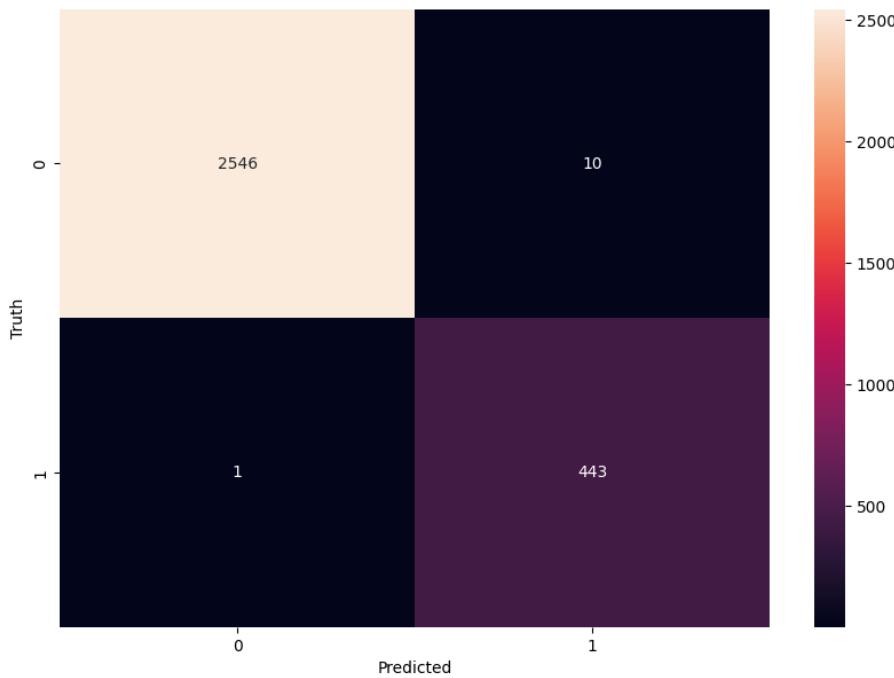


Figura 3.28: Mapa de calor de la clasificación con SVC de datos generados

Podemos observar cómo la distribución de los datos generados es parecida pero no idéntica a los datos reales. En los datos reales tenemos 2370 muestras de clase 1 y 630 muestras de clase 2. En los datos generados tenemos 2556 muestras de clase 0 (clase 1 en real) y 444 muestras de clase 1 (clase 2 en real). Además, hay 11 muestras mal clasificadas.

Como podemos ver en los siguientes dendogramas jerárquicos, la clasificación y jerarquía de los datos reales y generados es similar pero un poco diferente en cuanto jerarquía. Esto indica que el modelo no ha aprendido bien del todo la estructura subyacente de los datos.

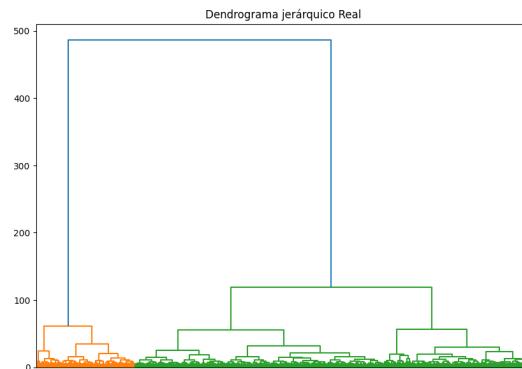


Figura 3.29: Dendograma jerárquico datos reales.

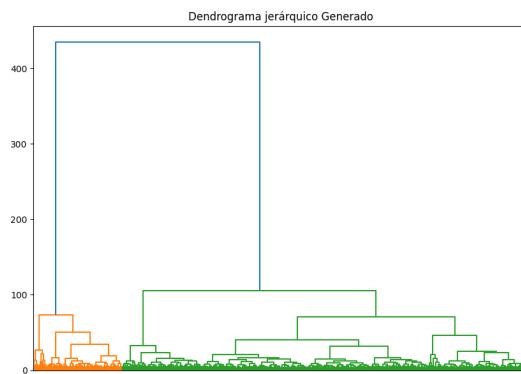


Figura 3.30: Dendograma jerarquico datos generados.

Deabajo podemos ver cómo usando un clasificador k-NN los datos reales se separan en dos clusters y los generados también, con algunos datos generados a medio camino.

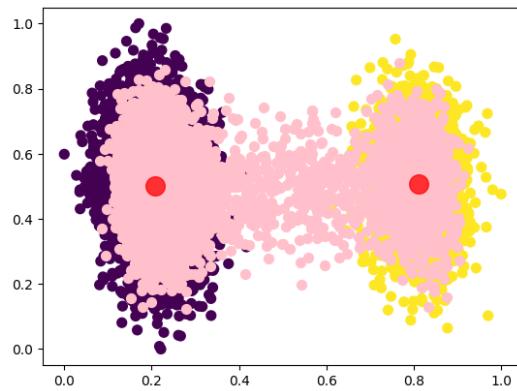


Figura 3.31: Clasificación usando K-NN. Amarillo y Azul datos reales. Rosa datos generados. Rojo centroides grupos

Como podemos ver en el siguiente diagrama del t-SNE los datos reales y generados también se separan en los dos clusters . Parece que el VAE también genera otro nuevo cluster pero con muchas mas muestras ahí..

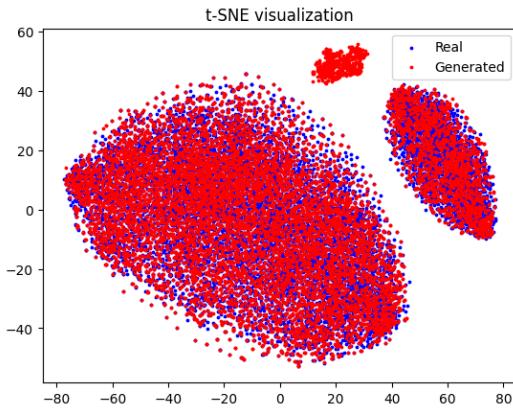


Figura 3.32: t-SNE

El UMAP nos muestra algo parecido.

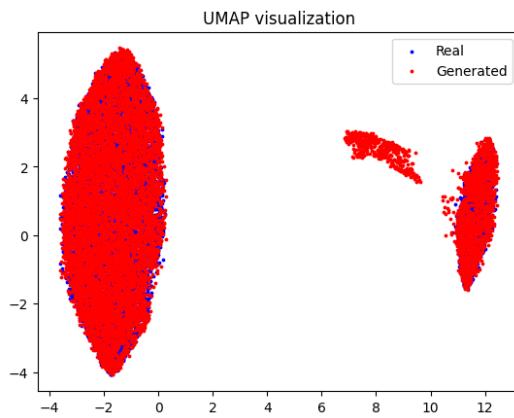


Figura 3.33: UMAP

En el PCA, los datos reales se separan en dos clusters y los generados también dejando algunos a medio camino entre los clusters. Pensamos que el cluster aislado visto en los gráficos anteriores debe ser debido a estas muestras que se quedan a medio camino.

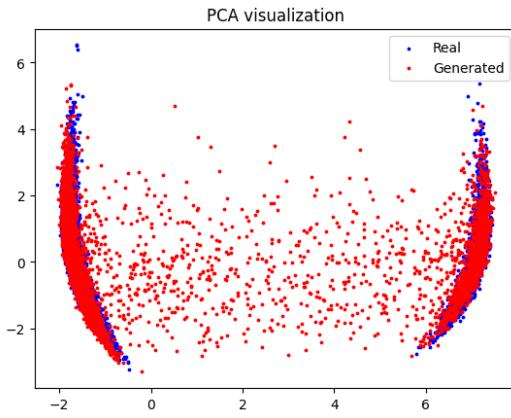


Figura 3.34: Principal Components Analysis

A continuación podemos ver los gráficos KDE para algunas variables al azar. Podemos observar como las distribuciones de los datos generados se asemejan mucho a la de los reales en esas variables.

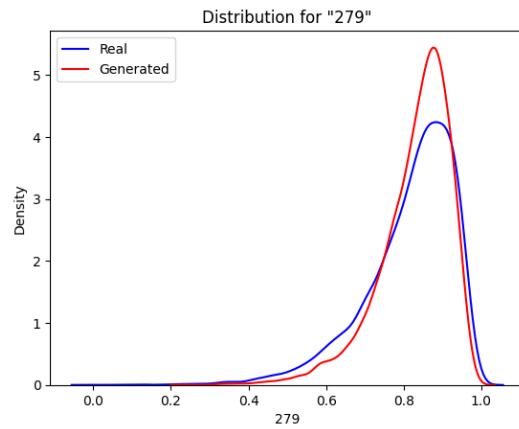


Figura 3.35: KDE

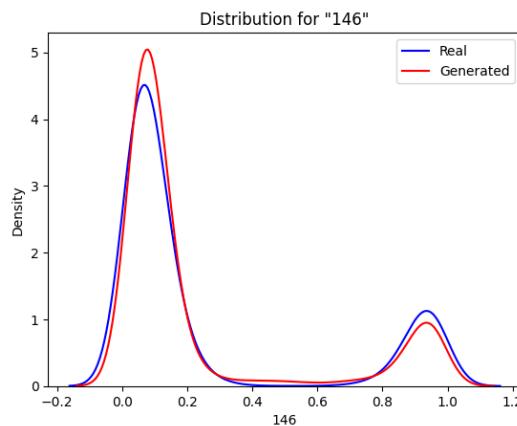


Figura 3.36: KDE

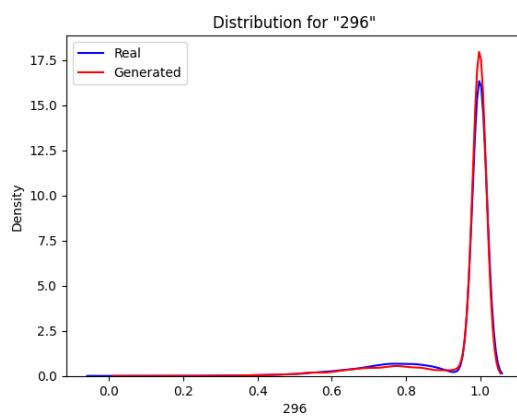


Figura 3.37: KDE

3.1.4. Modelo VAE Final con datos reales

En el modelo VAE Final con datos reales y todas las variables solo se pudo entrenar el modelo durante 6500 épocas, a diferencia de las 10000 de los datos simulados, por una cuestión de tiempo.

Se entrenaron también modelos con datos reales filtrando todas las variables con desviaciones estándar mayores al percentil 90 % durante 10000 épocas.

Se puede acceder al código y resultados de dichos modelos en:

<https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfal3utRY1si?usp=sharing>

Nombre de fichero: ModelVAEReALstd90.ipynb

En esta sección podemos ver los resultados del modelo entrenado durante 6500 épocas con los datos reales y todas las variables.

A continuación veremos los valores del modelo final de las métricas usadas para validar y comparar los modelos.

Wasserstein Distance: 0.12720828001893605
KS Statistic: 0.2260557530317857, P-Value: 0.0
Distancia Euclíadiana Promedio: 148.7584488313119

La distancia de Wasserstein es alta lo que indica que hay diferencias entre las dos distribuciones.

El estadístico del test Kolmogorov-Smirnov es muy alto, lo que sugiere que las diferencias entre las dos distribuciones son grandes. Además, el P-Valor es 0.0, lo que indica que estas pequeñas diferencias son estadísticamente significativas.

La distancia euclíadiana promedio no es comparable con la de los datos simulados por tener un mayor número de variables.

No se han podido calcular las matrices de correlación debido al gran número de variables

A continuación podemos ver el informe y el mapa de clasificación de la SVC que se ha usado para clasificar los datos reales y generados. En este mapa los datos generados son los de clase 0 y los reales de clase 1.

Cómo se ha explicado anteriormente esto nos da una medida de la calidad de los datos generados ya que usamos otro clasificador para medir cuánto se parecen los datos generados a los reales. Idealmente, el SVC no debería de poder distinguir entre datos reales y generados.

Cómo podemos ver la precisión es del 1. Lo ideal sería del 0.5 dado que indicaría que el SVC no puede distinguir entre reales y generados.

	Precision	Recall	F1-score	Support
0	1.00	1.00	1.00	9
1	1.00	1.00	1.00	14
Accuracy			1.00	23
Macro avg	1.00	1.00	1.00	23
Weighted avg	1.00	1.00	1.00	23

Cuadro 3.10: Clasificación datos reales y generados

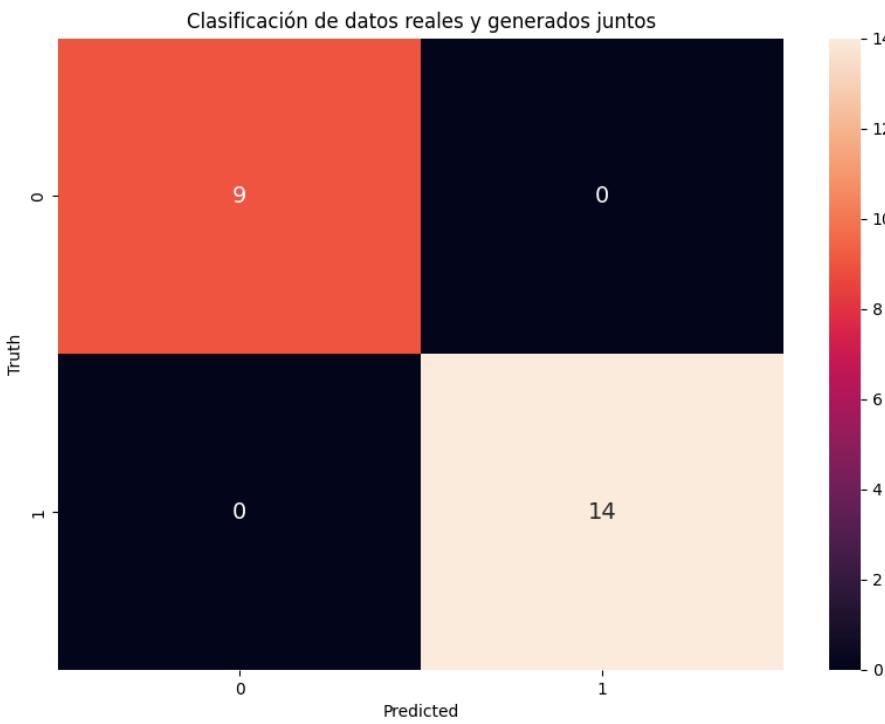


Figura 3.38: Mapa de calor de la clasificación con SVC de datos reales(1) y generados(0)

Los datos generados solo pertenecían a una clase así que no pudimos entrenar el clasificador SVC para comparar la distribución de la clasificación con los datos reales.

Como podemos ver en los siguientes dendogramas jerárquicos, la clasificación y jerarquía de los datos reales y generados es distinta. Esto indica que el modelo no ha aprendido bien la estructura subyacente de los datos.

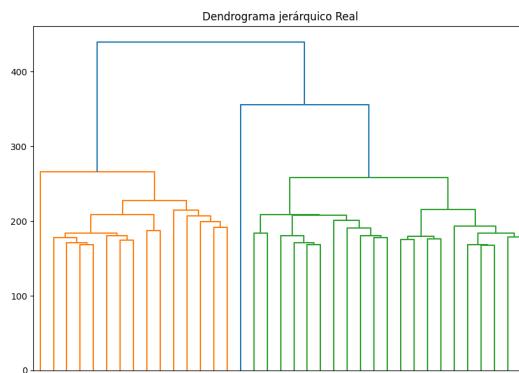


Figura 3.39: Dendograma jerarquico datos reales.

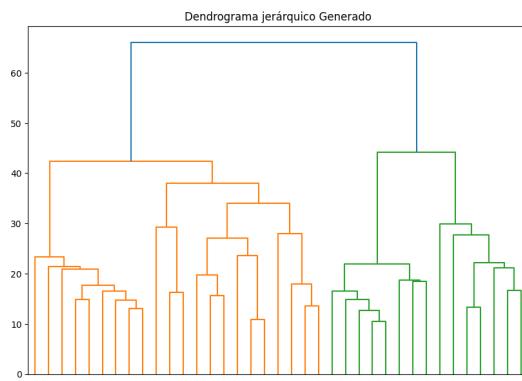


Figura 3.40: Dendrograma jerárquico datos generados.

Debajo podemos ver cómo usando un clasificador k-NN no consigue separar en dos clusters los datos reales y como los datos generados se reducen a una zona concreta.

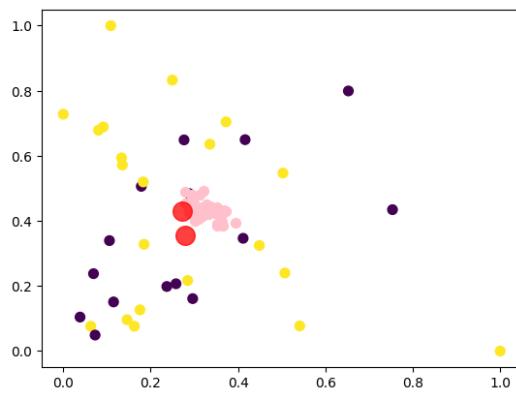


Figura 3.41: Clasificación usando K-NN. Amarillo y Azul datos reales. Rosa datos generados. Rojo centroides grupos

En este caso, igual que anteriormente en el GAN con datos reales, hemos generado otra vez 500 puntos para poder visualizar la generación de datos alrededor de los datos reales.

Como podemos ver en el siguiente diagrama del t-SNE los datos generados no coinciden con los reales pero parece que llenan el hueco que forman los reales.

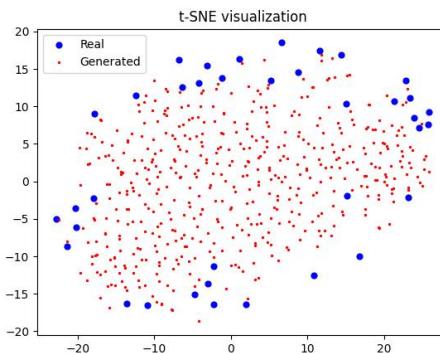


Figura 3.42: t-SNE

De igual forma el gráfico UMAP nos muestra algo parecido.

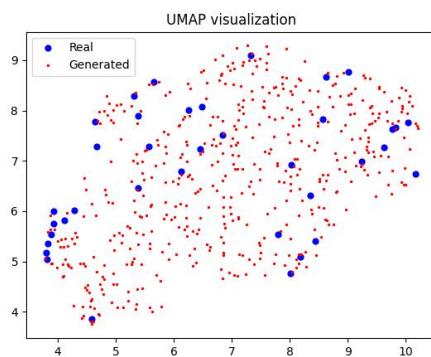


Figura 3.43: UMAP

En el PCA, podemos apreciar como los datos generados se reducen a una zona muy concreta y no se parecen en nada a los reales.

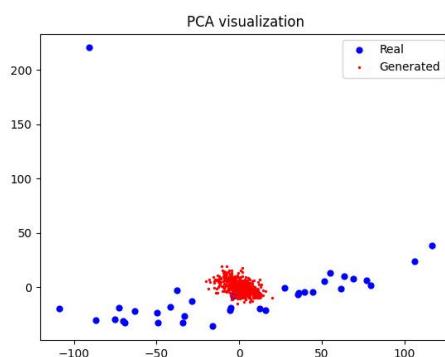


Figura 3.44: Principal Components Analysis

A continuación podemos ver los graficos KDE para algunas variables al azar. Podemos observar como las distribuciones de los datos generados no consiguen capturar el detalle de la de los reales.

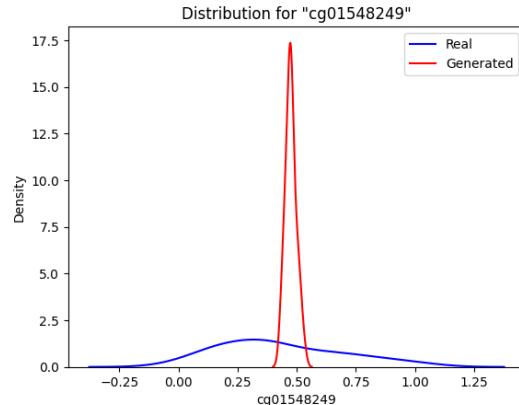


Figura 3.45: KDE

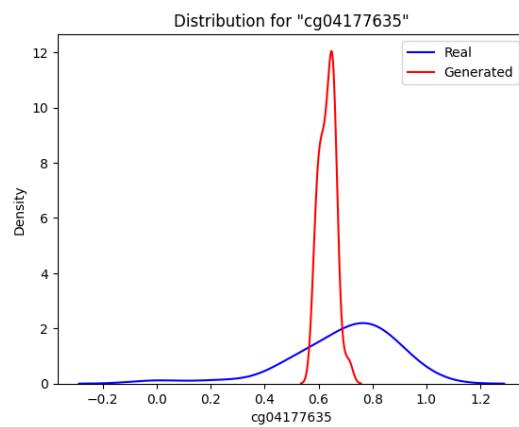


Figura 3.46: KDE

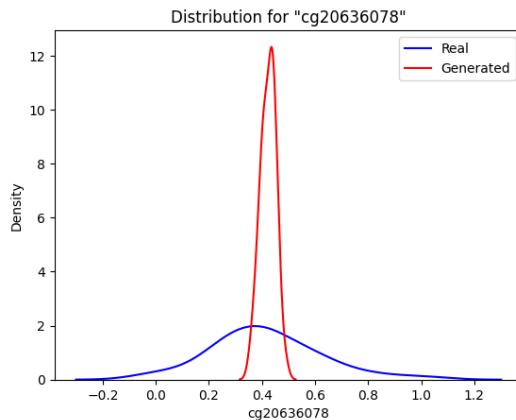


Figura 3.47: KDE

3.1.5. Comparación modelos GAN y VAE

Para esta comparación hemos usado los resultados con los datos simulados ya que con los datos reales el VAE tuvo un rendimiento muy pobre probablemente porque se tenía que haber adaptado el espacio latente a la gran dimensionalidad de los datos reales y por cuestión de tiempo no se pudo hacer.

En la siguiente tabla podemos ver la comparación de los dos modelos.

Métrica	Modelo GAN	Modelo VAE
Calidad de los Datos	9/10	8/10
Diversidad de la Generación	8/10	9/10
Distancia de Wasserstein	0.00262	0.00874
Tiempo de Entrenamiento	17h	6h
Estabilidad del Entrenamiento	9/10	9/10
Uso de Memoria	400 Mb	1800 Mb
Número de parámetros	4.393.460	27.073.987

Cuadro 3.11: Comparación entre Modelo GAN y Modelo VAE

3.2. Conversión datos ómicos

En este apartado se muestran los resultados de los modelos diseñados para convertir datos ómicos de expresión a metilación y viceversa

3.2.1. Modelos TRANSFORMER Conversión Expresión→Metilación

Debajo tenemos los resultados del modelo Transformer para la conversión de datos de expresión a metilación.

MSE: 0.011476878076791763

RMSE: 0.10713019218125096

PCC: 0.9505611883693352

Como podemos observar el MSE y el RMSE es muy bajo indicando que los datos de metilación convertidos a partir de datos de expresión se parecen mucho a los datos reales de metilación. Además el PCC es muy alto, indicando una fuerte correlación entre la metilación generada y la real.

Debajo podemos ver unos diagramas t-SNE, UMAP y PCA que nos muestran que los datos de metilación generados a partir de datos de expresión se agrupan de forma parecida aunque los datos generados no cubren a todos los datos reales. Esto indica que la GAN ha aprendido con bastante éxito, aunque no totalmente, la estructura subyacente y las relaciones complejas presentes en el conjunto de datos de metilación original.

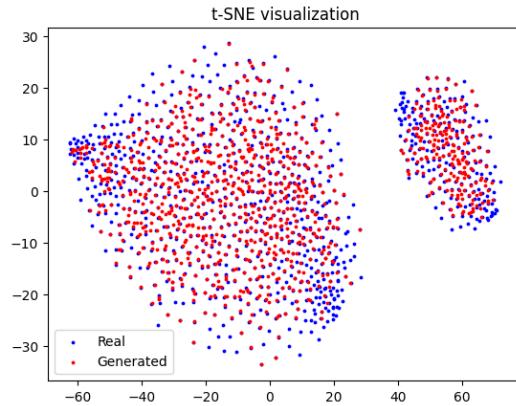


Figura 3.48: t-SNE

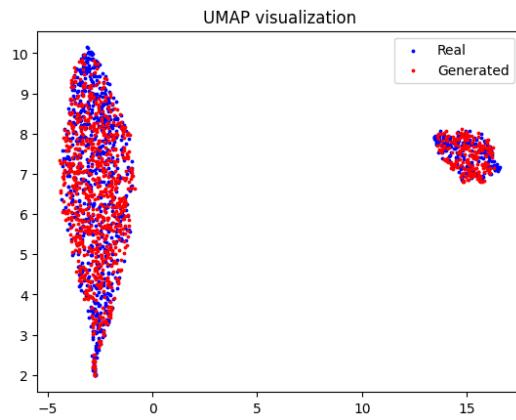


Figura 3.49: UMAP

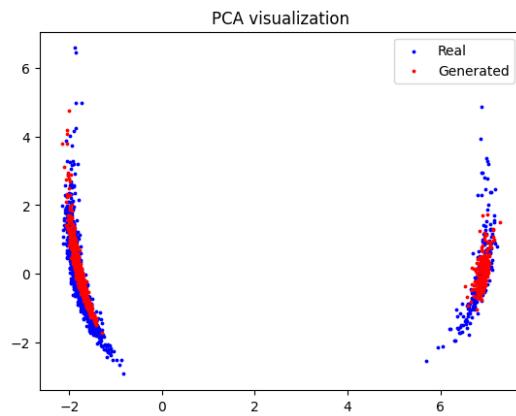


Figura 3.50: PCA

Se ha probado este modelo también con el conjunto de datos reales filtrando las variables con el SD en el percentil 90 % o superior.

Los resultados, como muestran las métricas y la figura de debajo, son bastante pobres. Esto posiblemente es debido al bajo número de muestras de entrenamiento.

MSE: 0.09385840594768524

RMSE: 0.30636319287356506

PCC: 0.1178213295327055

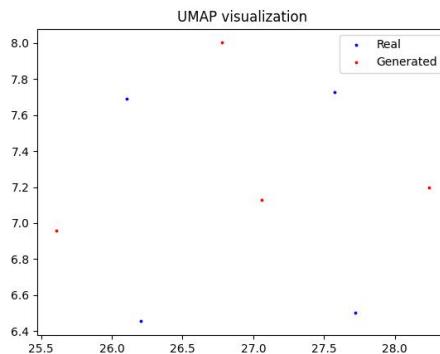


Figura 3.51: UMAP

3.2.2. Modelos GAN Conversión Expresión→Metilación

Debajo tenemos los resultados del modelo GAN para la conversión de datos de expresión a metilación.

MSE: 0.003932782914489508

RMSE: 0.062711904089172

PCC: 0.9833989615522315

Como podemos observar el MSE y el RMSE es muy bajo indicando que los datos de metilación convertidos a partir de datos de expresión se parecen mucho a los datos reales de metilación. Además el PCC es muy alto, indicando una fuerte correlación entre la metilación generada y la real.

Debajo podemos ver unos diagramas t-SNE, UMAP y PCA que nos muestran que los datos de metilación generados a partir de datos de expresión se agrupan de forma muy parecida. Esto indica que la GAN ha aprendido con éxito la estructura subyacente y las relaciones complejas presentes en el conjunto de datos de metilación original.

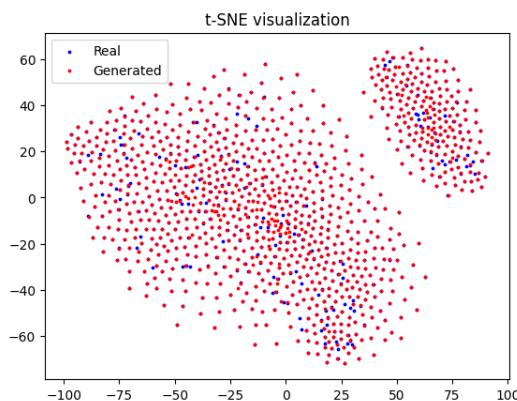


Figura 3.52: t-SNE

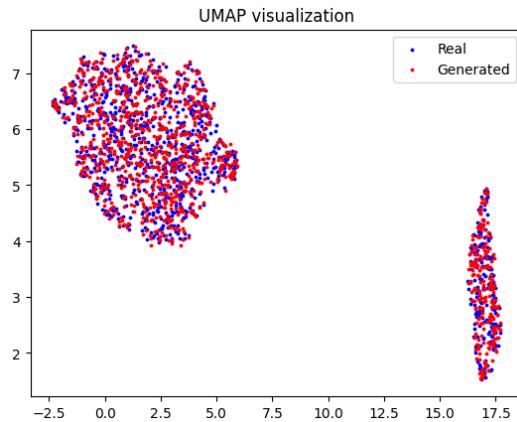


Figura 3.53: UMAP

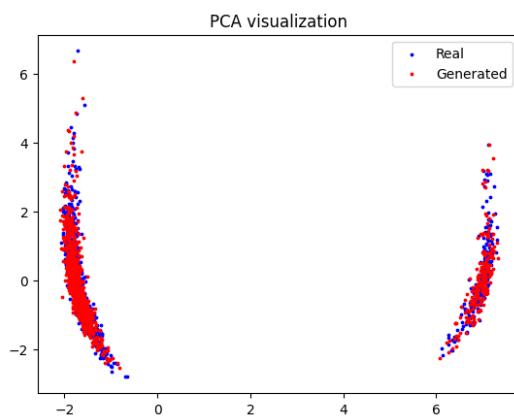


Figura 3.54: PCA

Podemos ver aquí como el modelo GAN supera al Transformer dando unos valores de RMSE y PCC mejores y mostrando una mejor adecuación a los datos reales en los gráficos.

Se ha probado este modelo también con el conjunto de datos reales filtrando las variables con el SD en el percentil 90 % o superior.

Los resultados, como muestran las métricas y la figura de debajo, son bastante pobres. Esto posiblemente es debido al bajo número de muestras de entrenamiento.

MSE: 0.08884239196777344

RMSE: 0.2980644090926883

PCC: 0.21142923371759942

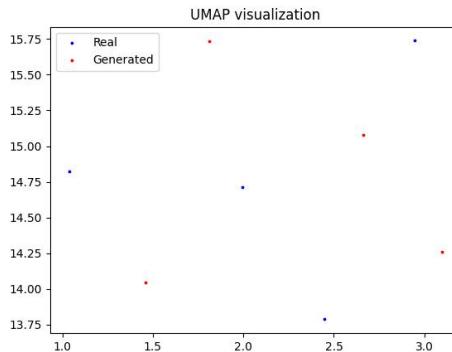


Figura 3.55: UMAP

3.2.3. Modelos TRANSFORMER Conversión Metilación→Expresión

Debajo tenemos los resultados del modelo Transformer para la conversión de datos de metilación a expresión.

MSE: 0.012659069150686264

RMSE: 0.1125125288609507

PCC: 0.8075531042446118

Como podemos observar el MSE y el RMSE es bajo (aunque no tanto como en la conversión contraria) indicando que los datos de expresión convertidos a partir de datos de metilación se parecen a los datos reales de expresión. Además el PCC es bastante alto, indicando una fuerte correlación entre la expresión generada y la real.

Debajo podemos ver unos diagramas t-SNE, UMAP y PCA que nos muestran que los datos de expresión generados a partir de datos de metilación se agrupan de forma muy parecida. Esto indica que la GAN ha aprendido con éxito la estructura subyacente y las relaciones complejas presentes en el conjunto de datos de expresión original.

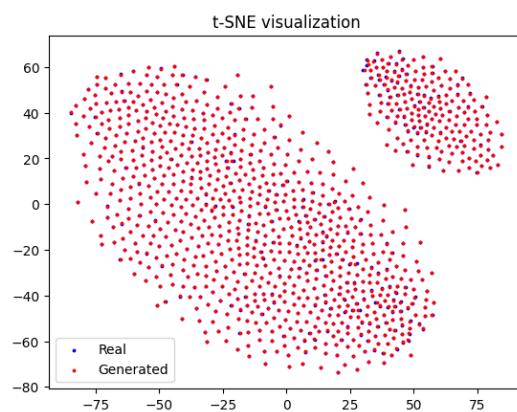


Figura 3.56: t-SNE

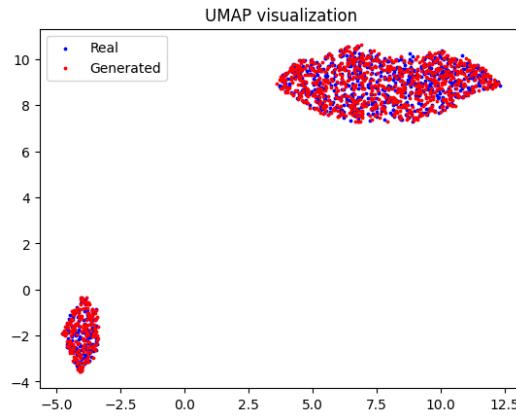


Figura 3.57: UMAP

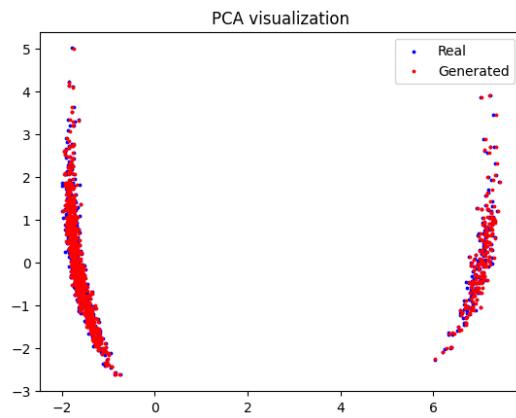


Figura 3.58: PCA

3.2.4. Modelos GAN Conversión Metilación→Expresión

Debajo tenemos los resultados del modelo GAN para la conversión de datos de metilación a expresión.

MSE : 0.020524505525827408

RMSE: 0.14326376208178887

PCC: 0.7032656399835951

Como podemos observar el MSE y el RMSE es bajo (aunque no tanto como en la conversión contraria) indicando que los datos de expresión convertidos a partir de datos de metilación se parecen a los datos reales de expresión. Además el PCC es bastante alto, indicando una fuerte correlación entre la expresión generada y la real.

Debajo podemos ver unos diagramas t-SNE, UMAP y PCA que nos muestran que los datos de expresión generados a partir de datos de metilación se agrupan de forma muy parecida. Esto indica que la GAN ha aprendido con éxito la estructura subyacente y las relaciones complejas presentes en el conjunto de datos de expresión original.

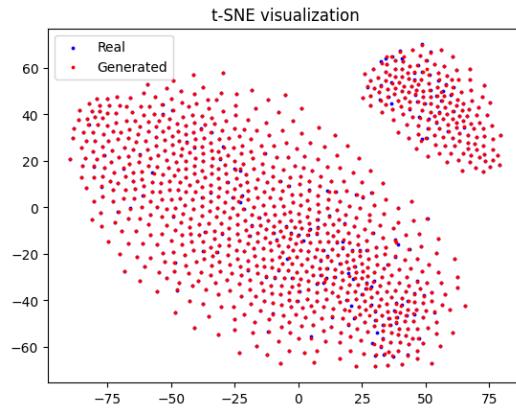


Figura 3.59: t-SNE

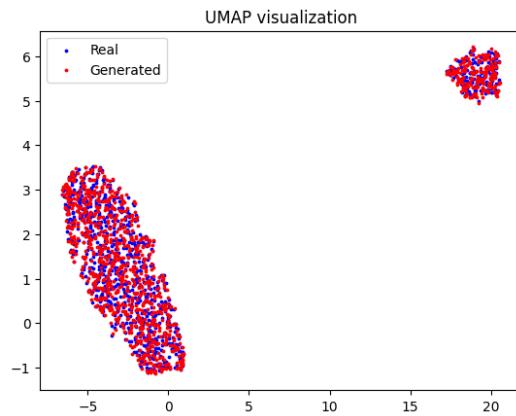


Figura 3.60: UMAP

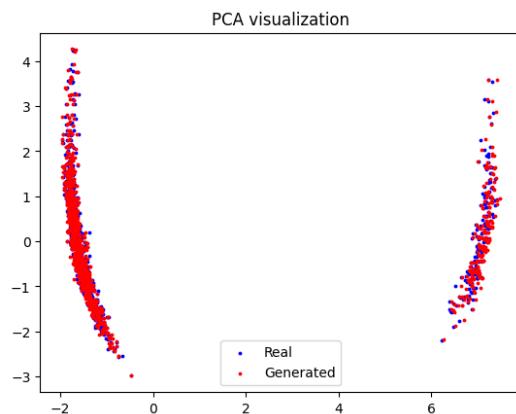


Figura 3.61: PCA

Este modelo de GAN parece tener peores resultados que el Transformer en la conversión de metilación a expresión. Al contrario de lo que pasaba en la conversión contraria.

3.2.5. Comparación modelos Transformer y GAN

En la siguientes tablas podemos ver la comparación de los dos modelos.

Para la conversión de expresión a metilación:

Métrica	Modelo GAN	Modelo Transformer
RMSE	0.0627	0.1071
PCC	0.9833	0.9505
Distribución datos convertidos	8/10	7/10
Tiempo de Entrenamiento	9h	0.2h
Estabilidad del Entrenamiento	9/10	9/10
Uso de Memoria	300 Mb	350 Mb
Número de parámetros	4.153.712	6.567.791

Cuadro 3.12: Comparación entre Modelo GAN y Modelo Transformer en conversión de expresión a metilación

Para la conversión de metilación a expresión:

Métrica	Modelo GAN	Modelo Transformer
RMSE	0.1432	0.1125
PCC	0.7032	0.8075
Distribución datos convertidos	9/10	9/10
Tiempo de Entrenamiento	13h	0.3h
Estabilidad del Entrenamiento	9/10	9/10
Uso de Memoria	450 Mb	650 Mb
Número de parámetros	13.328.516	25.717.891

Cuadro 3.13: Comparación entre Modelo GAN y Modelo Transformer en conversión de metilación a expresión

Capítulo 4

Conclusiones y trabajos futuros

En este estudio, se ha realizado una evaluación exhaustiva de la generación de datos sintéticos multi-ómicos utilizando Generative Adversarial Networks (GANs) y Variational Autoencoders (VAEs). Los resultados obtenidos han revelado que ambos métodos son herramientas viables, pero exhiben diferencias notables en términos de calidad y diversidad de los datos generados, así como en la eficiencia de su entrenamiento.

En el caso de los datos simulados, se observó que las GANs proporcionaban datos de mayor calidad, aunque con una diversidad algo limitada en comparación con las VAEs. Este aspecto contrasta con la menor demanda de memoria de las GANs durante el entrenamiento, a pesar de requerir más tiempo para esta tarea, un patrón opuesto al observado en las VAEs.

Cuando se aplicaron estos métodos a datos reales, las GANs demostraron una capacidad de aprendizaje superior. Este fenómeno podría explicarse por la necesidad de un ajuste más detallado del tamaño del espacio latente en los modelos VAE, para adaptarse a la complejidad y la gran cantidad de variables presentes en los datos reales.

De todos modos, quedó rápidamente de manifiesto que la baja cantidad de muestras de los datos reales era claramente insuficiente para el entrenamiento de estos modelos.

Una área particularmente interesante fue la conversión de datos ómicos de un dominio a otro. En este contexto, tanto las GANs como los Transformers mostraron resultados prometedores, siempre que se contara con una cantidad adecuada de datos de entrenamiento. Para los datos simulados, las GANs se destacaron en la conversión de expresión a metilación, mientras que los Transformers fueron más eficaces en la conversión inversa. Además, los Transformers conseguían un ajuste mucho más rápido que las GAN. Sin embargo, con datos reales limitados, los modelos enfrentaron desafíos significativos, impidiendo una conversión fiable.

Contrario a las expectativas iniciales, los resultados generales superaron las predicciones, especialmente en la precisión de la conversión de datos ómicos de expresión a metilación. Un aspecto sorprendente fue la capacidad de los modelos para realizar esta conversión con alta precisión. No obstante, no se lograron todos los objetivos planteados, particularmente en la mejora de los modelos GAN con el uso de Transformers. Esta situación se debió a que los modelos GANFORMER iniciales no proporcionaron los resultados esperados, lo que llevó a

un cambio en la dirección de la investigación.

La metodología original, de enfoque lineal, se adaptó a un proceso más iterativo y flexible, permitiendo la incorporación de mejoras incrementales y la repetición de ciertos pasos del proceso investigador. Este enfoque más dinámico fue fundamental para adaptarse a los hallazgos emergentes y a las necesidades cambiantes del proyecto.

En cuanto a los impactos en sostenibilidad se ha conseguido restringir el consumo energético no haciendo usos de servidores y computadores de alto rendimiento.

El resto de impactos negativos (sesgo en datos de entrenamiento, diferenciación adecuada de los datos sintéticos) no ha podido mitigarse. Y los impactos positivos no son posible evaluarlos en este momento.

De cara al futuro, se identifican varias áreas prometedoras para la investigación y mejoras. Estas incluyen la automatización en la exploración de modelos e hiperparámetros, la exploración del Cycle GAN para la generación de datos sintéticos, el entrenamiento de modelos VAE con un espacio latente ampliado utilizando datos reales, una comprensión más profunda de la arquitectura Transformer, la mejora y exploración adicional del modelo GANFORMER, y el entrenamiento de modelos de generación y de conversión con un mayor volumen de datos reales.

En resumen, este trabajo ha proporcionado al autor insights valiosos sobre la generación y conversión de datos sintéticos multi-ómicos, aportando conocimiento y experiencia para futuras investigaciones en este campo.

Glosario

1-Lipschitz Una propiedad matemática de funciones que garantiza estabilidad en GANs.. 21, 22

datos sintéticos Datos generados artificialmente, a menudo utilizados para entrenamiento de modelos de IA.. 3, 7, 10, 15–18

dendogramas jerárquicos Gráfico usado en análisis de clustering jerárquico.. 36, 39, 42, 56, 63, 70, 76

Discriminador En contextos de GAN, es la red que diferencia entre datos reales y generados.. 18–23, 35, 37, 38, 43, 48, 51

distancia de Wasserstein Una métrica para medir la distancia entre dos distribuciones de probabilidad.. 20, 36, 39, 42, 53, 61, 75

distancia euclíadiana promedio Promedio de distancias euclidianas entre puntos en espacios vectoriales.. 36, 39, 42, 53, 61, 75

espacio latente Espacio de representación de alta dimensionalidad en modelos generativos.. 23–25, 80

GAN Generative Adversarial Network: un modelo de redes neuronales para la generación de datos.. 7, 11–15, 17–20, 22, 35, 36, 44, 45, 47, 50, 60, 83, 84, 86

Generador En contextos de GAN, es la red que genera nuevos datos.. 18, 37, 43, 44, 48, 51

k-NN k-Nearest Neighbors: algoritmo de clasificación basado en vecindad.. 36, 39, 42, 57, 64, 71, 77

KDE Kernel Density Estimation: método para estimar la función de densidad de una variable aleatoria.. 36, 39, 42, 59, 66, 73, 79

matrices de correlación Matriz que representa las correlaciones entre variables.. 31, 33, 36, 39, 42, 53, 61, 67, 75

MSE Mean Squared Error: error cuadrático medio, una medida de calidad en modelos de regresión.. 45, 51

multi-ómicos Conjunto de datos ómicos pertenecientes a diferentes campos.. 3, 7–15, 17, 18

norma euclíadiana Una medida de longitud en espacios vectoriales.. 22, 23

PCA Principal Component Analysis: técnica estadística para reducción de dimensionalidad.. 36, 39, 42, 45, 52, 58, 65, 72, 78

PCC Pearson Correlation Coefficient: coeficiente de correlación de Pearson para medir la correlación lineal entre variables.. 45, 51, 84

penalización de gradiente Técnica en WGAN-GP para mejorar la estabilidad del entrenamiento.. 16, 22, 23, 35, 37, 48, 51

recorte de pesos Técnica en WGAN para mantener las funciones 1-Lipschitz.. 22, 41

RMSE Root Mean Squared Error: raíz del error cuadrático medio.. 45, 51, 84

sintéticos Datos generados artificialmente, a menudo utilizados para entrenamiento de modelos de IA.. 11

SVC Support Vector Classifier: clasificador basado en máquinas de vectores de soporte.. 35, 36, 39, 42, 54, 55, 61, 62, 68, 69, 75

t-SNE t-distributed Stochastic Neighbor Embedding: técnica de reducción de dimensionalidad.. 36, 39, 42, 45, 51, 57, 64, 71, 77

test Kolmogorov-Smirnov Prueba estadística para comparar distribuciones de probabilidad.. 36, 39, 42, 53, 61, 75

Transformer Modelo de atención que ha revolucionado el procesamiento del lenguaje natural.. 3, 4, 8–15, 25–27, 43–46, 48, 81, 84, 85

UMAP Uniform Manifold Approximation and Projection: técnica de reducción de dimensionalidad.. 36, 39, 42, 45, 52, 58, 65, 72, 78

VAE Variational Autoencoder: un tipo de red auto-codificadora para la generación y modelado de datos.. 23, 24, 39, 42, 71, 74, 80

WGAN Wasserstein Generative Adversarial Network: una variante de GAN que mejora la estabilidad del entrenamiento.. 20–22

WGAN-GP WGAN with Gradient Penalty: mejora de WGAN con penalización de gradiente para estabilidad en el entrenamiento.. 22, 35

ómicos Datos relacionados con campos como la genómica, proteómica, transcriptómica, entre otros.. 8, 9, 15, 17, 18, 32, 44, 45

Bibliografía

- [1] Ahmed, K. T., Sun, J., Cheng, S., Yong, J., and Zhang, W. (2021). Multi-omics data integration by generative adversarial network. *Bioinformatics*, 38(1):179–186.
- [2] altexsoft (2022). Omics Data Analysis and Integration: How and Where to Use Multi-Omics Data — altexsoft.com. <https://www.altexsoft.com/blog/omics-data-analysis/>. [Accessed 12-10-2023].
- [3] Antwerpen, A.-M. (2023). Generative Adversarial Networks: How Data Can Be Generated With Neural Networks — statworx.com. <https://www.statworx.com/en/content-hub/blog/generative-adversarial-networks-how-data-can-be-generated-with-neural-networks/>. [Accessed 24-12-2023].
- [4] Arjovsky, M. and Bottou, L. (2017). Towards principled methods for training generative adversarial networks.
- [5] Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein generative adversarial networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223. PMLR.
- [6] Borji, A. (2021). Pros and cons of gan evaluation measures: New developments.
- [7] Brock, A., Donahue, J., and Simonyan, K. (2019). Large scale gan training for high fidelity natural image synthesis.
- [8] Brombacher, E., Hackenberg, M., Kreutz, C., Binder, H., and Treppner, M. (2022). The performance of deep generative models for learning joint embeddings of single-cell multi-omics data. *Frontiers in Molecular Biosciences*, 9.
- [9] Choi, S. R. and Lee, M. (2023). Transformer architecture and attention mechanisms in genome data analysis: A comprehensive review. *Biology (Basel)*, 12(7).
- [10] Chong, M. J. and Forsyth, D. (2021). Gans n' roses: Stable, controllable, diverse image to image translation (works for videos too!).
- [11] Dubey, S. R. and Singh, S. K. (2023). Transformer-based generative adversarial networks in computer vision: A comprehensive survey.

- [12] Figueira, A. and Vaz, B. (2022). Survey on synthetic data generation, evaluation methods and gans. *Mathematics*, 10(15).
- [13] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks.
- [14] Google (2023). Machine learning course. <https://developers.google.com/machine-learning/gan/loss>. [Accessed 02-01-2024].
- [15] Guan, S. and Loew, M. (2021). A novel measure to evaluate generative adversarial networks based on direct analysis of generated images. *Neural Computing and Applications*, 33(20):13921–13936.
- [16] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017). Improved training of wasserstein gans.
- [17] Hess, M., Hackenberg, M., and Binder, H. (2020). Exploring generative deep learning for omics data using log-linear models. *Bioinformatics*, 36(20):5045–5053.
- [18] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. (2018). Gans trained by a two time-scale update rule converge to a local nash equilibrium.
- [19] Hudson, D. A. and Zitnick, C. L. (2021). Generative adversarial transformers.
- [20] Jiang, Y., Alford, K., Ketchum, F., Tong, L., and Wang, M. D. (2020). Tlsurv: Integrating multi-omics data by multi-stage transfer learning for cancer survival prediction. In *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB ’20, New York, NY, USA. Association for Computing Machinery.
- [21] Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2018). Progressive growing of gans for improved quality, stability, and variation.
- [22] Lee, M. (2023). Recent advances in generative adversarial networks for gene expression data: A comprehensive review. *Mathematics*, 11(14).
- [23] Leng, D., Zheng, L., Wen, Y., Zhang, Y., Wu, L., Wang, J., Wang, M., Zhang, Z., He, S., and Bo, X. (2022). A benchmark study of deep learning-based multi-omics data fusion methods for cancer. *Genome Biology*, 23(1):171.
- [24] Li, X., Ngu, A. H. H., and Metsis, V. (2022). TTS-CGAN: A transformer Time-Series conditional GAN for biosignal data augmentation.
- [25] Lucic, M., Kurach, K., Michalski, M., Gelly, S., and Bousquet, O. (2018). Are gans created equal? a large-scale study.
- [26] Martorell-Marugán, J., López-Domínguez, R., García-Moreno, A., Toro-Domínguez, D., Villatoro-García, J. A., Barturen, G., Martín-Gómez, A., Troule, K., Gómez-López, G., Al-Shahrour, F., González-Rumayor, V., Peña-Chilet, M., Dopazo, J., Sáez-Rodríguez, J.,

Alarcón-Riquelme, M. E., and Carmona-Sáez, P. (2021). A comprehensive database for integrated analysis of omics data in autoimmune diseases. *BMC Bioinformatics*, 22(1):343.

- [27] Mescheder, L., Geiger, A., and Nowozin, S. (2018). Which training methods for gans do actually converge?
- [28] Miyato, T., Kataoka, T., Koyama, M., and Yoshida, Y. (2018). Spectral normalization for generative adversarial networks.
- [29] Moon, S. and Lee, H. (2022). MOMA: a multi-task attention learning algorithm for multi-omics data interpretation and classification. *Bioinformatics*, 38(8):2287–2296.
- [30] Nassiri, K. and Akhloufi, M. (2022). Transformer models used for text-based question answering systems. *Applied Intelligence*, 53.
- [31] Poinsignon, T., Poulain, P., Gallopin, M., and Lelandais, G. (2023). *Working with Omics Data: An Interdisciplinary Challenge at the Crossroads of Biology and Computer Science*, pages 313–330. Springer US, New York, NY.
- [32] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved techniques for training gans.
- [33] Shende, R. (2023). Autoencoders, Variational Autoencoders (VAE) and β -VAE — rushikesh.shende. <https://medium.com/@rushikesh.shende/autoencoders-variational-autoencoders-vae-and-%CE%B2-vae-ceba9998773d>. [Accessed 24-12-2023].
- [34] Shi, M., Li, X., Li, M., and Si, Y. (2023). Attention-based generative adversarial networks improve prognostic outcome prediction of cancer from multimodal data. *Briefings in Bioinformatics*, 24(6):bbad329.
- [35] Subramanian, I., Verma, S., Kumar, S., Jere, A., and Anamika, K. (2020). Multi-omics data integration, interpretation, and its application. *Bioinform. Biol. Insights*, 14:1177932219899051.
- [36] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.
- [37] Viñas, R., Andrés-Terré, H., Liò, P., and Bryson, K. (2021). Adversarial generation of gene expression data. *Bioinformatics*, 38(3):730–737.
- [38] Xu, R., Xu, X., Chen, K., Zhou, B., and Loy, C. C. (2021). The nuts and bolts of adopting transformer in GANs.

Capítulo 5

Anexos

5.1. Anexo A. Requisitos entorno Python

Lista de requisitos del entorno Python:

```
absl-py==2.0.0
anyio==4.0.0
argon2-cffi==23.1.0
argon2-cffi-bindings==21.2.0
arrow==1.2.3
asttokens==2.4.0
astunparse==1.6.3
async-lru==2.0.4
attrs==23.1.0
Babel==2.12.1
backcall==0.2.0
beautifulsoup4==4.12.2
biopython==1.81
bleach==6.0.0
cachetools==5.3.1
certifi==2023.7.22
cffi==1.15.1
charset-normalizer==3.2.0
comm==0.1.4
contourpy==1.1.1
cycler==0.12.1
debugpy==1.8.0
decorator==5.1.1
defusedxml==0.7.1
executing==1.2.0
fastjsonschema==2.18.0
```

```
filelock==3.12.4
flatbuffers==23.5.26
fonttools==4.43.1
fqdn==1.5.1
fsspec==2023.9.2
gast==0.5.4
google-auth==2.23.2
google-auth-oauthlib==1.0.0
google-pasta==0.2.0
graphviz==0.20.1
grpcio==1.58.0
h5py==3.9.0
hiddenlayer==0.3
idna==3.4
iniconfig==2.0.0
ipykernel==6.25.2
ipytest==0.13.3
ipython==8.15.0
isoduration==20.11.0
jedi==0.19.0
Jinja2==3.1.2
joblib==1.3.2
json5==0.9.14
jsonpointer==2.4
jsonschema==4.19.1
jsonschema-specifications==2023.7.1
jupyter-events==0.7.0
jupyter-lsp==2.2.0
jupyter_client==8.3.1
jupyter_core==5.3.1
jupyter_server==2.7.3
jupyter_server_terminals==0.4.4
jupyterlab==4.0.6
jupyterlab-pygments==0.2.2
jupyterlab_server==2.25.0
keras==2.14.0
kiwisolver==1.4.5
libclang==16.0.6
llvmlite==0.41.1
Markdown==3.4.4
MarkupSafe==2.1.3
matplotlib==3.8.0
matplotlib-inline==0.1.6
mistune==3.0.1
ml-dtypes==0.2.0
```

```
mpmath==1.3.0
nbclient==0.8.0
nbconvert==7.8.0
nbformat==5.9.2
nest-asyncio==1.5.8
networkx==3.1
notebook==7.0.4
notebook_shim==0.2.3
numba==0.58.1
numpy==1.26.0
nvidia-cublas-cu12==12.1.3.1
nvidia-cuda-cupti-cu12==12.1.105
nvidia-cuda-nvrtc-cu12==12.1.105
nvidia-cuda-runtime-cu12==12.1.105
nvidia-cudnn-cu12==8.9.2.26
nvidia-cufft-cu12==11.0.2.54
nvidia-curand-cu12==10.3.2.106
nvidia-cusolver-cu12==11.4.5.107
nvidia-cusparse-cu12==12.1.0.106
nvidia-nccl-cu12==2.18.1
nvidia-nvjitlink-cu12==12.2.140
nvidia-nvtx-cu12==12.1.105
oauthlib==3.2.2
opt-einsum==3.3.0
overrides==7.4.0
packaging==23.1
pandas==2.1.1
pandocfilters==1.5.0
parso==0.8.3
pexpect==4.8.0
pickleshare==0.7.5
Pillow==10.0.1
platformdirs==3.10.0
pluggy==1.3.0
prometheus-client==0.17.1
prompt-toolkit==3.0.39
protobuf==4.24.3
psutil==5.9.5
ptyprocess==0.7.0
pure-eval==0.2.2
pyasn1==0.5.0
pyasn1-modules==0.3.0
pycparser==2.21
Pygments==2.16.1
pynndescent==0.5.11
```

```
pyparsing==3.1.1
pytest==7.4.2
python-dateutil==2.8.2
python-json-logger==2.0.7
pytz==2023.3.post1
PyYAML==6.0.1
pyzmq==25.1.1
referencing==0.30.2
requests==2.31.0
requests-oauthlib==1.3.1
rfc3339-validator==0.1.4
rfc3986-validator==0.1.1
rpds-py==0.10.3
rsa==4.9
scikit-learn==1.3.1
scipy==1.11.3
seaborn==0.13.0
Send2Trash==1.8.2
six==1.16.0
sklearn==0.0.post10
sniffio==1.3.0
soupsieve==2.5
stack-data==0.6.2
sympy==1.12
tensorboard==2.14.1
tensorboard-data-server==0.7.1
tensorflow==2.14.0
tensorflow-estimator==2.14.0
tensorflow-io-gcs-filesystem==0.34.0
termcolor==2.3.0
terminado==0.17.1
threadpoolctl==3.2.0
tinyrss==1.2.1
torch==2.1.0
torchvision==0.16.0
torchviz==0.0.2
tornado==6.3.3
tqdm==4.66.1
traitlets==5.10.1
triton==2.1.0
typing_extensions==4.8.0
tzdata==2023.3
umap-learn==0.5.5
uri-template==1.3.0
urllib3==2.0.5
```

```

wcwidth==0.2.6
webcolors==1.13
webencodings==0.5.1
websocket-client==1.6.3
Werkzeug==2.3.7
wrapt==1.14.1
xlrd==2.0.1

```

5.2. Anexo B. Análisis estadísticos de los datos

5.2.1. Análisis estadístico de los datos simulados

Debajo podemos ver un pequeño análisis estadístico de los datos generados una vez preprocesados.

Descripción de los datos de expresión génica:						
	0	1	2	3	4	\
↪000000	count	10000.000000	10000.000000	10000.000000	10000.000000	10000.
↪491423	mean	0.328066	0.503368	0.659170	0.507475	0.
↪129720	std	0.247159	0.135970	0.218681	0.134313	0.
↪000000	min	0.000000	0.000000	0.000000	0.000000	0.
↪405544	25 %	0.181019	0.411101	0.660420	0.415131	0.
↪491109	50 %	0.223655	0.502669	0.741512	0.506127	0.
↪577160	75 %	0.287407	0.595957	0.795608	0.599708	0.
↪000000	max	1.000000	1.000000	1.000000	1.000000	1.
	5	6	7	8	9	\
↪000000	count	10000.000000	10000.000000	10000.000000	10000.000000	10000.
↪688998	mean	0.515437	0.538387	0.372585	0.477964	0.
↪248348	std	0.139351	0.118994	0.201184	0.128374	0.
↪000000	min	0.000000	0.000000	0.000000	0.000000	0.
↪722007	25 %	0.419519	0.457670	0.244740	0.390067	0.
↪791539	50 %	0.515467	0.537833	0.298861	0.475744	0.
↪837835	75 %	0.610432	0.618913	0.378159	0.565544	0.

```

max      1.000000      1.000000      1.000000      1.000000      1.
↪000000

...      121      122      123      124 \
count   ... 10000.000000 10000.000000 10000.000000 10000.000000
mean    ... 0.477599 0.636008 0.490606 0.529590
std     ... 0.133783 0.202083 0.128140 0.119684
min     ... 0.000000 0.000000 0.000000 0.000000
25%    ... 0.386852 0.617451 0.403134 0.448619
50%    ... 0.478629 0.706271 0.490642 0.529872
75%    ... 0.566472 0.766722 0.577844 0.609389
max    ... 1.000000 1.000000 1.000000 1.000000

125      126      127      128      129 \
↪000000 count   10000.000000 10000.000000 10000.000000 10000.000000 10000.
mean    0.330869 0.463462 0.508159 0.479401 0.
↪515511 std     0.253633 0.129912 0.129188 0.139450 0.
↪127172 min     0.000000 0.000000 0.000000 0.000000 0.
↪000000 25%    0.180438 0.374595 0.420580 0.384403 0.
↪429902 50%    0.223848 0.462896 0.506870 0.476825 0.
↪515455 75%    0.290139 0.552293 0.595583 0.572998 0.
↪601939 max     1.000000 1.000000 1.000000 1.000000 1.
↪000000

130
↪000000 count   10000.000000
mean    0.493513
std     0.114537
min     0.000000
25%    0.415434
50%    0.494622
75%    0.570624
max    1.000000

[8 rows x 131 columns]

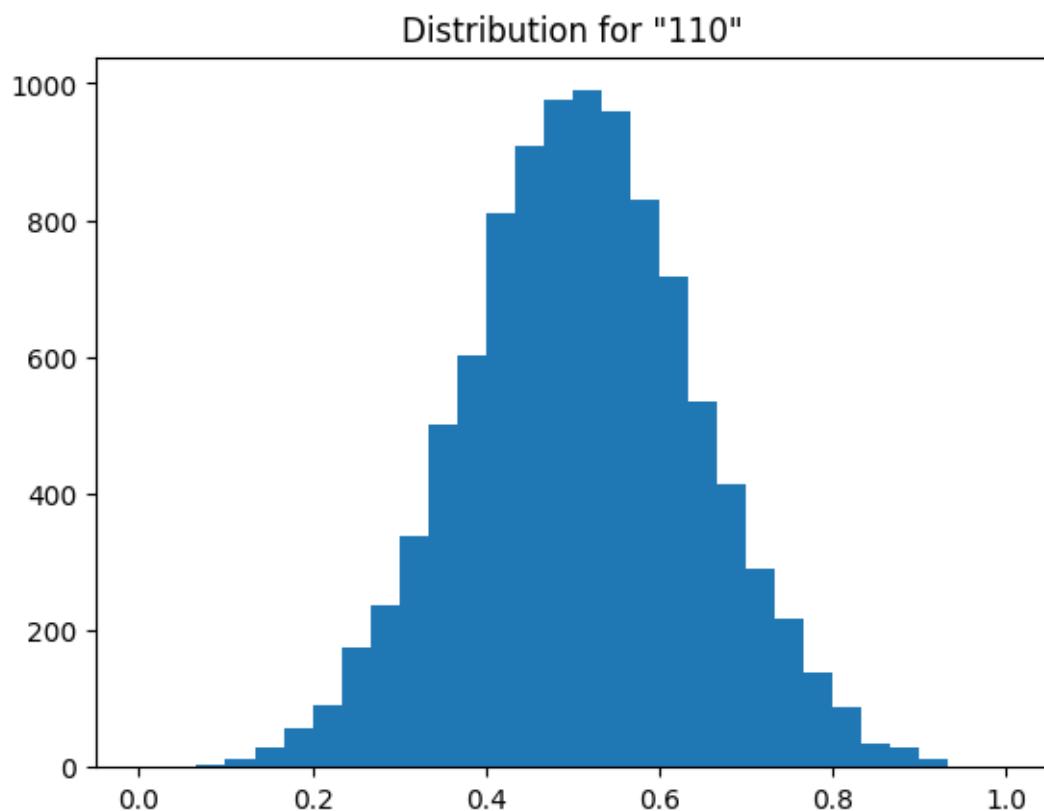
Descripción de los datos de metilación:
0      1      2      3      4 \
↪000000 count   10000.000000 10000.000000 10000.000000 10000.000000 10000.
mean   0.204762 0.159800 0.540466 0.790593 0.
↪362466 std     0.340618 0.107479 0.213204 0.352868 0.
↪160532 min     0.000000 0.000000 0.000000 0.000000 0.
↪000000
```

↪242640	25 %	0.025442	0.085479	0.376603	0.943070	0.
↪344396	50 %	0.037265	0.134154	0.546332	0.963836	0.
↪468068	75 %	0.063757	0.204007	0.709760	0.974830	0.
↪000000	max	1.000000	1.000000	1.000000	1.000000	1.
	5	6	7	8	9	\
↪000000	count	10000.000000	10000.000000	10000.000000	10000.000000	10000.
↪391542	mean	0.114827	0.493113	0.159022	0.719423	0.
↪314377	std	0.094541	0.187540	0.109058	0.137443	0.
↪000000	min	0.000000	0.000000	0.000000	0.000000	0.
↪163817	25 %	0.051546	0.355795	0.082575	0.640891	0.
↪268020	50 %	0.088427	0.492609	0.131204	0.740254	0.
↪477743	75 %	0.148801	0.631841	0.206913	0.820229	0.
↪000000	max	1.000000	1.000000	1.000000	1.000000	1.
	...	357	358	359	360	\
↪000000	count	... 10000.000000	10000.000000	10000.000000	10000.000000	
↪152342	mean	0.217216	0.158452	0.160549	0.060219	
↪122368	std	0.348284	0.104495	0.093111	0.066537	
↪000000	min	0.000000	0.000000	0.000000	0.000000	
↪066496	25 %	0.031505	0.083870	0.093557	0.020502	
↪118313	50 %	0.046667	0.133887	0.141188	0.039502	
↪199280	75 %	0.078439	0.206679	0.206658	0.075322	
	max	... 1.000000	1.000000	1.000000	1.000000	
	361	362	363	364	365	\
↪000000	count	10000.000000	10000.000000	10000.000000	10000.000000	10000.
↪152342	mean	0.125866	0.709152	0.149968	0.231109	0.
↪122368	std	0.096891	0.113650	0.103256	0.112522	0.
↪000000	min	0.000000	0.000000	0.000000	0.000000	0.
↪066496	25 %	0.059500	0.638298	0.077298	0.149771	0.
↪118313	50 %	0.100609	0.719866	0.125035	0.212479	0.
↪199280	75 %	0.163395	0.790180	0.193647	0.292032	0.

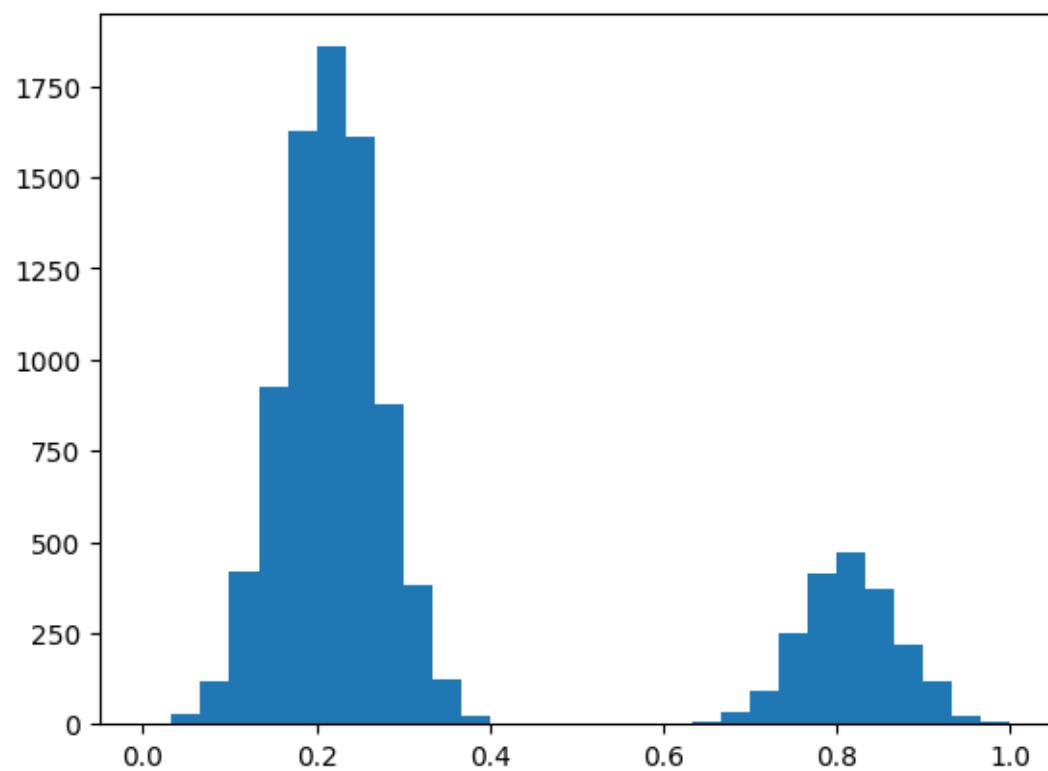
```
max      1.000000      1.000000      1.000000      1.000000      1.  
↪000000  
  
366  
count  10000.000000  
mean   0.173442  
std    0.104391  
min    0.000000  
25 %   0.098803  
50 %   0.150882  
75 %   0.223022  
max    1.000000
```

[8 rows x 367 columns]

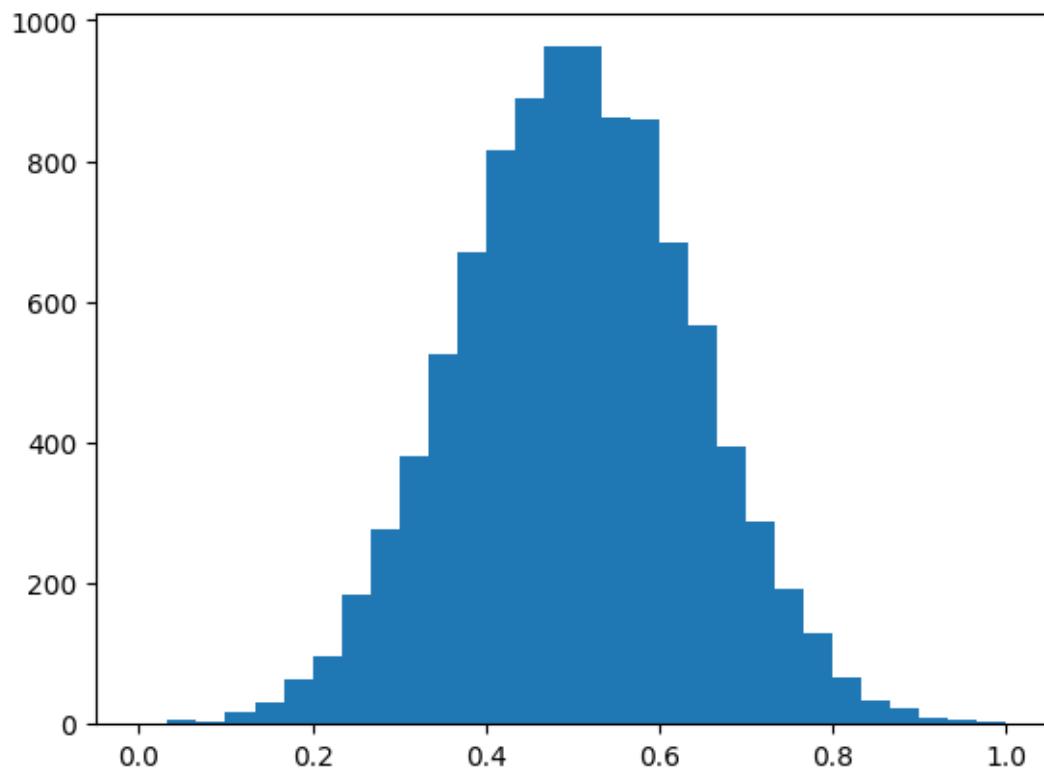
Histogramas de 5 variables al azar de los datos de expresión génica:



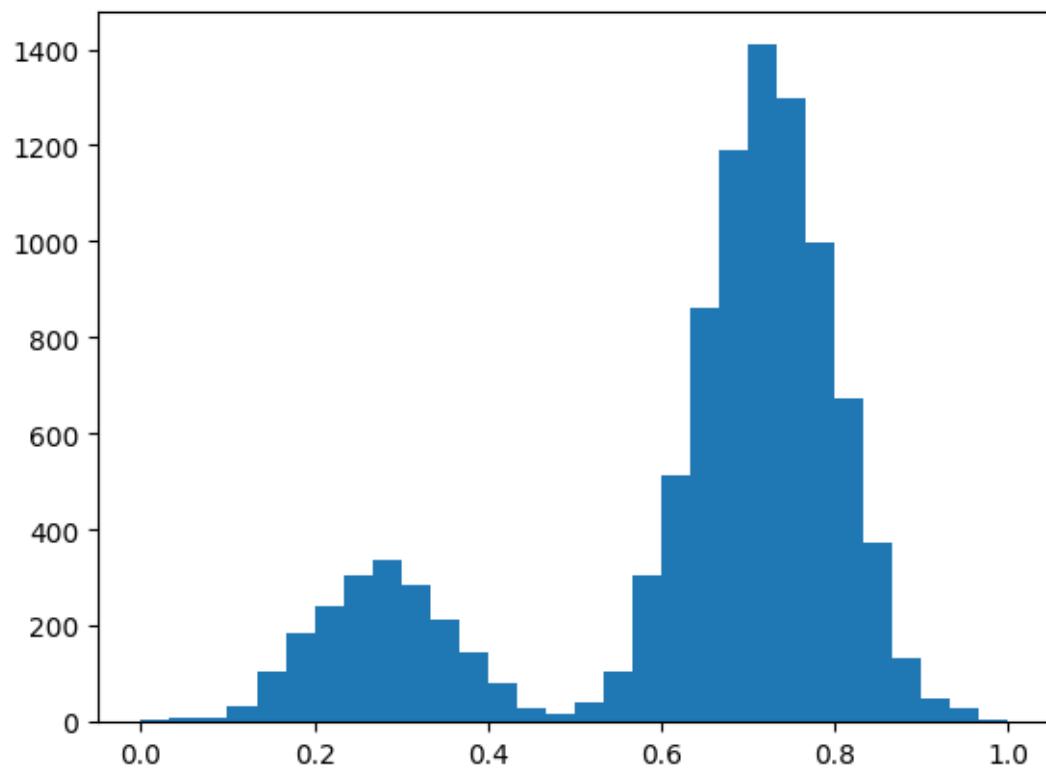
Distribution for "117"



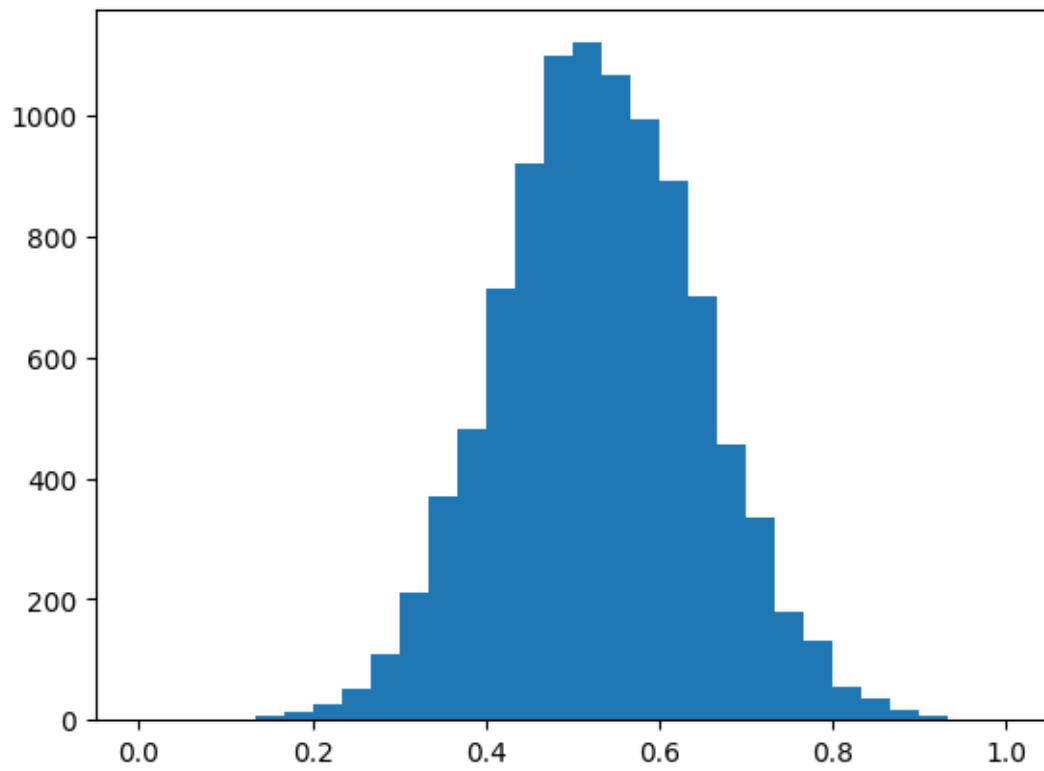
Distribution for "1"



Distribution for "113"

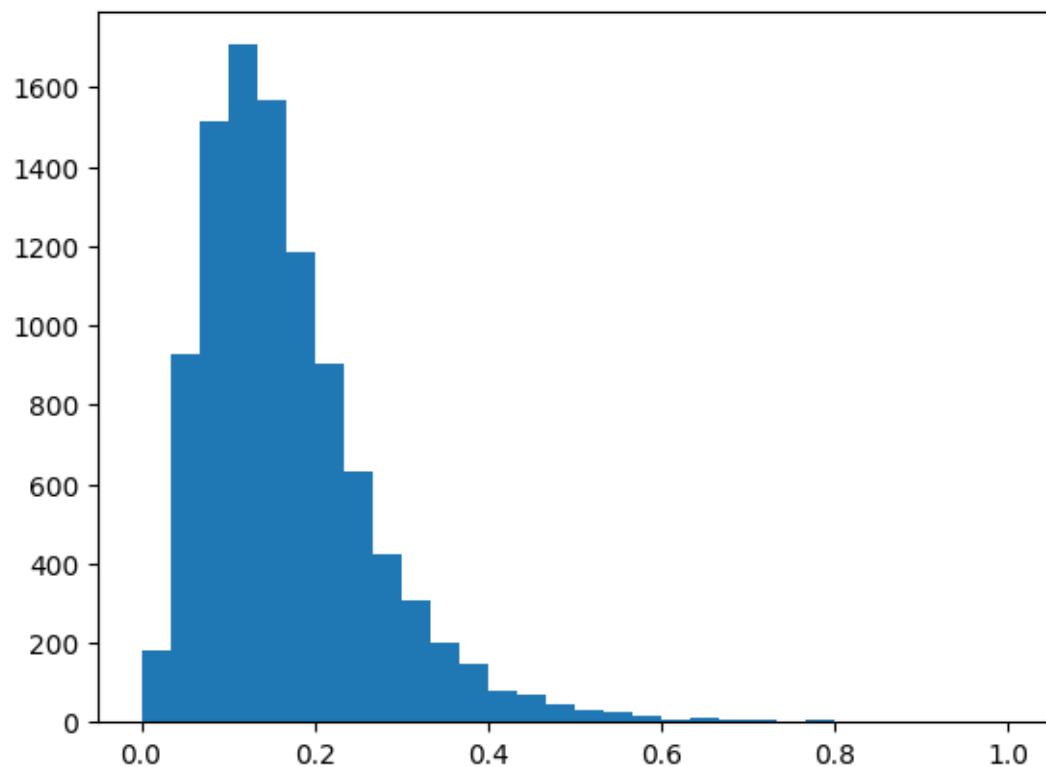


Distribution for "16"

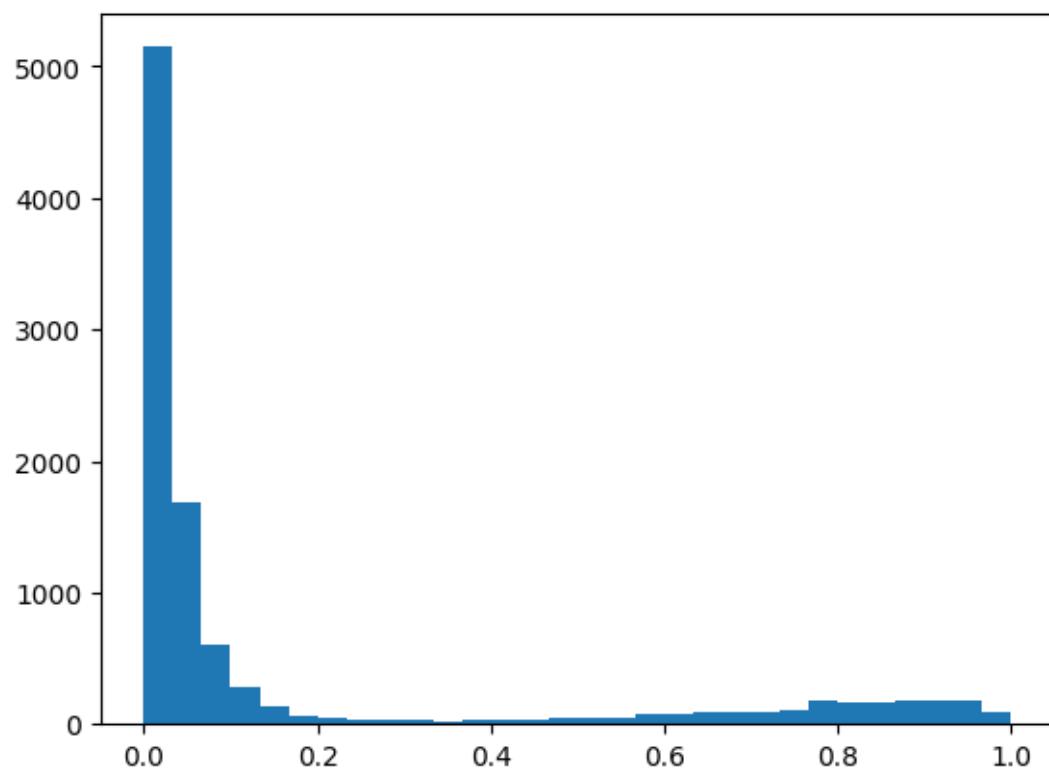


Histogramas de 5 variables al azar de los datos de metilación:

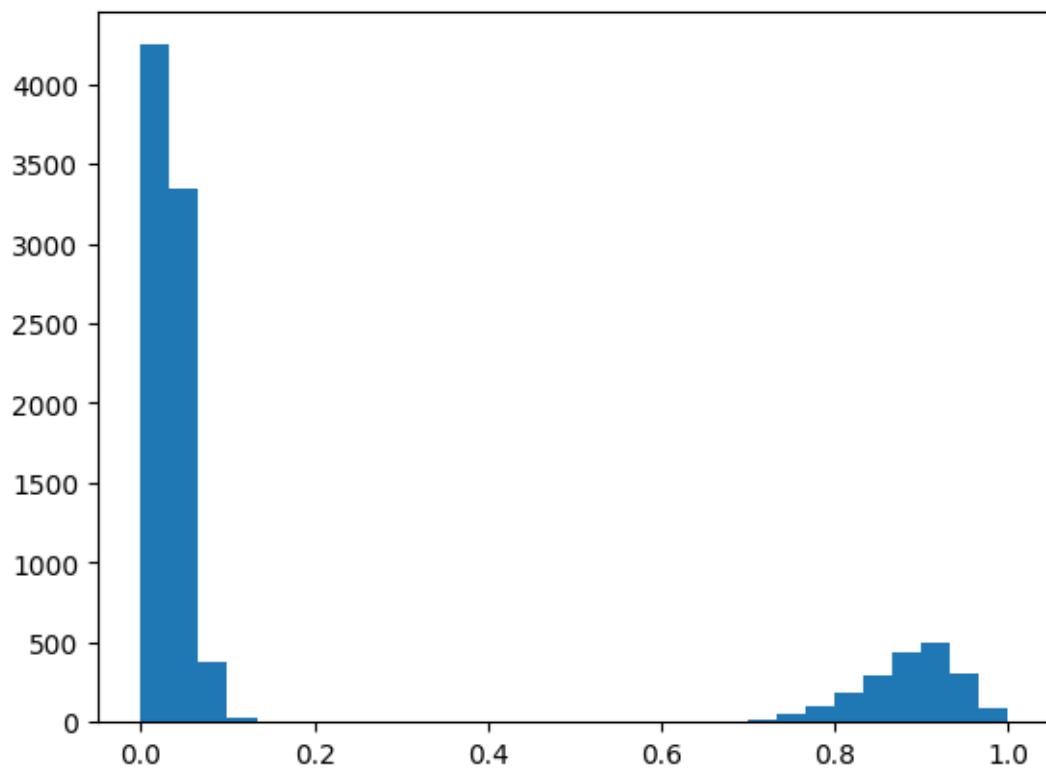
Distribution for "33"



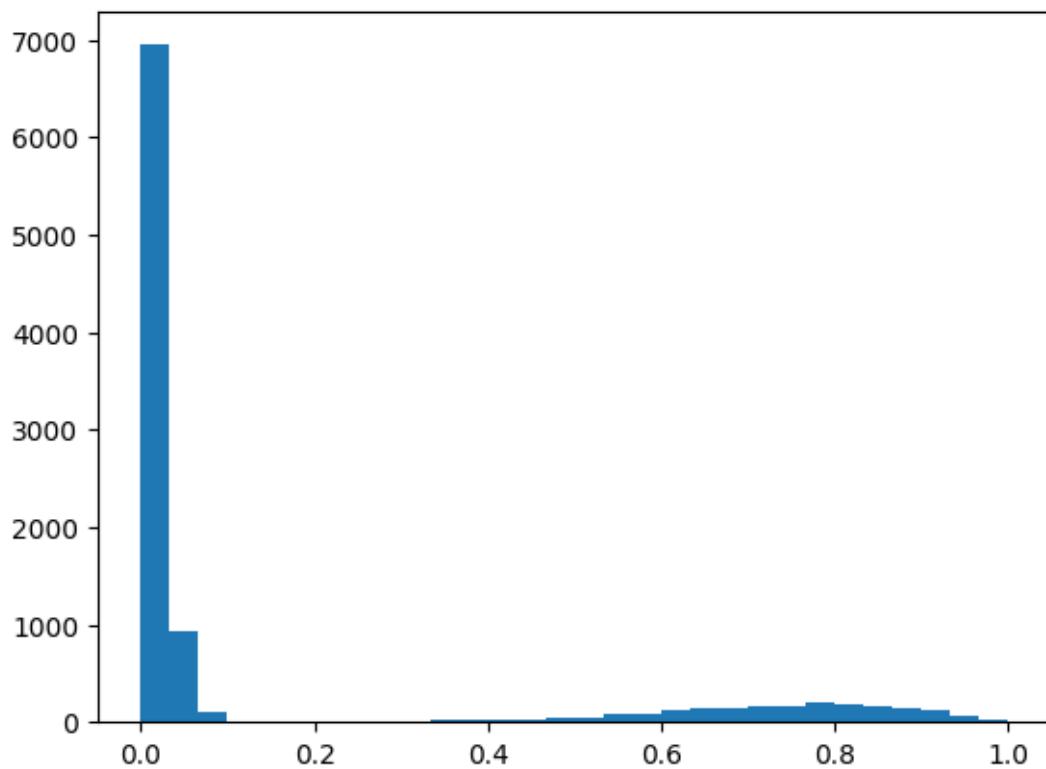
Distribution for "335"



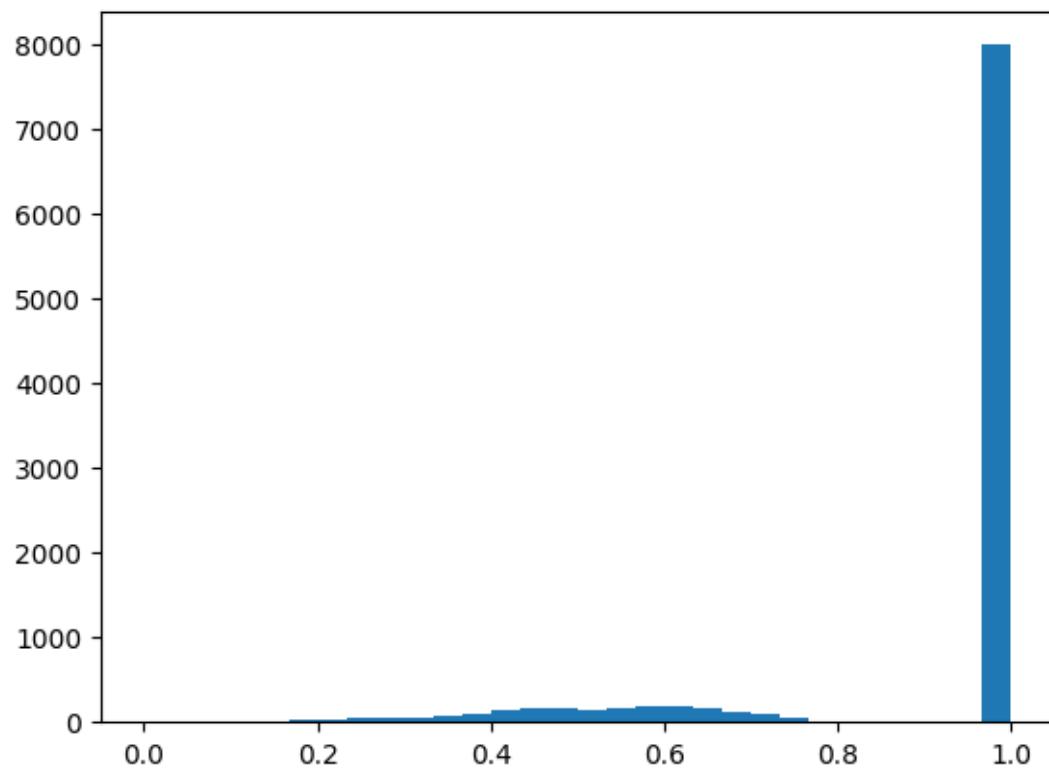
Distribution for "0"



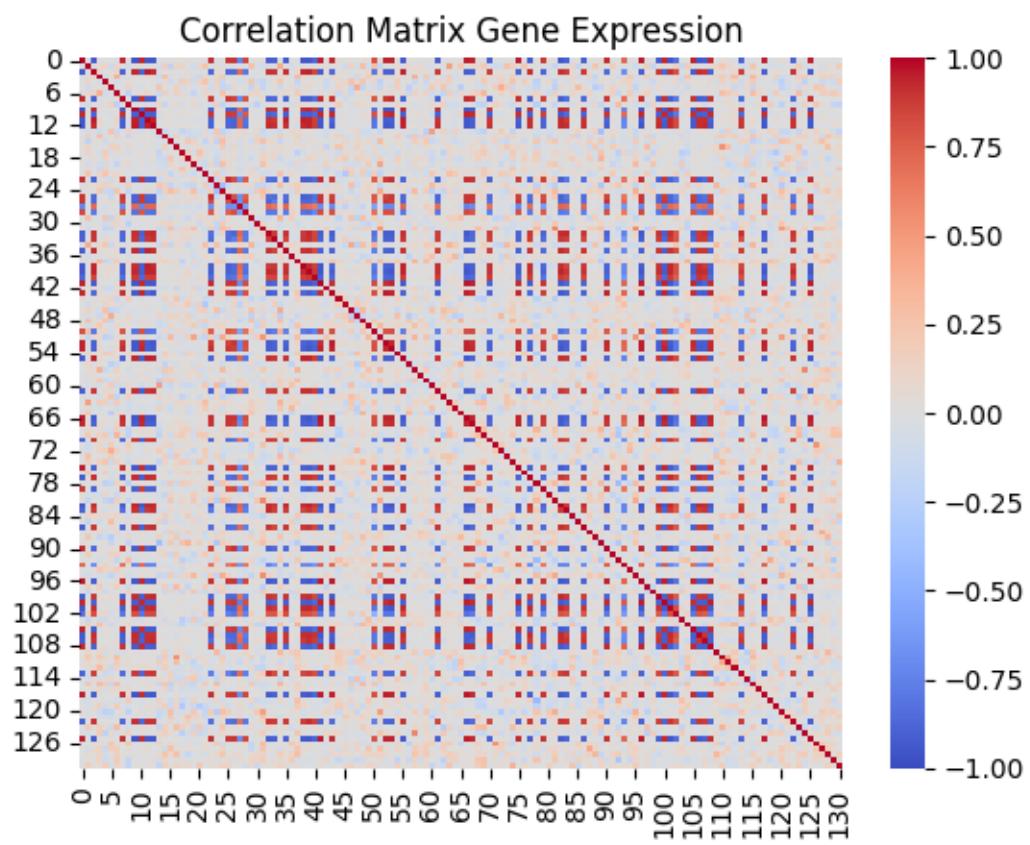
Distribution for "65"

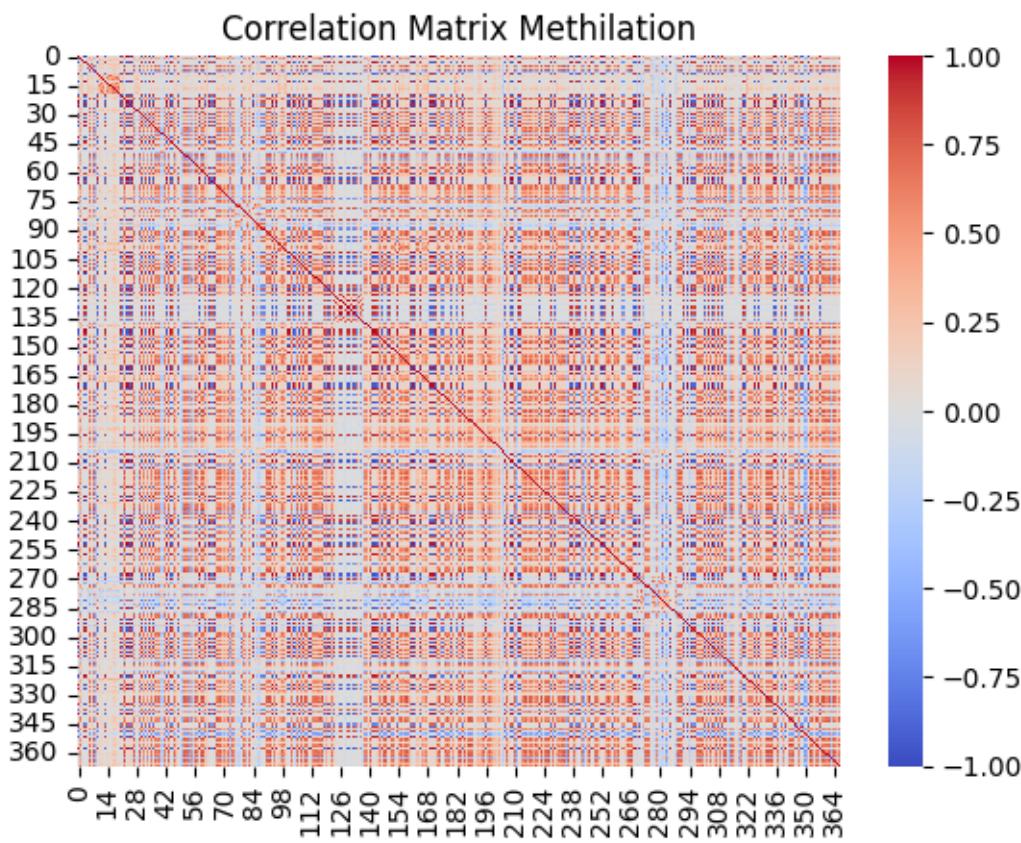


Distribution for "240"



Matrices de correlación:





Matriz de Covarianza:

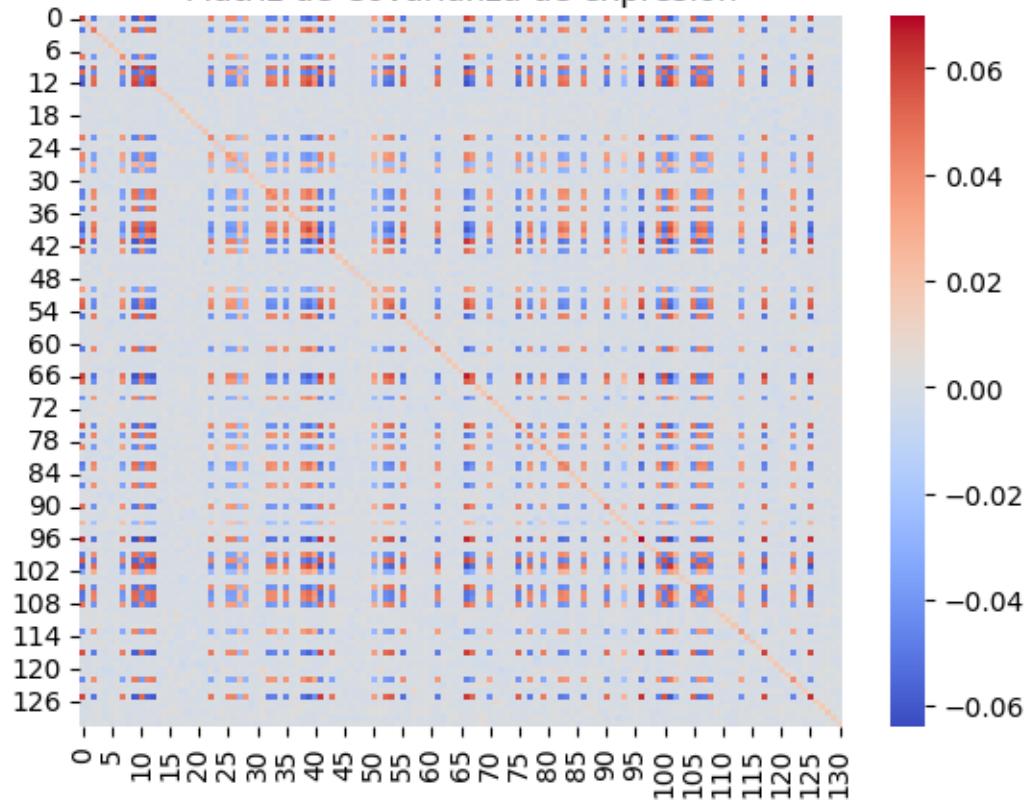
0	1	2	3	4	5	6	\
0	0.061088	-0.000020	-0.050650	-0.000658	-0.000199	-0.000251	-0.000640
1	-0.000020	0.018488	-0.000694	0.001251	0.002205	-0.002198	0.001433
2	-0.050650	-0.000694	0.047822	0.003394	-0.000030	0.000385	0.003421
3	-0.000658	0.001251	0.003394	0.018040	-0.001045	0.000564	0.006205
4	-0.000199	0.002205	-0.000030	-0.001045	0.016827	0.000500	-0.001365
...
362	-0.000053	0.000053	0.000099	0.000101	0.000049	-0.000178	-0.000055
363	-0.000021	0.000163	-0.000021	-0.000157	0.000058	0.000235	0.000195
364	-0.000035	0.000234	-0.000004	-0.000129	0.000118	0.000123	0.000214
365	-0.000275	0.000218	0.000195	-0.000057	-0.000087	-0.000041	-0.000071
366	0.000414	0.000031	-0.000414	-0.000060	-0.000136	0.000033	0.000106
7	8	9	...	357	358	359	\
0	0.046322	0.000222	-0.058695	...	0.083911	0.000354	0.000194
1	0.000527	-0.001672	0.000393	...	0.000988	0.000208	0.000197
2	-0.039492	-0.000544	0.050648	...	-0.072216	-0.000277	-0.000164
3	0.000335	-0.000315	0.000806	...	0.000473	-0.000192	-0.000112
4	-0.000646	0.000865	0.000191	...	-0.000363	0.000109	0.000134
...
362	-0.000060	-0.000126	0.000065	...	-0.001429	-0.005342	-0.004247
363	0.000080	-0.000406	0.000044	...	0.001962	0.007840	0.007568

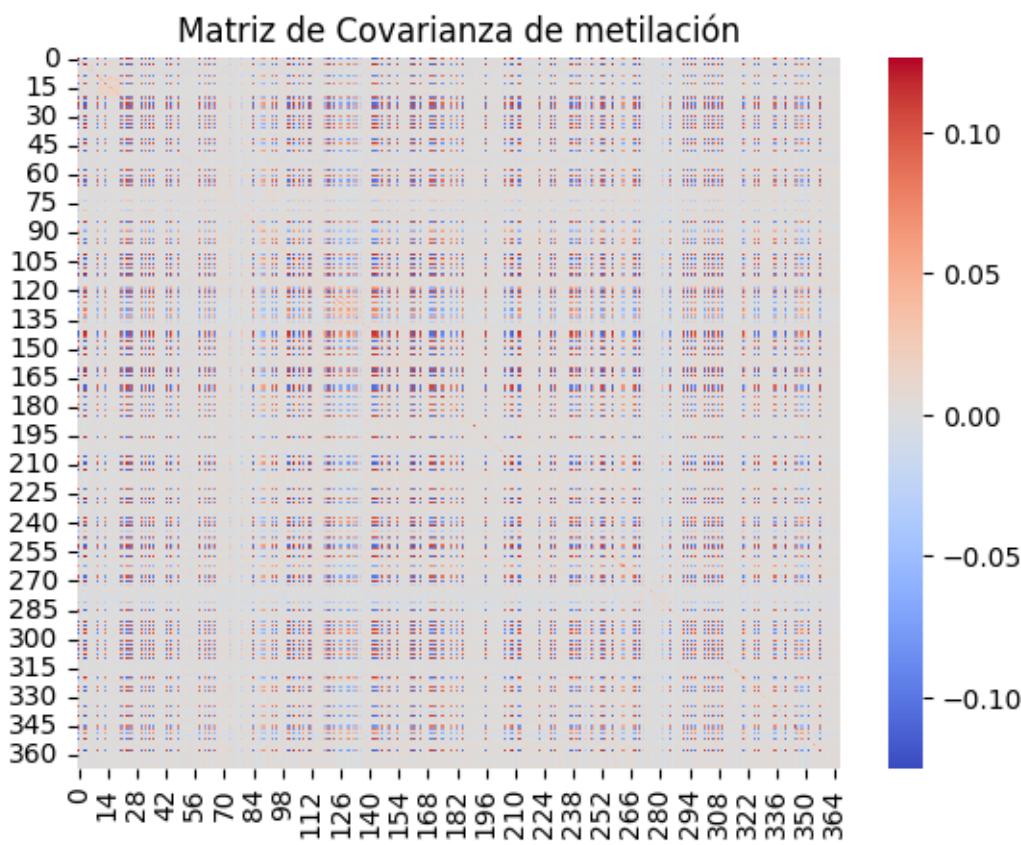
	364	0.000015	-0.000414	0.000135	...	0.001821	0.007054	0.007972
	365	-0.000060	0.000037	0.000240	...	0.000609	0.003105	0.004211
	366	0.000444	-0.000155	-0.000396	...	0.002356	0.006196	0.006196
	360	361	362	363	364	365	366	
↪000414	0	0.000372	1.036053e-04	-0.000053	-0.000021	-0.000035	-0.000275	0.
↪000031	1	0.000056	7.531148e-05	0.000053	0.000163	0.000234	0.000218	0.
↪000414	2	-0.000302	-8.671326e-05	0.000099	-0.000021	-0.000004	0.000195	-0.
↪000060	3	-0.000086	-6.477208e-07	0.000101	-0.000157	-0.000129	-0.000057	-0.
↪000136	4	0.000058	3.468909e-05	0.000049	0.000058	0.000118	-0.000087	-0.
↪
↪004691	362	-0.001974	-4.020002e-03	0.012916	-0.004370	-0.003396	-0.001905	-0.
↪007067	363	0.004677	8.296082e-03	-0.004370	0.010662	0.008524	0.004325	0.
↪006698	364	0.004243	7.509824e-03	-0.003396	0.008524	0.012661	0.005112	0.
↪005127	365	0.003124	4.287567e-03	-0.001905	0.004325	0.005112	0.014974	0.
↪010897	366	0.004640	7.066556e-03	-0.004691	0.007067	0.006698	0.005127	0.

[498 rows x 498 columns]

Matrices de covarianza:

Matriz de Covarianza de expresión





5.2.2. Análisis estadístico de los datos reales

Deabajo podemos ver un pequeño análisis estadístico de los datos una vez procesados. SE ha limitado el numero de variables a 1000 de expresión y 10000 de metilación en el análisis.

Descripción de los datos de expresión génica:						
gene	ERCC-00098	ERCC-00095	ERCC-00077	ERCC-00126	TRIM44	□
↳ C15orf39 \ 000000	count	37.000000	37.000000	37.000000	37.000000	37.000000
↳ 375251	mean	0.274743	0.400184	0.620007	0.444825	0.683416
↳ 205616	std	0.215829	0.264651	0.188099	0.232822	0.198297
↳ 000000	min	0.000000	0.000000	0.000000	0.000000	0.000000
↳ 274592	25 %	0.115098	0.161156	0.552205	0.272242	0.559708
↳ 366070	50 %	0.236963	0.346549	0.629690	0.390928	0.729935
↳ 497167	75 %	0.372541	0.635638	0.745447	0.599998	0.808248

↪000000	max	1.000000	1.000000	1.000000	1.000000	1.000000	1.
	gene	PCDHGA9	STAMBPL1	MGC3771	LOC100130872	...	SNORA33 \
	count	37.000000	37.000000	37.000000	37.000000	...	37.000000
	mean	0.319649	0.636231	0.387641	0.356318	...	0.395264
	std	0.188954	0.248544	0.264985	0.232540	...	0.259025
	min	0.000000	0.000000	0.000000	0.000000	...	0.000000
	25 %	0.201780	0.560228	0.194489	0.147400	...	0.214355
	50 %	0.296558	0.655960	0.313399	0.375285	...	0.361294
	75 %	0.354942	0.790095	0.562637	0.476264	...	0.574657
	max	1.000000	1.000000	1.000000	1.000000	...	1.000000
↪\	gene	PGAP2	FPGS	CRYBG1	UBE2E3	LOC729375	RPP25L ↴
	count	37.000000	37.000000	37.000000	37.000000	37.000000	37.000000
	mean	0.618784	0.408033	0.518566	0.532638	0.532345	0.358699
	std	0.239203	0.271886	0.221570	0.152072	0.231881	0.175548
	min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
	25 %	0.450498	0.207356	0.365901	0.504134	0.396052	0.304085
	50 %	0.668358	0.398389	0.521493	0.531871	0.549964	0.341697
	75 %	0.785022	0.528976	0.638907	0.610239	0.684316	0.422655
	max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
	gene	TBC1D2	B4GALT3	USP4			
	count	37.000000	37.000000	37.000000			
	mean	0.483316	0.425319	0.511400			
	std	0.238607	0.193003	0.266693			
	min	0.000000	0.000000	0.000000			
	25 %	0.335158	0.313608	0.383635			
	50 %	0.448051	0.382794	0.539801			
	75 %	0.674808	0.533062	0.693042			
	max	1.000000	1.000000	1.000000			

[8 rows x 1000 columns]

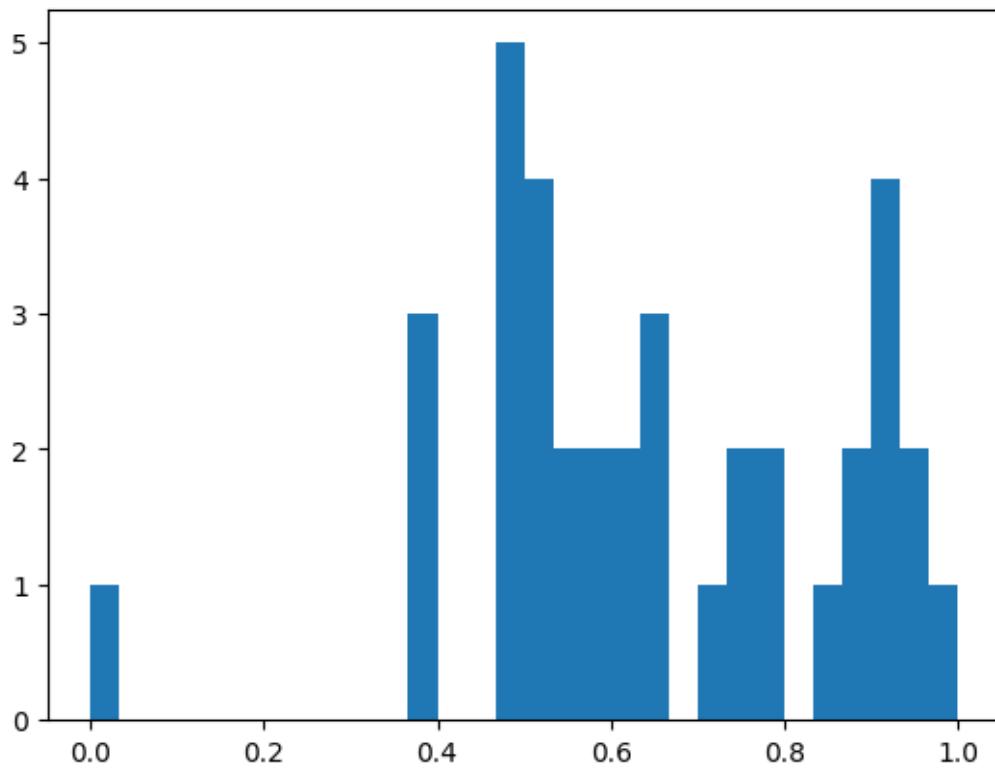
↪cg23803172 ↪\		Descripción de los datos de metilación:				
↪000000	gene	cg24669183	cg15560884	cg01014490	cg17505339	cg11954957 ↴
↪447394	count	37.000000	37.000000	37.000000	37.000000	37.000000
↪231662	mean	0.602112	0.595946	0.190121	0.634959	0.604886
↪000000	std	0.199764	0.177274	0.190543	0.182223	0.166078
↪289179	min	0.000000	0.000000	0.000000	0.000000	0.000000
↪425483	25 %	0.506011	0.503104	0.090716	0.528683	0.524939
↪619019	50 %	0.606643	0.594113	0.121135	0.669476	0.620510
	75 %	0.748317	0.697913	0.219947	0.750441	0.700973

↪000000	max	1.000000	1.000000	1.000000	1.000000	1.000000	1.
	gene	cg16736630	cg05898754	cg03128332	cg16619049	...	cg15470672 ↴
	count	37.000000	37.000000	37.000000	37.000000	...	37.000000
	mean	0.498525	0.573329	0.430325	0.222907	...	0.396659
	std	0.244107	0.240246	0.221699	0.206580	...	0.172427
	min	0.000000	0.000000	0.000000	0.000000	...	0.000000
	25 %	0.335640	0.475534	0.269732	0.129106	...	0.301154
	50 %	0.488001	0.600915	0.446361	0.177869	...	0.374189
	75 %	0.671278	0.727012	0.573409	0.238589	...	0.460477
	max	1.000000	1.000000	1.000000	1.000000	...	1.000000
↪cg11422402 ↴	gene	cg24514954	cg16730266	cg13468252	cg22131571	cg22359362 ↴	
↪000000 ↴	count	37.000000	37.000000	37.000000	37.000000	37.000000	37.
↪378308	mean	0.438861	0.362256	0.491778	0.419700	0.493915	0.
↪222762	std	0.185720	0.182223	0.270571	0.218398	0.240429	0.
↪000000	min	0.000000	0.000000	0.000000	0.000000	0.000000	0.
↪237576	25 %	0.349664	0.264843	0.357602	0.302884	0.332240	0.
↪342899	50 %	0.419630	0.343457	0.465247	0.433195	0.498194	0.
↪541650	75 %	0.539671	0.446691	0.717074	0.507782	0.642675	0.
↪000000	max	1.000000	1.000000	1.000000	1.000000	1.000000	1.
	gene	cg09477198	cg24044651	cg04667458			
	count	37.000000	37.000000	37.000000			
	mean	0.561151	0.422761	0.506047			
	std	0.199356	0.229764	0.283329			
	min	0.000000	0.000000	0.000000			
	25 %	0.439835	0.309963	0.264550			
	50 %	0.535556	0.406425	0.563642			
	75 %	0.738852	0.585994	0.719747			
	max	1.000000	1.000000	1.000000			

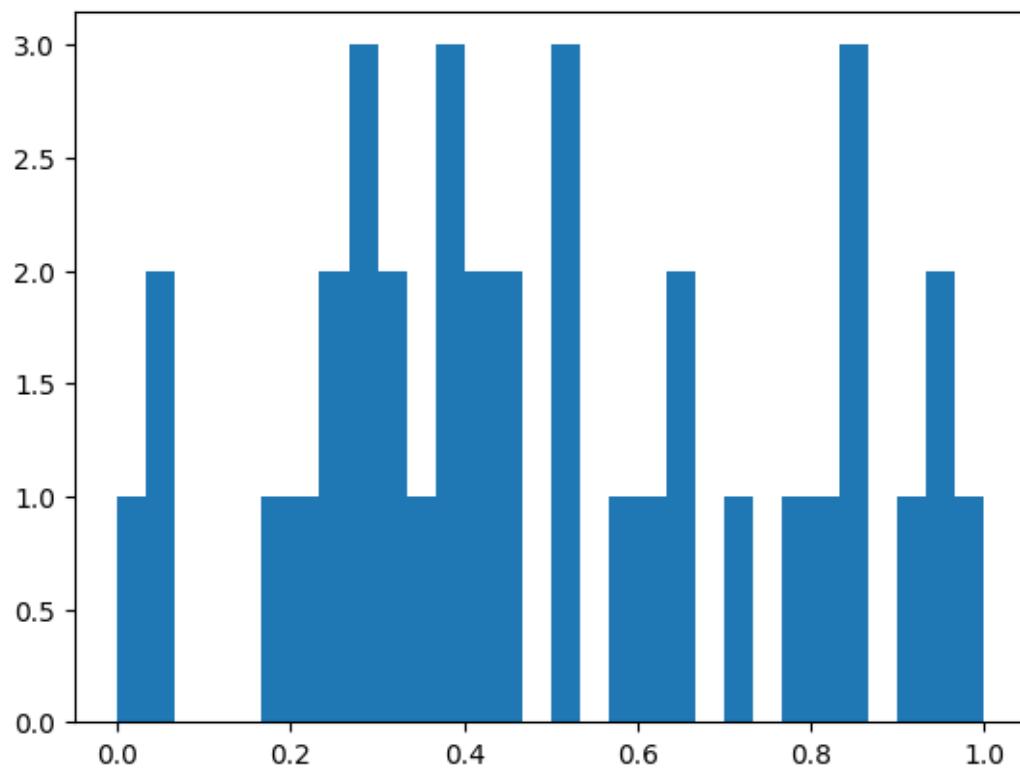
[8 rows x 10000 columns]

Deabajo podemos ver el histograma de 5 variables de expresión elegidas al azar.

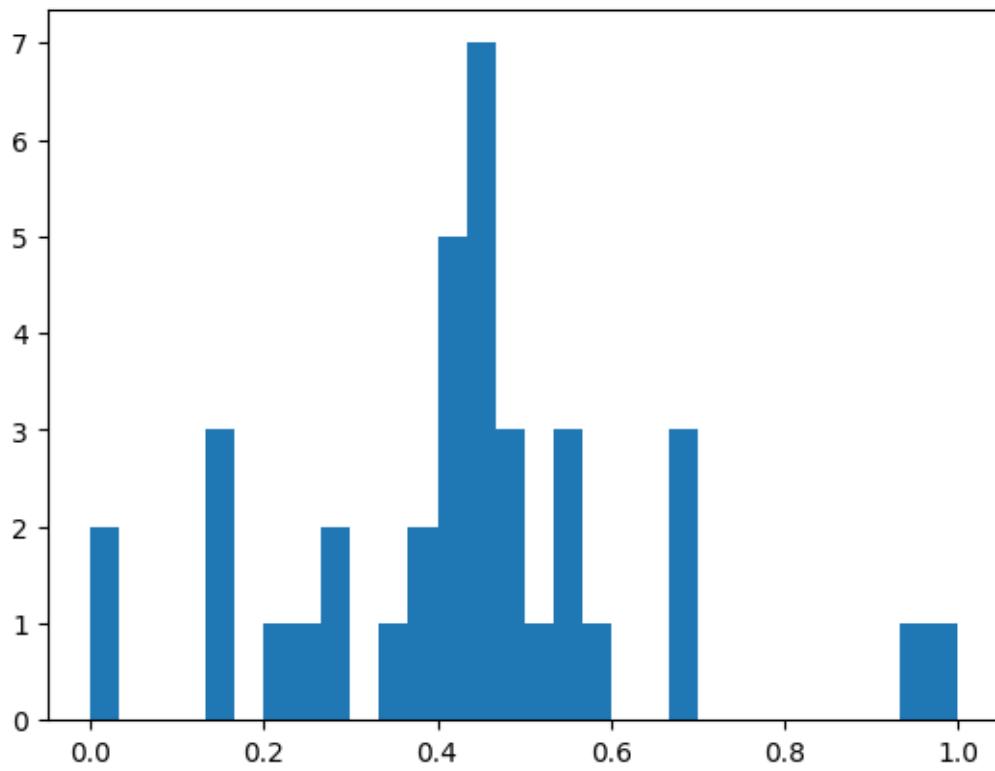
Distribution for "KRCC1"



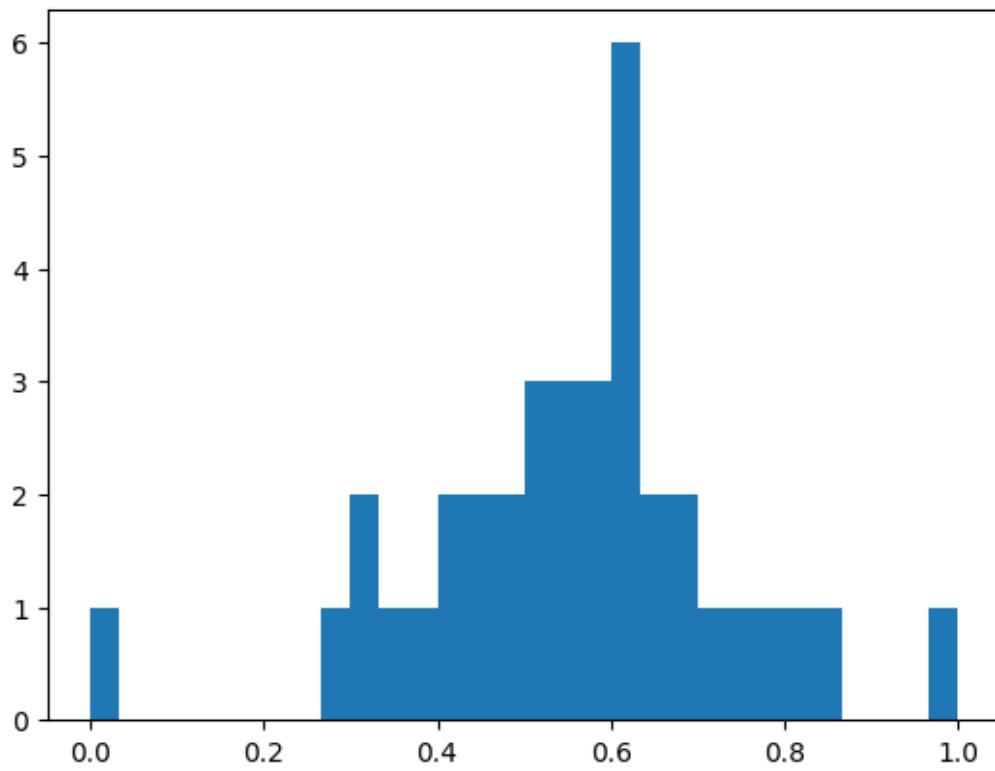
Distribution for "LIPE"



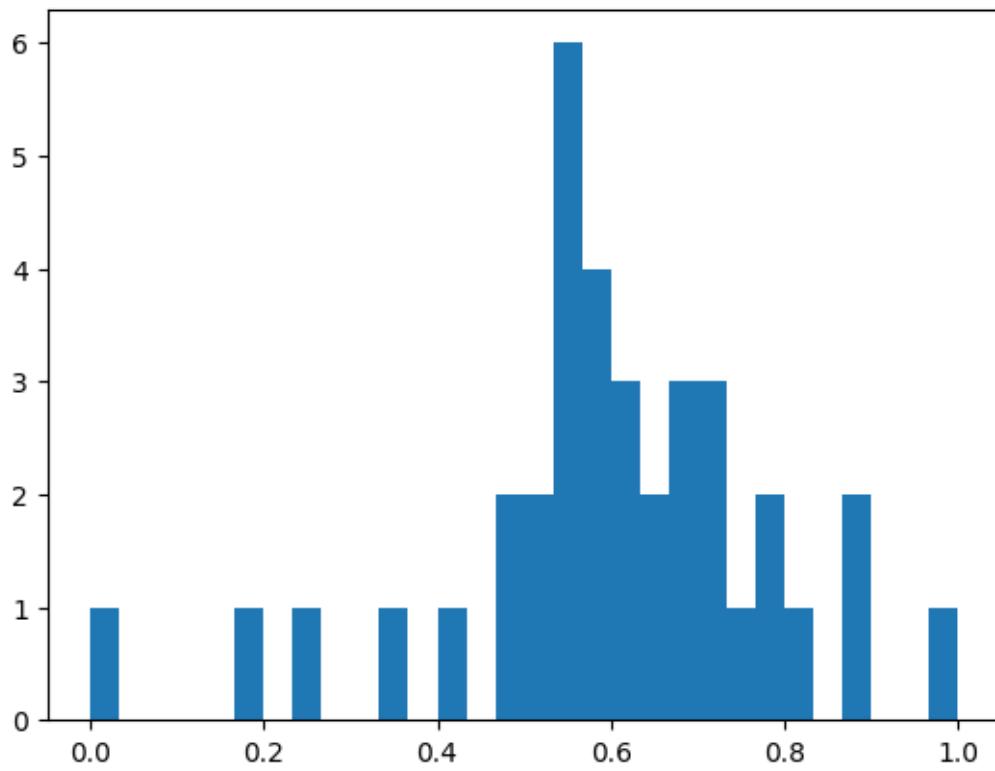
Distribution for "RPS13"



Distribution for "CYTH3"

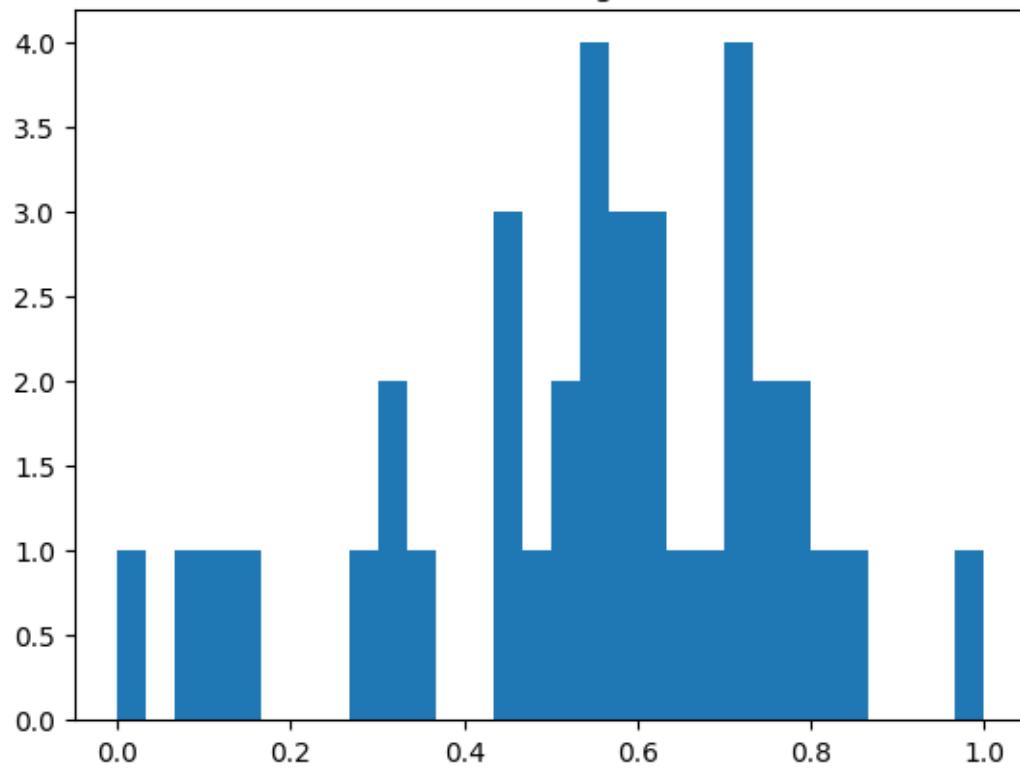


Distribution for "VPS8"

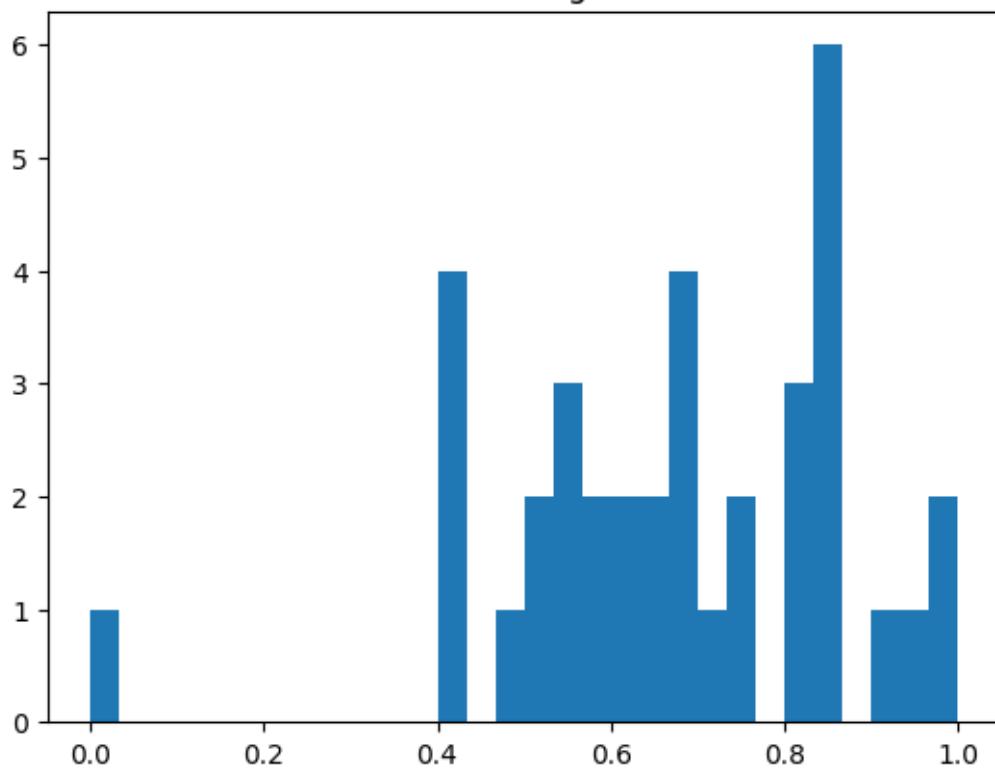


Debajo podemos ver el histograma de 5 variables de metilación elegidas al azar.

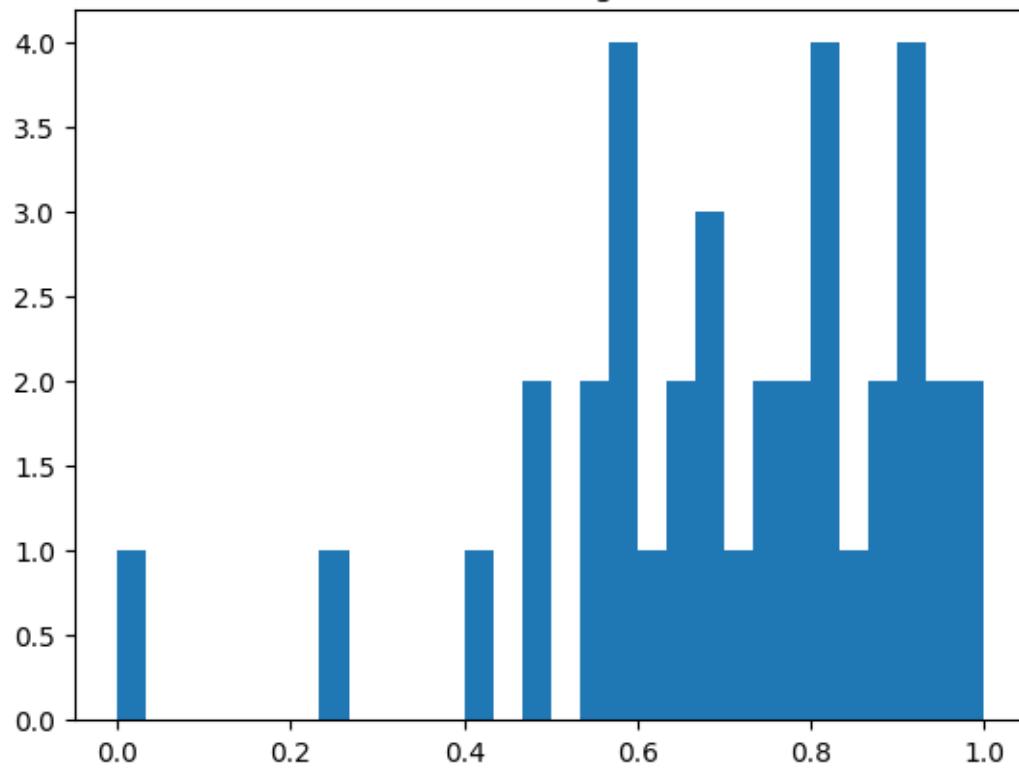
Distribution for "cg12270813"



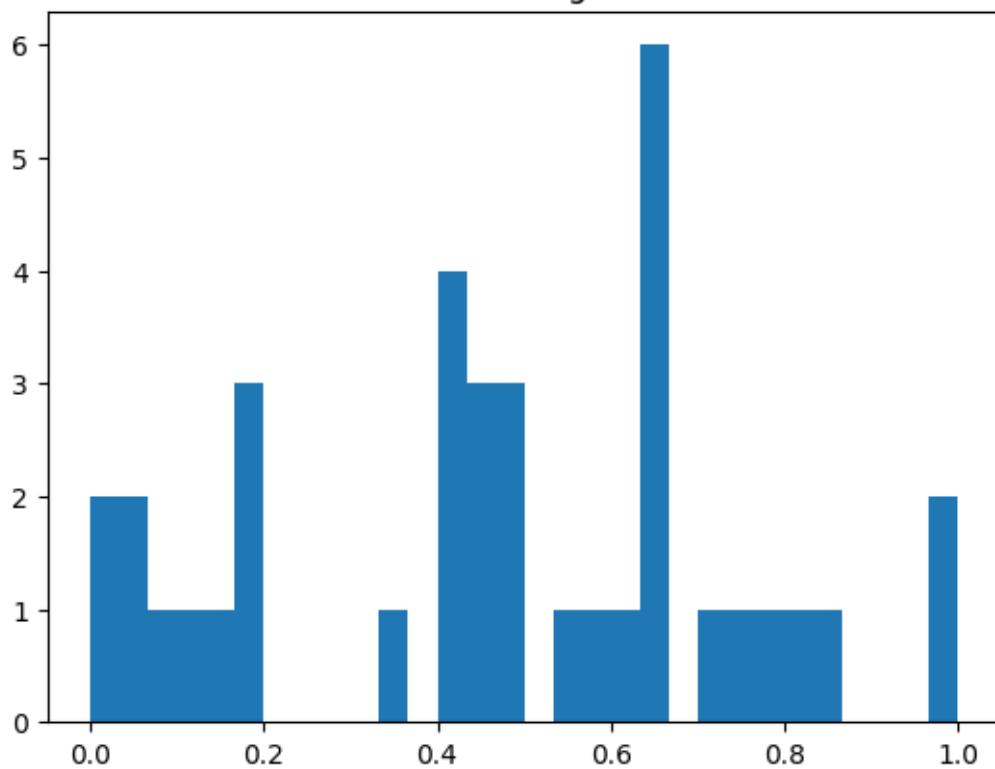
Distribution for "cg08865559"



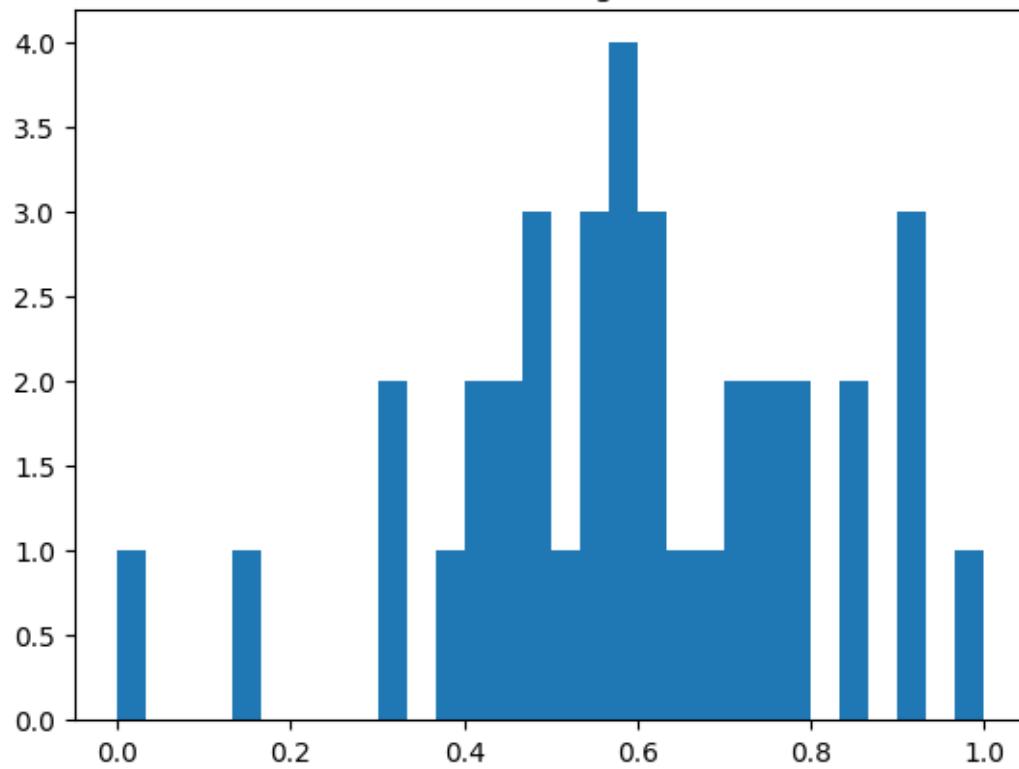
Distribution for "cg11186344"



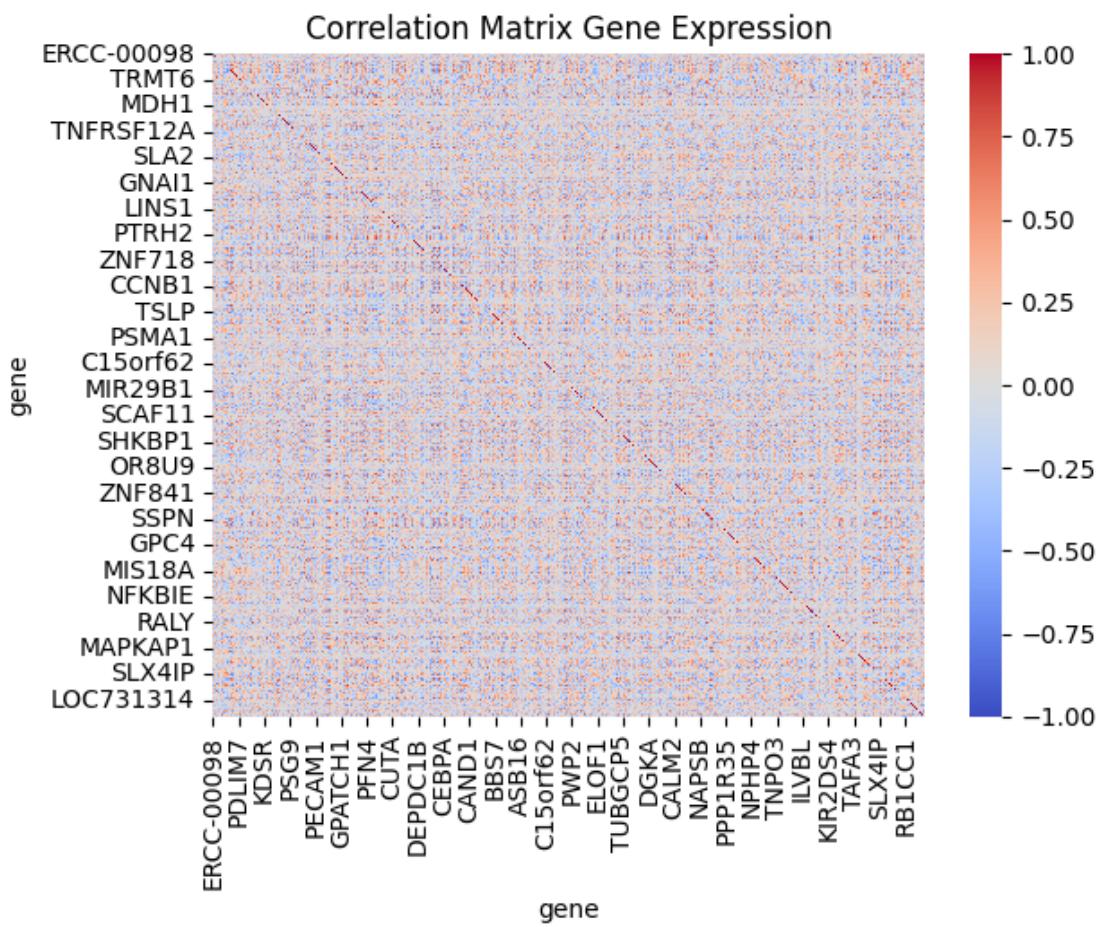
Distribution for "cg00618291"

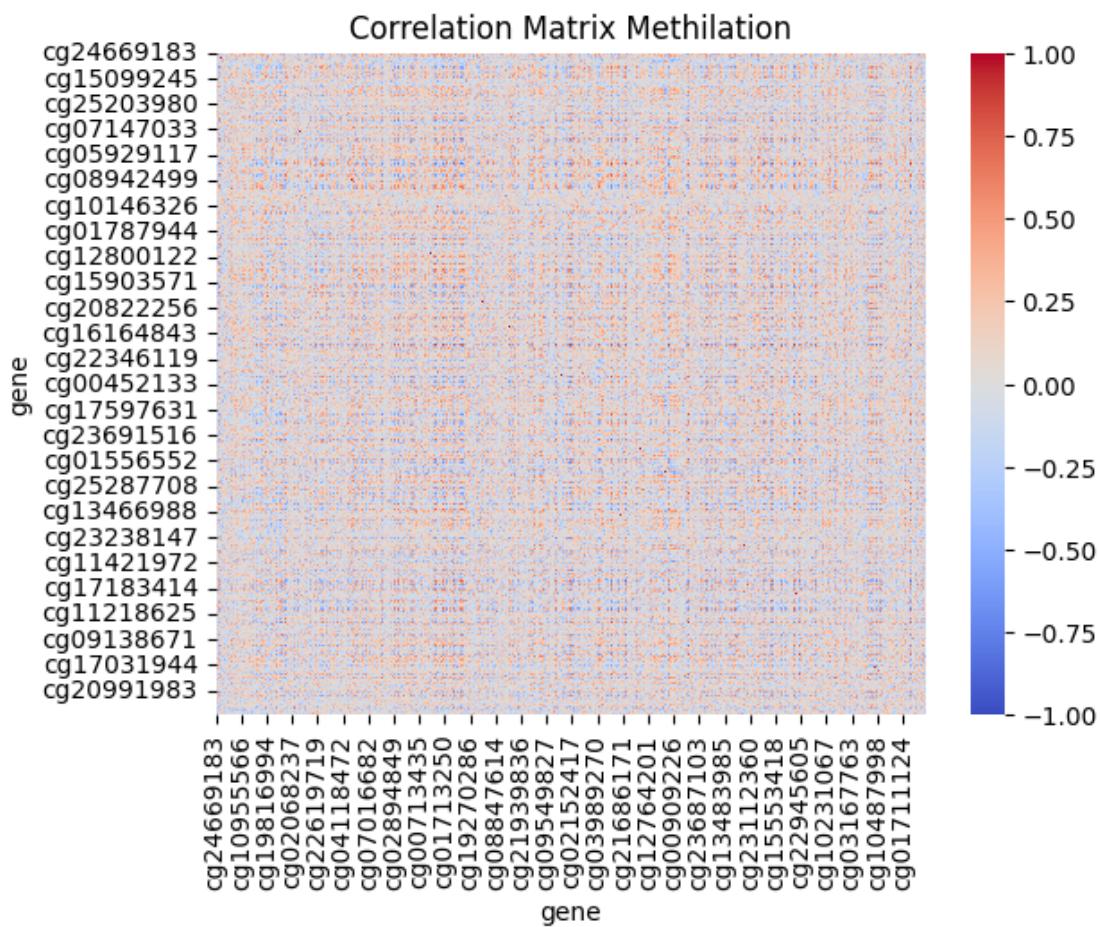


Distribution for "cg06447122"



Debajo podemos ver las matrices de correlación para la expresión y la metilación





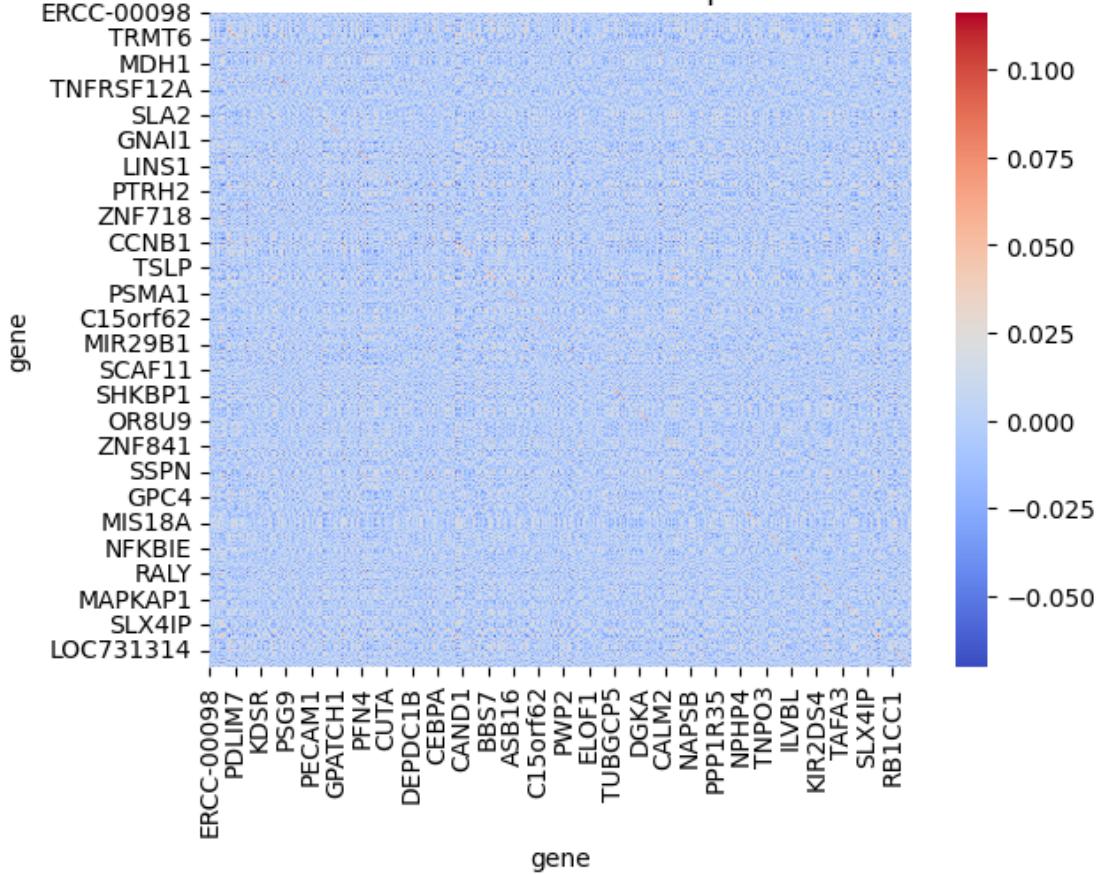
Matriz de Covarianza:

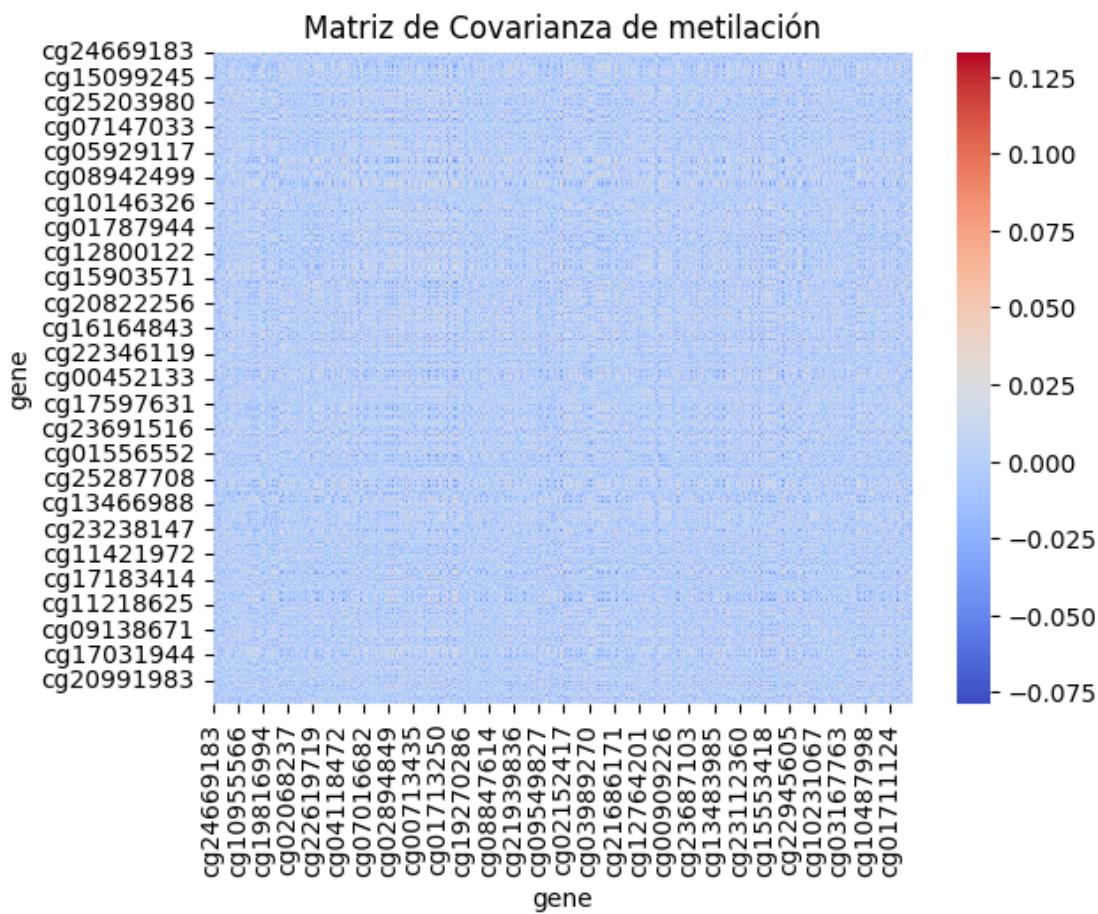
gene	ERCC-00098	ERCC-00095	ERCC-00077	ERCC-00126	TRIM44	\
gene						
ERCC-00098	0.046582	-0.003502	0.007791	-0.018488	-0.002647	
ERCC-00095	-0.003502	0.070040	0.006842	-0.005661	0.005888	
ERCC-00077	0.007791	0.006842	0.035381	0.009619	0.005514	
ERCC-00126	-0.018488	-0.005661	0.009619	0.054206	-0.004341	
TRIM44	-0.002647	0.005888	0.005514	-0.004341	0.039322	
...
cg22359362	0.004050	-0.002084	-0.000621	0.001734	-0.000599	
cg11422402	0.004713	-0.010694	0.000601	0.003453	-0.000515	
cg09477198	0.005987	-0.000858	0.002998	0.008389	-0.006401	
cg24044651	-0.016458	-0.000479	-0.006148	0.006828	0.011582	
cg04667458	-0.006533	0.013812	-0.009930	0.004440	0.005887	
gene	C15orf39	PCDHGA9	STAMBPL1	MGC3771	LOC100130872	...
gene						
ERCC-00098	0.004801	-0.010036	0.009816	-0.012979	0.002849	...
ERCC-00095	0.013729	0.012452	0.013114	-0.017029	-0.014757	...
ERCC-00077	0.008618	-0.019015	-0.001448	-0.009471	0.001352	...
ERCC-00126	0.001778	-0.003853	-0.014880	0.002617	0.002391	...

TRIM44	-0.013893	-0.009859	0.012176	-0.017868	0.008040	...
...
cg22359362	0.000902	0.001123	0.007018	-0.009706	0.027986	...
cg11422402	-0.002283	-0.003843	-0.002962	0.003000	-0.015525	...
cg09477198	-0.003292	-0.006566	-0.006792	-0.008891	0.006787	...
cg24044651	-0.020247	-0.002011	0.018052	-0.000607	0.009016	...
cg04667458	-0.002391	0.004149	0.022486	0.011833	0.006191	...
gene	cg15470672	cg24514954	cg16730266	cg13468252	cg22131571	\
gene						
ERCC-00098	-0.001902	0.009739	0.013329	-0.002234	-0.001042	
ERCC-00095	-0.004458	-0.002336	0.005161	0.007170	0.024026	
ERCC-00077	-0.001410	-0.003318	-0.000430	0.010506	-0.007879	
ERCC-00126	-0.005529	-0.003337	-0.009324	0.006874	-0.016110	
TRIM44	0.007647	-0.000247	-0.000073	-0.008334	0.010501	
...
cg22359362	0.011702	0.010259	0.013971	-0.025600	0.009407	
cg11422402	-0.017742	0.015963	-0.002605	0.013068	-0.006546	
cg09477198	0.000555	0.002337	0.002060	-0.012444	0.001699	
cg24044651	0.002532	0.022632	-0.000105	-0.003158	0.017621	
cg04667458	0.007323	0.006844	0.012219	-0.014066	0.021779	
gene	cg22359362	cg11422402	cg09477198	cg24044651	cg04667458	
gene						
ERCC-00098	0.004050	0.004713	0.005987	-0.016458	-0.006533	
ERCC-00095	-0.002084	-0.010694	-0.000858	-0.000479	0.013812	
ERCC-00077	-0.000621	0.000601	0.002998	-0.006148	-0.009930	
ERCC-00126	0.001734	0.003453	0.008389	0.006828	0.004440	
TRIM44	-0.000599	-0.000515	-0.006401	0.011582	0.005887	
...
cg22359362	0.057806	-0.012471	0.006658	0.011056	0.006988	
cg11422402	-0.012471	0.049623	-0.002860	0.007739	0.008193	
cg09477198	0.006658	-0.002860	0.039743	0.003435	-0.012212	
cg24044651	0.011056	0.007739	0.003435	0.052792	0.016134	
cg04667458	0.006988	0.008193	-0.012212	0.016134	0.080276	

[11000 rows x 11000 columns]

Matriz de Covarianza de expresión





5.3. Anexo C. Código de preprocesamiento y validación

Todo el código del proyecto se puede encontrar en <https://drive.google.com/drive/folders/1k34uYiedf1ssYfsp77BsJfaI3utRY1si?usp=sharing>

5.3.1. Preprocesamiento datos simulados

```
# Leer datos
expression_data = pd.read_csv(fnexpr)
methylation_data = pd.read_csv(fnmeth)
assign_data = pd.read_csv(fnassig)

expression_data = expression_data.iloc[:, 1:]
methylation_data = methylation_data.iloc[:, 1:]
assign_data = assign_data.iloc[:, 2:]

# Convertir todas las columnas a tipo float
expression_data = expression_data.apply(pd.to_numeric, errors='coerce')
methylation_data = methylation_data.apply(pd.to_numeric, errors='coerce')
assign_data = assign_data.apply(pd.to_numeric, errors='coerce')
```

```

# Lidiar con valores NaN (si los hay). Pone 0(CAMBIAR)
#expression_data.fillna(0, inplace=True)
#methylation_data.fillna(0, inplace=True)
#assign_data.fillna(0, inplace=True)

#Normalizamos
scaler = MinMaxScaler()

expression_data_scaled = scaler.fit_transform(expression_data)
methylation_data_scaled = scaler.fit_transform(methylation_data)
assign_data_scaled = assign_data

# Convertir a DataFrame
expression_data = pd.DataFrame(expression_data_scaled, index=expression_data.index, columns=expression_data.columns)
methylation_data = pd.DataFrame(methylation_data_scaled, index=methylation_data.index, columns=methylation_data.columns)
assign_data = pd.DataFrame(assign_data_scaled, index=assign_data.index, columns=assign_data.columns)

#print(torch.cuda.is_available())
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device="cpu"

# Convertir a Tensor y pasar a GPU
expression_data = torch.FloatTensor(expression_data.values).to(device)
methylation_data = torch.FloatTensor(methylation_data.values).to(device)
assign_data = torch.FloatTensor(assign_data.values).to(device)
combined_data = torch.cat((expression_data,methylation_data,assign_data), 1).to(device)

```

Preprocesamiento datos reales

```

import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
import time
from sklearn.preprocessing import StandardScaler,MinMaxScaler

fnexpr='GSE117931/GSE117931_GPL14951.tsv'
fnmet='GSE117931/GSE117931_GPL13534.tsv'
fnassig='GSE117931/assig.csv'
nsamp=37
fmmodel='G-WGANGPREAL'
fmmodeld='D-WGANGPREAL'
filename = "wgangpreal-"

# Leer datos
expression_data = pd.read_csv(fnexpr,sep='\t', index_col=0)
expression_data=expression_data.T
methylation_data = pd.read_csv(fmmodel,sep='\t', index_col=0)
methylation_data=methylation_data.T
assign_data = pd.read_csv(fnassig,sep='\t')

expression_data = expression_data.iloc[:, 1:]
methylation_data = methylation_data.iloc[:, 1:]
assign_data = assign_data.iloc[:, 1:]

# Convertir todas las columnas a tipo float
expression_data = expression_data.apply(pd.to_numeric, errors='coerce')
methylation_data = methylation_data.apply(pd.to_numeric, errors='coerce')
assign_data = assign_data.apply(pd.to_numeric, errors='coerce')

# Lidiar con valores NaN (si los hay).

```

```

methylolation_data.dropna(axis=1, inplace=True)
#expression_data.fillna(0, inplace=True)
#methylolation_data.fillna(0, inplace=True)
#assign_data.fillna(0, inplace=True)

#Normalizamos
scaler = MinMaxScaler()

expression_data_scaled = scaler.fit_transform(expression_data)
methylolation_data_scaled = scaler.fit_transform(methylolation_data)
assign_data_scaled = assign_data

# Convertir a DataFrame
expression_data = pd.DataFrame(expression_data_scaled, index=expression_data.index, columns=expression_data.columns)
methylolation_data = pd.DataFrame(methylolation_data_scaled, index=methylolation_data.index, columns=methylolation_data.columns)
assign_data = pd.DataFrame(assign_data_scaled, index=assign_data.index, columns=assign_data.columns)

#print(torch.cuda.is_available())
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device="cpu"

# Convertir a Tensor y pasar a GPU
expression_data = torch.FloatTensor(expression_data.values).to(device)
methylolation_data = torch.FloatTensor(methylolation_data.values).to(device)
assign_data = torch.FloatTensor(assign_data.values).to(device)
combined_data = torch.cat((expression_data,methylolation_data,assign_data), 1).to(device)

```

5.3.2. Validación modelo GAN

[7]:

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import random

from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from scipy.spatial.distance import cdist
from scipy.stats import wasserstein_distance, ks_2samp
import sys
sys.setrecursionlimit(20000)

# Cargamos los datos
expr_data = pd.read_csv(fnexpr)
meth_data = pd.read_csv(fnmet)
assig_data = pd.read_csv(fnassig)
expr_data = expr_data.iloc[:, 1:]
meth_data = meth_data.iloc[:, 1:]
assig_data = assig_data.iloc[:, 2:]

# Convertir todas las columnas a tipo float
expr_data = expr_data.apply(pd.to_numeric, errors='coerce')
meth_data = meth_data.apply(pd.to_numeric, errors='coerce')
assig_data = assig_data.apply(pd.to_numeric, errors='coerce')

# Lidiar con valores NaN (si los hay). Pone 0 (CAMBIAR)

```

```

#expr_data.fillna(0, inplace=True)
#meth_data.fillna(0, inplace=True)
#assig_data.fillna(0, inplace=True)

#Normalizamos
expr_data = scaler.fit_transform(expr_data)
meth_data = scaler.fit_transform(meth_data)
expr_data = pd.DataFrame(expr_data)
meth_data = pd.DataFrame(meth_data)
assig_data = pd.DataFrame(assig_data)

# Combinamos los datos de expresión génica y metilación
X_real = pd.concat([expr_data, meth_data], axis=1)

# Generar datos de expresión sintética
gen2 = Generator(100, expression_data.shape[1]+methylation_data.shape[1]+assign_data.
→shape[1]).to(device)
gen2.load_state_dict(torch.load(fnmodel))
gen2.eval()
expressions_list = []
for i in range(nsamp):
noise = torch.randn(1, 100).to(device)
synthetic_expression = gen2(noise).to(device)
#print(synthetic_expression)
expressions_list.append(synthetic_expression.detach().cpu().numpy().squeeze())

X_gan = pd.DataFrame(expressions_list)

y = X_gan.iloc[:, -1].values.astype(int) # Suponiendo que la asignación está en la primera
→columna

X = X_gan.iloc[:, :-1]

last_column = X_gan.columns[-1]
X_gan = X_gan.drop(columns=[last_column])
X_gan.columns = X_real.columns
# Concatena los datos reales con los generados
X_combined = np.vstack([X_real, X_gan])

# Aplanar los datos para cálculos estadísticos
real_data_flattened = X_real.values.flatten()
generated_data_flattened = X_gan.values.flatten()

# Wasserstein Distance
w_distance = wasserstein_distance(real_data_flattened, generated_data_flattened)

# KS Test y Wassertein Distance
ks_statistic, ks_pvalue = ks_2samp(real_data_flattened, generated_data_flattened)

print(f"Wasserstein Distance: {w_distance}")
print(f"KS Statistic: {ks_statistic}, P-Value: {ks_pvalue}")

# Distancia Euclíadiana
distancias_euclidianas = cdist(X_real, X_gan, metric='euclidean')
distancia_promedio_euclidiana = np.mean(distancias_euclidianas)
print("Distancia Euclíadiana Promedio:", distancia_promedio_euclidiana)

# Calcula las matrices de correlación
corr_real = X_real.corr()
corr_gan = X_gan.corr()

# Dibuja las matrices de correlación
plt.figure(figsize=(12, 6))

```

```

plt.subplot(1, 2, 1)
sns.heatmap(corr_real, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Matrix (Real Data)')

plt.subplot(1, 2, 2)
sns.heatmap(corr_gan, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Matrix (Generated Data)')

plt.show()

# Etiquetas: 1 para real, 0 para generado
y_real = [1] * X_real.shape[0]
y_gan = [0] * X_gan.shape[0]

# Combina los datos y las etiquetas
y_combined = y_real + y_gan

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_combined, y_combined, test_size=0.3,random_state=42)

# Inicializa y entrena el clasificador SVC
svc = SVC()
svc.fit(X_train, y_train)

# Realiza predicciones en el conjunto de prueba
y_pred = svc.predict(X_test)
# Evalúa el rendimiento del clasificador
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.savefig(filename+"SVC1.jpg")
plt.show()

# Dividimos los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Entrenamos el modelo SVC
clf = SVC()
unique_classes = np.unique(y_train)
if len(unique_classes) > 1:
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    print(classification_report(y_test, y_pred))

    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt='g')
    plt.xlabel('Predicted')
    plt.ylabel('Truth')
    plt.savefig(filename+"SVC2.jpg")
    plt.show()
else:
    print("SVC: Solo una clase")

# Clustering jerárquico
linked1 = linkage(X_real, method='ward')
linked2 = linkage(X_gan, method='ward')

```

```

plt.figure(figsize=(10, 7))
dendrogram(linked1, orientation='top', no_labels=True)
plt.title("Dendrograma jerárquico Real")
plt.savefig(filename+"DJR.jpg")
plt.show()

plt.figure(figsize=(10, 7))
dendrogram(linked2, orientation='top', no_labels=True)
plt.title("Dendrograma jerárquico Generado")
plt.savefig(filename+"DJG.jpg")
plt.show()

# K-Means
k = 2
kmeans = KMeans(n_clusters=k)
kmeans.fit(X_real)
labels = kmeans.predict(X_real)
centroids = kmeans.cluster_centers_
X_array = X_real.values
generated_data=X_gan.values
plt.scatter(X_array[:, 0], X_array[:, 1], c=labels, s=50, cmap='viridis')
plt.scatter(generated_data[:, 0], generated_data[:, 1], s=50, color='pink', label='Datos ↵Generados')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=200, alpha=0.75)
plt.savefig(filename+"KNN.jpg")
plt.show()

# Aplicar t-SNE
# Crea etiquetas para los datos (1 para reales, 0 para GAN)
labels = np.concatenate([np.ones(X_real.shape[0]), np.zeros(X_gan.shape[0])])
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_combined)

plt.scatter(X_tsne[labels==1, 0], X_tsne[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_tsne[labels==0, 0], X_tsne[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('t-SNE visualization')
plt.savefig(filename+"tsne.jpg")
plt.show()

#PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_combined)

plt.scatter(X_pca[labels==1, 0], X_pca[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_pca[labels==0, 0], X_pca[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('PCA visualization')
plt.savefig(filename+"pca.jpg")
plt.show()

#KDE
column_names = X_real.columns.tolist()
random_columns = random.sample(column_names, 20)
for column_name in random_columns:
    sns.kdeplot(X_real[column_name], label='Real', color='blue')
    sns.kdeplot(X_gan[column_name], label='Generated', color='red')
plt.legend()
plt.title(f'Distribution for "{column_name}"')
plt.savefig(f"{filename}kde-{column_name}.jpg")
plt.show()

```

5.3.3. Validación modelo VAE

[53]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import random

from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from scipy.spatial.distance import cdist
from scipy.stats import wasserstein_distance, ks_2samp
import sys
sys.setrecursionlimit(20000)

# Cargamos los datos
expr_data = pd.read_csv(fnexpr)
meth_data = pd.read_csv(fnmeth)
assig_data = pd.read_csv(fnassig)
expr_data = expr_data.iloc[:, 1:]
meth_data = meth_data.iloc[:, 1:]
assig_data = assig_data.iloc[:, 2:]

# Convertir todas las columnas a tipo float
expr_data = expr_data.apply(pd.to_numeric, errors='coerce')
meth_data = meth_data.apply(pd.to_numeric, errors='coerce')
assig_data = assig_data.apply(pd.to_numeric, errors='coerce')

# Lidiar con valores NaN (si los hay). Pone 0 (CAMBIAR)
#expr_data.fillna(0, inplace=True)
#meth_data.fillna(0, inplace=True)
#assig_data.fillna(0, inplace=True)

expr_data = scaler.fit_transform(expr_data)
meth_data = scaler.fit_transform(meth_data)
expr_data = pd.DataFrame(expr_data)
meth_data = pd.DataFrame(meth_data)
assig_data = pd.DataFrame(assig_data)

# Combinamos los datos de expresión génica y metilación
X_real = pd.concat([expr_data, meth_data], axis=1)

# Generar datos de expresión sintética
vae = VAE(combined_data.shape[1], ldim).to(device)
vae.load_state_dict(torch.load(fnmmodel))
vae.eval()

# Función para reparametrización (para generar muestras del espacio latente)

# Generar datos de expresión sintética

with torch.no_grad():
    z = torch.randn(nsamp, ldim).to(device)
    synthetic_expression = vae.decoder(z).to(device)
    synthetic_expression = torch.sigmoid(synthetic_expression)
    #print(synthetic_expression)
    expressions_list=synthetic_expression.detach().cpu().numpy().squeeze()

X_gan = pd.DataFrame(expressions_list)
#X = X.apply(pd.to_numeric, errors='coerce')
y = X_gan.iloc[:, -1].values.astype(int)
```

```

X = X_gan.iloc[:, :-1]

last_column = X_gan.columns[-1]
X_gan = X_gan.drop(columns=[last_column])
X_gan.columns = X_real.columns
# Concatena los datos reales con los generados
X_combined = np.vstack([X_real, X_gan])

# Aplanar los datos para cálculos estadísticos
real_data_flattened = X_real.values.flatten()
generated_data_flattened = X_gan.values.flatten()

# Wasserstein Distance
w_distance = wasserstein_distance(real_data_flattened, generated_data_flattened)

# KS Test
ks_statistic, ks_pvalue = ks_2samp(real_data_flattened, generated_data_flattened)

print(f"Wasserstein Distance: {w_distance}")
print(f"KS Statistic: {ks_statistic}, P-Value: {ks_pvalue}")

# Suponiendo que X_real y X_gan son tus dos grupos de datos
distancias_euclidianas = cdist(X_real, X_gan, metric='euclidean')

# Calcular la distancia promedio
distancia_promedio_euclidiana = np.mean(distancias_euclidianas)
print("Distancia Euclidiana Promedio:", distancia_promedio_euclidiana)

# Calcula las matrices de correlación
corr_real = X_real.corr()
corr_gan = X_gan.corr()

# Dibuja las matrices de correlación
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
sns.heatmap(corr_real, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Matrix (Real Data)')

plt.subplot(1, 2, 2)
sns.heatmap(corr_gan, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('Correlation Matrix (Generated Data)')

plt.show()

# Etiquetas: 1 para real, 0 para generado
y_real = [1] * X_real.shape[0]
y_gan = [0] * X_gan.shape[0]

# Combina los datos y las etiquetas
#X_combined = pd.concat([X_real, X_gan], axis=0)
y_combined = y_real + y_gan

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_combined, y_combined, test_size=0.3,random_state=42)

# Inicializa y entrena el clasificador SVC
svc = SVC()
svc.fit(X_train, y_train)

# Realiza predicciones en el conjunto de prueba
y_pred = svc.predict(X_test)
# Evalúa el rendimiento del clasificador
print("Accuracy:", accuracy_score(y_test, y_pred))

```

```

print("\nClassification Report:\n", classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.savefig(filename+"SVC1.jpg")
plt.show()

# Dividimos los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Entrenamos el modelo SVC
clf = SVC()
unique_classes = np.unique(y_train)
if len(unique_classes) > 1:
    clf.fit(X_train, y_train)
    # Predicciones
    y_pred = clf.predict(X_test)

    # Métricas de clasificación
    print(classification_report(y_test, y_pred))

    # Matriz de confusión
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt='g')
    plt.xlabel('Predicted')
    plt.ylabel('Truth')
    plt.savefig(filename+"SVC2.jpg")
    plt.show()
else:
    print("SVC: Solo una clase")

# Clustering jerárquico
linked1 = linkage(X_real, method='ward')
linked2 = linkage(X_gan, method='ward')

plt.figure(figsize=(10, 7))
dendrogram(linked1, orientation='top', no_labels=True)
plt.title("Dendrograma jerárquico Real")
plt.savefig(filename+"DJR.jpg")
plt.show()

plt.figure(figsize=(10, 7))
dendrogram(linked2, orientation='top', no_labels=True)
plt.title("Dendrograma jerárquico Generado")
plt.savefig(filename+"DJG.jpg")
plt.show()

# K-Means
k = 2
kmeans = KMeans(n_clusters=k)
kmeans.fit(X_real)
labels = kmeans.predict(X_real)
centroids = kmeans.cluster_centers_
X_array = X_real.values
generated_data=X_gan.values
plt.scatter(X_array[:, 0], X_array[:, 1], c=labels, s=50, cmap='viridis')
plt.scatter(generated_data[:, 0], generated_data[:, 1], s=50, color='pink', label='Datos ↪Generados')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=200, alpha=0.75)
plt.savefig(filename+"KNN.jpg")

```

```

plt.show()

# t-SNE
# Crea etiquetas para los datos (1 para reales, 0 para GAN)
labels = np.concatenate([np.ones(X_real.shape[0]), np.zeros(X_gan.shape[0])])
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_combined)

plt.scatter(X_tsne[labels==1, 0], X_tsne[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_tsne[labels==0, 0], X_tsne[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('t-SNE visualization')
plt.savefig(filename+"tsne.jpg")
plt.show()

#PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_combined)

# Dibujar el resultado
plt.scatter(X_pca[labels==1, 0], X_pca[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_pca[labels==0, 0], X_pca[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('PCA visualization')
plt.savefig(filename+"pca.jpg")
plt.show()

#KDE
column_names = X_real.columns.tolist()
random_columns = random.sample(column_names, 20)
for column_name in random_columns:
    sns.kdeplot(X_real[column_name], label='Real', color='blue')
    sns.kdeplot(X_gan[column_name], label='Generated', color='red')
plt.legend()
plt.title(f'Distribution for "{column_name}"')
plt.savefig(f"{filename}kde-{column_name}.jpg")
plt.show()

```

5.3.4. Procesamiento datos conversión E <->M

```

[4]: # Leer datos
expression_data = pd.read_csv(fnexpr)
methylation_data = pd.read_csv(fmnet)
assign_data = pd.read_csv(fnassig)

expression_data = expression_data.iloc[:, 1:]
methylation_data = methylation_data.iloc[:, 1:]
assign_data = assign_data.iloc[:, 2:]

# Convertir todas las columnas a tipo float
expression_data = expression_data.apply(pd.to_numeric, errors='coerce')
methylation_data = methylation_data.apply(pd.to_numeric, errors='coerce')
assign_data = assign_data.apply(pd.to_numeric, errors='coerce')

# Lidiar con valores NaN (si los hay). Pone 0(CAMBIAR)
#expression_data.fillna(0, inplace=True)
#methylation_data.fillna(0, inplace=True)
#assign_data.fillna(0, inplace=True)

#Normalizamos
scaler = MinMaxScaler()

```

```

expression_data_scaled = scaler.fit_transform(expression_data)
methylation_data_scaled = scaler.fit_transform(methylation_data)
assign_data_scaled = assign_data

# Convertir a DataFrame
expression_data = pd.DataFrame(expression_data_scaled, index=expression_data.index,columns=expression_data.columns)
methylation_data = pd.DataFrame(methylation_data_scaled, index=methylation_data.index,columns=methylation_data.columns)
assign_data = pd.DataFrame(assign_data_scaled, index=assign_data.index, columns=assign_data.columns)

#print(torch.cuda.is_available())
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device="cpu"

expression_data_tensor = torch.FloatTensor(expression_data.values).to(device)
methylation_data_tensor = torch.FloatTensor(methylation_data.values).to(device)
assign_data_tensor = torch.FloatTensor(assign_data.values).to(device)

# Dividir en conjuntos de entrenamiento y prueba
dataset = TensorDataset(expression_data_tensor, methylation_data_tensor)
train_size = int(0.9 * len(dataset)) # Ajusta esto según tu necesidad
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

# Convertir los Subset en tensores para el entrenamiento
expression_train, methylation_train= zip(*[(e, m) for e, m in train_dataset])
expression_train = torch.stack(expression_train)
methylation_train = torch.stack(methylation_train)
#assign_train = torch.stack(assign_train)

# Convertir los Subset en tensores para la validación/prueba
expression_test, methylation_test = zip(*[(e, m) for e, m in test_dataset])
expression_test = torch.stack(expression_test)
methylation_test = torch.stack(methylation_test)
#assign_test = torch.stack(assign_test)
input_dim = expression_data.shape[1]
output_dim = methylation_data.shape[1]

```

5.3.5. Validación modelo TRANSFORMER E <->M

[10]:

```

import torch
import pandas as pd
from sklearn.metrics import mean_squared_error
from torch.nn.functional import mse_loss
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import random

from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from scipy.spatial.distance import cdist
from scipy.stats import wasserstein_distance, ks_2samp
import sys
import umap
from scipy.stats import pearsonr

```

```

from math import sqrt

# Convertir los tensores a numpy y luego a DataFrame de pandas
expression_data_test_np = expression_test.cpu().numpy()
methylation_data_test_np = methylation_test.cpu().numpy()

expression_data_test_df = pd.DataFrame(expression_data_test_np)
methylation_data_test_df = pd.DataFrame(methylation_data_test_np)

X_real = pd.concat([expression_data_test_df, methylation_data_test_df], axis=1)

print(torch.cuda.is_available())
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device="cpu"

# Cargar el modelo Transformer previamente entrenado
model = TransformerModel(input_dim, output_dim, nhead, nhid, nlayers, dropout).to(device) ↴
# Asegúrate de proporcionar los parámetros correctos aquí
model.load_state_dict(torch.load(fnmodel))
model.eval()
model.to(device)

# Generar datos de metilación utilizando el modelo Transformer
with torch.no_grad():
    # Necesitamos crear una máscara de secuencia para la inferencia
    src_mask = generate_square_subsequent_mask(expression_test.size(0)).to(device)
    generated_methyl = model(expression_test, src_mask)

    # Convertir los datos generados a formato numpy
    generated_methyl = generated_methyl.cpu().numpy()

    generated_methyl_data = generated_methyl[:, :methylation_test.shape[1]]

    # Convertir a DataFrame de pandas
    generated_methyl_data_df = pd.DataFrame(generated_methyl_data)

X_gan = pd.concat([expression_data_test_df, generated_methyl_data_df], axis=1)
X_gan.columns = X_real.columns

# Concatena los datos reales con los generados
X_combined = np.vstack([X_real, X_gan])

# Calcular el MSE
mse = mean_squared_error(methylation_data_test_np, generated_methyl_data)

print(f'MSE entre los datos de metilación reales y los generados: {mse}')
rmse = sqrt(mse)
print(f'RMSE: {rmse}')
pcc, _ = pearsonr(methylation_data_test_np.flatten(), generated_methyl_data.flatten())
print(f'PCC: {pcc}')

# t-SNE
labels = np.concatenate([np.ones(X_real.shape[0]), np.zeros(X_gan.shape[0])])
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_combined)

plt.scatter(X_tsne[labels==1, 0], X_tsne[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_tsne[labels==0, 0], X_tsne[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('t-SNE visualization')
# plt.savefig(filename+"tsne.jpg")
plt.show()

# Configurando y entrenando UMAP
umap_model = umap.UMAP(n_neighbors=15, min_dist=0.1, n_components=2, random_state=42)

```

```

X_umap = umap_model.fit_transform(X_combined)

# Dibujando la visualización
plt.scatter(X_umap[labels==1, 0], X_umap[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_umap[labels==0, 0], X_umap[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('UMAP visualization')
#plt.savefig(filename+"umap.jpg")
plt.show()

#PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_combined)

plt.scatter(X_pca[labels==1, 0], X_pca[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_pca[labels==0, 0], X_pca[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('PCA visualization')
#plt.savefig(filename+"pca.jpg")
plt.show()

```

5.3.6. Validación modelo GAN E <->M

```

[4]:
import torch
import pandas as pd
from sklearn.metrics import mean_squared_error
from torch.nn.functional import mse_loss
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import random

from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from scipy.spatial.distance import cdist
from scipy.stats import wasserstein_distance, ks_2samp
import sys
import umap
from scipy.stats import pearsonr
from math import sqrt

# Convertir los tensores a numpy y luego a DataFrame de pandas
expression_data_test_np = expression_test.cpu().numpy()
methylation_data_test_np = methylation_test.cpu().numpy()

expression_data_test_df = pd.DataFrame(expression_data_test_np)
methylation_data_test_df = pd.DataFrame(methylation_data_test_np)

X_real = pd.concat([expression_data_test_df, methylation_data_test_df], axis=1)

print(torch.cuda.is_available())
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device="cpu"

```

```

# Carga del modelo previamente entrenado
gen = Generator(expression_data.shape[1], methylation_data.shape[1]).to(device)
gen.load_state_dict(torch.load(fnmmodel))
gen.eval()
gen.to(device)

# Pasar todos los datos de expr.csv a través del generador
with torch.no_grad():
    generated_methyl = gen(expression_test)

    generated_methyl = generated_methyl.cpu().numpy()

    generated_methyl_data = generated_methyl[:, :methylation_test.shape[1]]

    generated_methyl_data_df=pd.DataFrame(generated_methyl_data)
    X_gan = pd.concat([expression_data_test_df, generated_methyl_data_df], axis=1)

    X_gan.columns = X_real.columns

# Concatena los datos reales con los generados
X_combined = np.vstack([X_real, X_gan])

# Calcular el MSE
mse = mean_squared_error(methylation_data_test_np, generated_methyl_data)

print(f"MSE entre los datos de metilación reales y los generados: {mse}")
rmse = sqrt(mse)
print(f"RMSE: {rmse}")

pcc, _ = pearsonr(methylation_data_test_np.flatten(), generated_methyl_data.flatten())
print(f"PCC: {pcc}")

#t-SNE
labels = np.concatenate([np.ones(X_real.shape[0]), np.zeros(X_gan.shape[0])])
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_combined)

plt.scatter(X_tsne[labels==1, 0], X_tsne[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_tsne[labels==0, 0], X_tsne[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('t-SNE visualization')
plt.savefig(filename+"tsne.jpg")
plt.show()

# Configurando y entrenando UMAP
umap_model = umap.UMAP(n_neighbors=15, min_dist=0.1, n_components=2, random_state=42)
X_umap = umap_model.fit_transform(X_combined)

# Dibujando la visualización
plt.scatter(X_umap[labels==1, 0], X_umap[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_umap[labels==0, 0], X_umap[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('UMAP visualization')
plt.savefig(filename+"umap.jpg")
plt.show()

#PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_combined)

plt.scatter(X_pca[labels==1, 0], X_pca[labels==1, 1], c='blue', label='Real', s=3)
plt.scatter(X_pca[labels==0, 0], X_pca[labels==0, 1], c='red', label='Generated', s=3)
plt.legend()
plt.title('PCA visualization')

```

```
plt.savefig(filename+"pca.jpg")
plt.show()
```