TRABAJO FIN DE MASTER

ENHANCING THE EXFOR NUCLEAR DATA LIBRARY USING MACHINE LEARNING TECHNIQUES

TRABAJO FIN DE MASTER
PARA LA OBTENCIÓN DEL
TÍTULO DE **MASTER EN CIENCIA Y TECNOLOGÍA NUCLEAR**

SEPTIEMBRE 2023

Juan Antonio Monleón de la Lluvia Mazagatos

DIRECTORES DEL TRABAJO FIN DE MASTER:

Oscar Cabellos

INDUSTRIALES
ETSII | UPM

POLITÉCNICA

"All models are wrong, but some are useful."

- George E. P. Box

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my mother, for her continuous support and encouragement throughout my life, and for always believing in me, even when I didn't believe in myself.

I am also grateful to my thesis mentor, Dr. Oscar Cabellos, for giving me the opportunity to work on this project and for his guidance and support throughout the process. His expertise and knowledge were invaluable to me.

I would also like to thank the Nuclear Department of the UPM and Eduardo Gallego in particular, for providing me with a grant that made this research possible.

Finally, I would like to thank all of the other people who have helped me along the way, including my friends, family, colleagues, and mentors. I could not have done this without you.

RESUMEN EJECUTIVO

La detección de valores atípicos es crucial en la física experimental, especialmente en el ámbito de las reacciones nucleares. Estos valores pueden sesgar análisis y generar resultados poco fiables. Esta problemática cobra especial importancia en el tratamiento de datos relativos a protones incidentes, los cuales son cada vez más relevantes en terapias médicas como el tratamiento de cáncer mediante protonterapia. En esta técnica, se usan haces de protones para irradiar células cancerígenas con alta precisión, minimizando daños a los tejidos sanos circundantes. Debido a su eficacia y precisión, la protonterapia se está consolidando como una opción de tratamiento preferente para ciertos tipos de cáncer. Sin embargo, la presencia de valores atípicos en los datos experimentales sobre protones incidentes puede comprometer la calidad y seguridad de estos tratamientos. Por tanto, es fundamental llevar a cabo un riguroso proceso de detección y eliminación de valores atípicos para asegurar resultados fiables y precisos en el ámbito de la física nuclear aplicada a la medicina y demás campos.

El principal objetivo de esta tesis de máster es evaluar la aplicabilidad de diversas técnicas de aprendizaje automático para la detección de valores atípicos en la biblioteca de datos nucleares EXFOR. No se trata solo de implementar algoritmos, sino de adaptarlos eficazmente al conjunto de datos especializado y evaluar cuáles resultan más eficaces.

Un elemento clave de la investigación es el preprocesado de datos, dado que es esencial para el éxito de cualquier algoritmo de aprendizaje automático. Dada la complejidad de la estructura de datos de EXFOR, se dedicarán esfuerzos significativos para investigar métodos eficientes de extracción, transformación y preparación de los datos. Sobre esta base, se implementarán diversos algoritmos para la detección de valores atípicos, incluidos métodos estadísticos tradicionales y técnicas más avanzadas como redes neuronales, en este caso un Autoencoder. Posteriormente, se compararán críticamente los métodos y resultados mediante análisis visuales y cuantitativos, con el fin de evaluar la fiabilidad, ventajas y limitaciones de cada enfoque.

El estudio tiene el objetivo de contextualizar estos hallazgos en la literatura científica, ofreciendo perspectivas sobre los retos y el potencial futuro de aplicar el aprendizaje automático para la detección de valores atípicos en datos nucleares.

Con el contexto y los objetivos ya establecidos, es importante profundizar en algunos de los conceptos más esenciales para esta investigación, como son son la librería de datos nucleares EXFOR, la detección de valores atípicos y la naturaleza del aprendizaje automático.

EXFOR (EXchange FORmat) es una biblioteca extensa y meticulosa de datos de reacciones nucleares experimentales, desarrollada para facilitar la investigación y aplicaciones en el campo nuclear. Creada para promover la intercambio de datos entre centros de investigación, EXFOR es gestionada por la Red Internacional de Centros de Datos de Reacciones Nucleares (NRDC), bajo supervisión de la Sección de Datos Nucleares de la AIEA. Con más de 24,000 experimentos registrados hasta octubre de 2022, la plataforma busca ser una fuente completa y confiable de datos en reacciones inducidas por neutrones a baja energía, partículas cargadas y fotones. Este recurso es especialmente relevante para investigadores y profesionales en ciencias y tecnologías nucleares, incluyendo a académicos, laboratorios nacionales e industria, quienes se centran en el diseño y análisis de reactores nucleares, ciclos de combustible y otras aplicaciones nucleares.

El Aprendizaje Automático (AA) es una subdisciplina de la Inteligencia Artificial (IA) que se enfoca en diseñar sistemas que puedan aprender a tomar decisiones o hacer predicciones a partir de datos. A diferencia de programar tareas de manera manual, estos sistemas mejoran su rendimiento de forma adaptativa al ser expuestos a nuevos datos. En el mundo actual, saturado

de información, la capacidad del AA para manejar conjuntos de datos grandes y complejos ha resultado ser invaluable. Además, se pueden distinguir principalmente tres tipos de aprendizaje automático: el aprendizaje supervisado, donde el modelo se entrena con datos etiquetados; el aprendizaje no supervisado, que trabaja con datos no etiquetados para encontrar patrones; y el aprendizaje por refuerzo, donde un agente aprende a tomar decisiones mediante ensayo y error.

Estos tres tipos de aprendizaje automático, aunque distintos, comparten un objetivo común: aprender de los datos para hacer decisiones inteligentes o predicciones. La principal diferencia radica en el tipo de retroalimentación que reciben y el contexto de aprendizaje, que puede ir desde tener un mapeo claro de entradas a salidas en aprendizaje supervisado, hasta aprender de una serie de acciones y recompensas en un contexto dinámico en el caso del aprendizaje por refuerzo.

Es importante además diferenciar entre los términos relacionados pero no idénticos de Inteligencia Artificial, Aprendizaje Automático y Aprendizaje Profundo. La IA es el término más amplio que abarca cualquier técnica que permita a las máquinas imitar la inteligencia humana. El Aprendizaje Automático es una especialización dentro de la IA que usa técnicas estadísticas para mejorar con la experiencia. El Aprendizaje Profundo, por otro lado, es un subcampo del Aprendizaje Automático que se inspira en la estructura y función del cerebro humano y se aplica en tecnologías como los coches autónomos para tareas como la identificación de objetos.

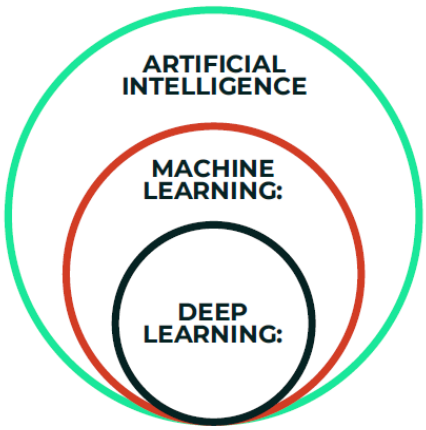

Figure 0.1: Estructura anidada de la Inteligencia Artificial [1].

Los valores atípicos son observaciones que se desvían significativamente del resto de los datos y pueden afectar los resultados del análisis de forma adversa. Su detección y gestión son cruciales en cualquier análisis de datos. Existen dos enfoques principales para detectarlos: métodos estadísticos y métodos basados en aprendizaje automático. Los métodos estadísticos, como el Z-Score y el IQR, son más adecuados para datos univariantes, mientras que las técnicas de aprendizaje automático, como los algoritmos de agrupación y detección de anomalías, son más versátiles y no requieren suposiciones sobre la distribución de los datos. La elección del método depende de las características del conjunto de datos y del objetivo del análisis. Sin embargo, es esencial tener en cuenta que no todos los valores atípicos son indeseables o errores. En algunos contextos, pueden señalar áreas para una investigación más detallada, especialmente si los datos son escasos o emergentes en esa área particular. Por lo tanto, el tratamiento de los valores atípicos debe hacerse con un entendimiento cuidadoso de su contexto.

La extracción y el preprocesamiento de datos constituyen, probablemente, la etapa más crucial en el proceso de implementación de algoritmos de aprendizaje automático (machine learning).

Contar con una base de datos limpia y adecuadamente estructurada es fundamental para el óptimo rendimiento de estos algoritmos. A continuación, se describen los aspectos clave de la metodología desarrollada para esta tarea, desde la selección de la fuente de datos hasta la generación de la base de datos final sobre la cual se aplicarán técnicas de aprendizaje automático.

- **Fuente de Datos:** El conjunto de datos principal se extrae de la biblioteca de datos nucleares EXFOR, que es una colección exhaustiva pero compleja que abarca más de 24,000 experimentos con características variadas.
- **Análisis de Datos con `EXFORTABLES`:** Para facilitar la manipulación y la legibilidad de los datos de EXFOR, se utiliza `EXFORTABLES` para descomponer la biblioteca en directorios estructurados lógicamente, estableciendo así un punto de partida efectivo para la extracción y manipulación de datos.
- **Comprensión Objetiva:** El primer paso consiste en identificar el tipo específico de información que se necesita para las aplicaciones de aprendizaje automático. El enfoque se centra en determinar la estructura en la que dicha información debería almacenarse para maximizar su utilidad.
- **Estructuración de Datos en DataFrames:** Se utilizan los dataframes de Python como la estructura de datos preferente, facilitando tanto la manipulación de datos como su almacenamiento eficiente. En estos dataframes, cada fila representa un punto de datos experimental y cada columna representa una característica.
- **Codificación de Datos:** Debido a que los algoritmos de aprendizaje automático suelen requerir datos numéricos, los valores categóricos presentes en el conjunto de datos se transforman en representaciones numéricas, a menudo mediante la técnica de codificación *One-hot*.
- **Diseño Orientado a Objetos en Python:** Se implementa una clase llamada `Experiment` para encapsular todos los atributos y comportamientos relacionados con un experimento individual. Esto permite aprovechar las ventajas inherentes a la Programación Orientada a Objetos (OOP), como la modularidad y la reutilización de código.
- **Capacidades Funcionales:** Se desarrollan diversas funciones con el objetivo de manipular y consultar el conjunto de datos de manera eficaz. Estas funciones incluyen, pero no se limitan a, el filtrado de atributos, la clasificación de experimentos y las capacidades para generar gráficos.
- **Agrupación de Datos para Análisis:** Con un enfoque especial en la detección de valores atípicos, los datos se agrupan en función de la variable que se está midiendo, lo que facilita las etapas posteriores del análisis de datos.
- **Conversión de Datos en DataFrame Final:** Finalmente, los datos organizados y clasificados se consolidan en un dataframe final, conforme a las especificaciones previamente establecidas.

Cada uno de los puntos clave descritos contribuye a asegurar que los datos se encuentren en un formato limpio, estructurado y apto para análisis, dejando el terreno preparado para la aplicación de las técnicas de aprendizaje automático. Este proceso de preparación y análisis de datos no solo se documenta detalladamente en el texto, sino que también se ha implementado de forma práctica a través de varios ficheros de Python. El código fuente de estas funciones se encuentra en los anexos de este trabajo y está disponible para su descarga y uso libre en GitHub EXFOR-ProtonReactions-Analysis Repository.

Además, el código en este repositorio, en conjunción con `EXFORTABLES`, permite reproducir todos los análisis realizados en este trabajo. Sin embargo, la utilidad del código trasciende el mero preprocesamiento de datos para el aprendizaje automático. Una vez que los experimentos están cargados, las funciones implementadas ofrecen una serie de herramientas que facilitan tareas como consultar información de cualquier experimento, graficar múltiples experimentos simultáneamente, filtrar experimentos en base a distintos atributos y clasificar los experimentos según las necesidades específicas. Esto convierte el preprocesamiento en una herramienta versátil, útil no solo para los objetivos de este trabajo sino también para cualquier otro análisis de datos de EXFOR que se desee realizar.

El siguiente paso crucial en nuestra investigación involucra la identificación y gestión de los valores atípicos o 'outliers'. En este sentido, hemos implementado varias técnicas de detección de outliers, tanto estadísticas como basadas en aprendizaje automático, para asegurarnos de que nuestra análisis es tanto robusto como exhaustivo. Cada uno de estos métodos ha sido cuidadosamente implementado en archivos Jupyter Notebooks, que también están disponibles para consulta y uso en el repositorio de GitHub previamente mencionado.

Método IQR (Rango Intercuartílico): Este método estadístico es especialmente eficaz para identificar *outliers* en una distribución univariante. Utiliza el rango intercuartílico, que es la diferencia entre el tercer cuartil (Q3) y el primer cuartil (Q1) del conjunto de datos. Los datos que caen fuera de los límites establecidos por $Q1 - 1.5 \times \text{IQR}$ y $Q3 + 1.5 \times \text{IQR}$ se consideran valores atípicos. Este enfoque es robusto y eficiente para distribuciones que no necesariamente siguen una normalidad.

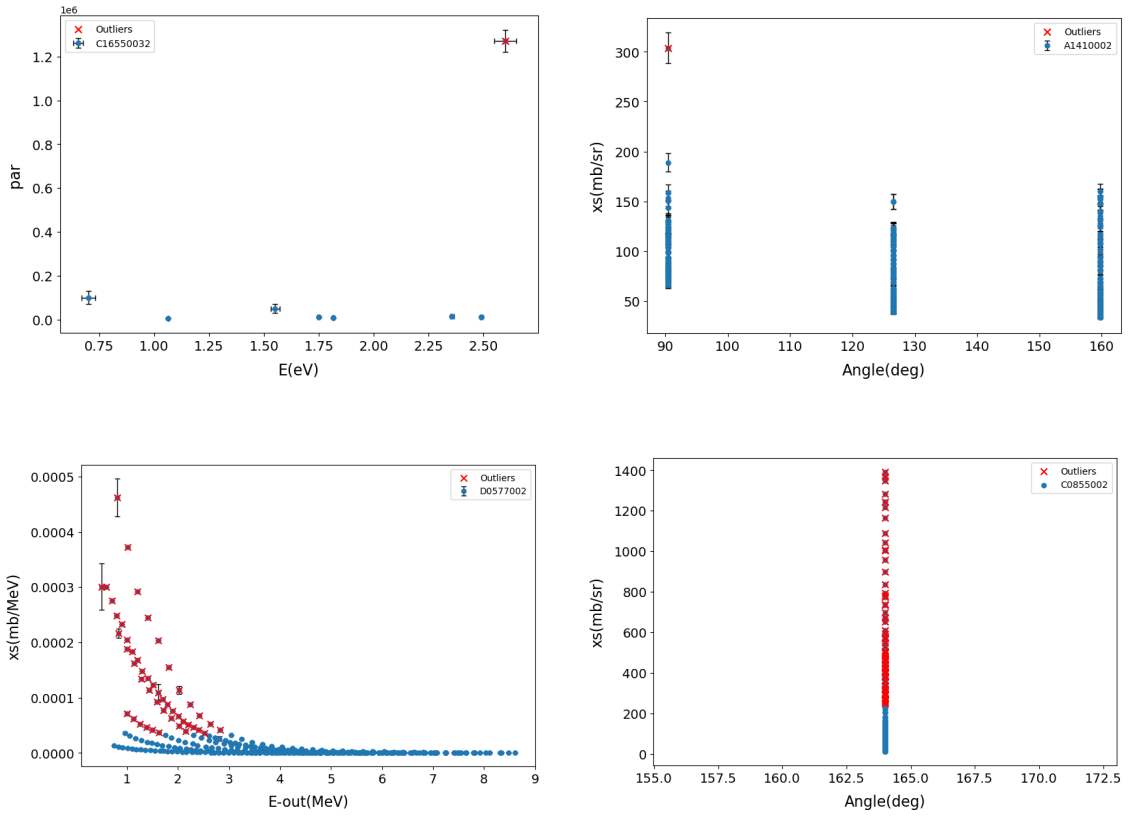

Figure 0.2: Detección de valores atípicos utilizando el método IQR.

Algunos de los resultados de aplciar este método se muestran en la Figura 0.2. En general el

método IQR es una técnica estadística que incorpora incertidumbres en sus cálculos, lo cual es beneficioso para minimizar falsos positivos. No obstante, nuestra investigación señala que el uso del método IQR se ve limitado para nuestros conjuntos de datos específicos. Este método muestra un rendimiento óptimo con datos de distribución uniforme, lo cual no corresponde al tipo de datos que estamos manejando.

En particular, cualquier pico en los datos, derivado de su distribución inherente, corre el riesgo de ser incorrectamente etiquetado como un valor atípico. Esto limita la utilidad del método IQR, ya que no puede distinguir de manera fiable entre valores atípicos auténticos y puntos de datos que forman parte natural de una distribución no uniforme.

Isolation Forest: Este método es una técnica de aprendizaje automático para la detección de anomalías que se basa en árboles de decisión. A diferencia de otros métodos, Isolation Forest no requiere un modelo previo de los datos "normales". En lugar de eso, el algoritmo funciona aislando registros que son anómalos. Particularmente, el método selecciona al azar una característica y un valor para esa característica, y utiliza estas condiciones para dividir los datos. Este proceso se repite de manera recursiva hasta que los datos quedan completamente aislados. Cuanto más corto sea el camino para aislar un registro, más probable es que sea una anomalía. Isolation Forest es especialmente eficiente para conjuntos de datos de gran tamaño y se destaca por su capacidad para detectar outliers en un tiempo computacional relativamente bajo.

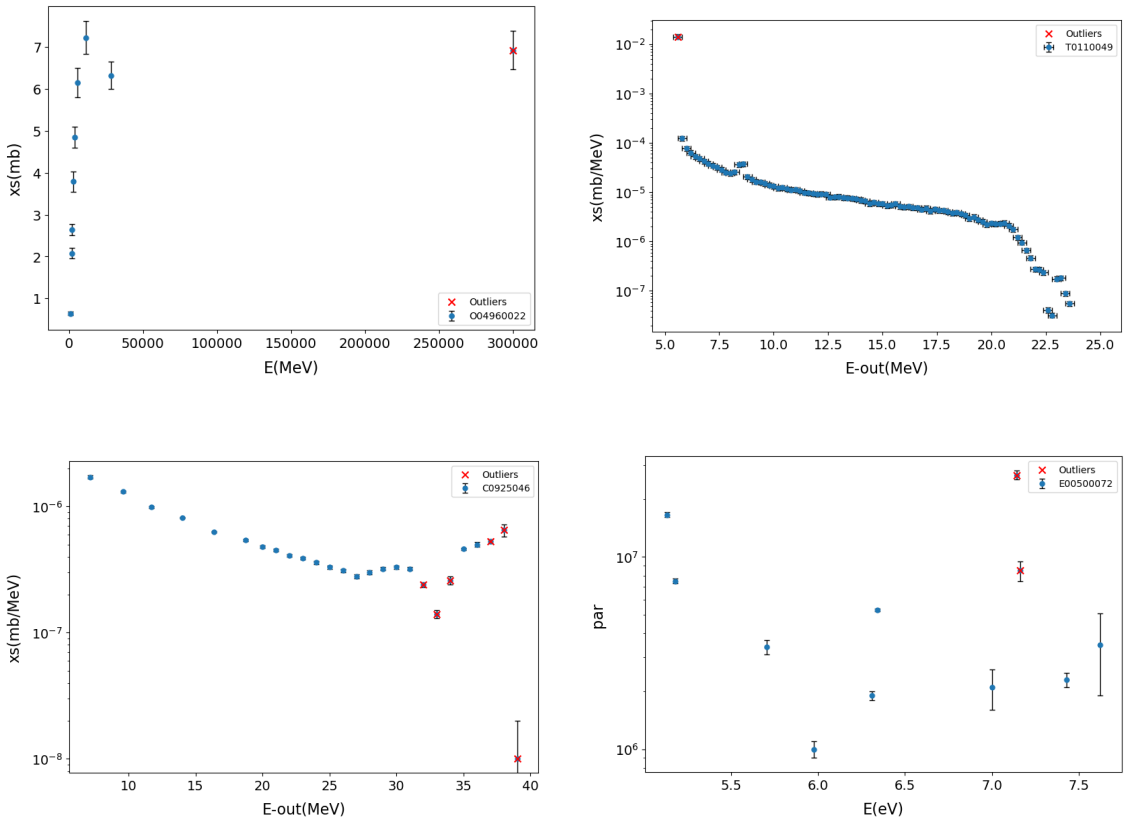

Figure 0.3: Detección de valores atípicos utilizando el método Isolation Forest.

Como se puede observar en la Figura 0.3, el método de Isolation Forest demuestra su eficacia en la detección de valores atípicos y, en general, supera a enfoques estadísticos tradicionales como el método IQR. No obstante, este algoritmo muestra una alta sensibilidad a la distribución específica del conjunto de datos evaluado. Una limitación notable, es su manejo de áreas de baja

densidad en el conjunto de datos. Puntos que están aislados debido a un muestreo insuficiente pueden ser erróneamente etiquetados como atípicos, lo que representa un desafío considerable en el contexto de nuestro estudio.

DBSCAN: Este método de aprendizaje automático es especialmente útil para identificar conglomerados en un conjunto de datos en función de la densidad de los puntos. DBSCAN agrupa puntos que están próximos entre sí según un umbral de distancia y requiere un número mínimo de puntos para formar un grupo. Los puntos que no pertenecen a ningún grupo se consideran outliers. A diferencia de otros métodos de clustering como K-means, DBSCAN no requiere especificar el número de grupos a priori y puede encontrar grupos de formas arbitrarias, lo que lo hace particularmente útil para la detección de outliers en conjuntos de datos complejos.

Según los resultados presentados en la Figura 0.4, el método DBSCAN se configura como otra herramienta útil de aprendizaje automático para la detección de valores atípicos. Este enfoque se basa en la densidad de puntos en el conjunto de datos, lo que lo convierte en una opción especialmente apta para manejar conjuntos de datos con densidades variables. Para el presente estudio, se han utilizado los hiperparámetros predeterminados de DBSCAN en una fase exploratoria inicial. Sin embargo, es importante subrayar, que el algoritmo es particularmente sensible a la selección de estos parámetros.

Es fundamental destacar que, DBSCAN puede ser computacionalmente costoso, más aún con conjuntos de datos de gran tamaño. En nuestro contexto específico, se hizo necesario subdividir alguno de los conjuntos de datos para facilitar un procesamiento computacional viable del algoritmo.

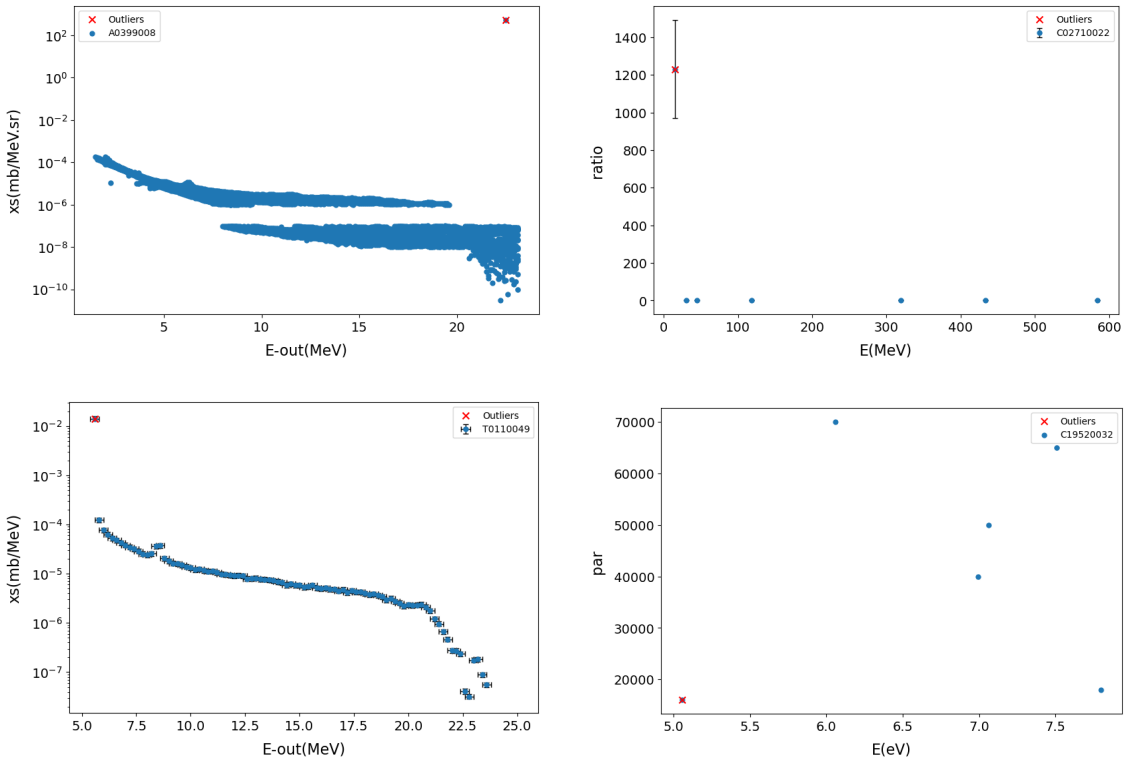

Figure 0.4: Detección de valores atípicos utilizando el método DBSCAN.

LOF (Local Outlier Factor): Este algoritmo se utiliza para identificar outliers en conjuntos de datos basándose en la densidad local de los puntos. En lugar de medir la distancia global desde un punto a todos los demás en el conjunto de datos, LOF compara la densidad de un

punto con la de sus vecinos más cercanos. Si un punto tiene una densidad significativamente menor que la de sus vecinos, se le asigna un factor de atipicidad alto, lo que lo identifica como un outlier. Este enfoque es especialmente eficaz en conjuntos de datos donde la densidad de los puntos no es uniforme, permitiendo una detección de outliers más precisa en tales escenarios.

Como se ilustra en la Figura 0.5, el método Local Outlier Factor (LOF) destaca por su versatilidad en adaptarse a diversas distribuciones de datos. A pesar de esta ventaja, el método no está exento de los retos habituales que enfrentan los algoritmos de detección de valores atípicos, tales como la sensibilidad a las particularidades del conjunto de datos. Este problema se intensifica cuando el conjunto de datos presenta huecos o falta de información en ciertos rangos, incrementando así la probabilidad de detectar falsos positivos.

La selección de los hiperparámetros es un elemento esencial para la eficacia del LOF. Este requerimiento puede convertir el proceso en una tarea pesada, ya que distintos conjuntos de datos pueden necesitar configuraciones variadas. Sin embargo, una de las fortalezas de LOF es su robustez para evitar falsos positivos en casos donde los datos presentan picos o agrupaciones pronunciadas. Este es un escenario en el que otros métodos de detección de valores atípicos suelen ser menos efectivos.

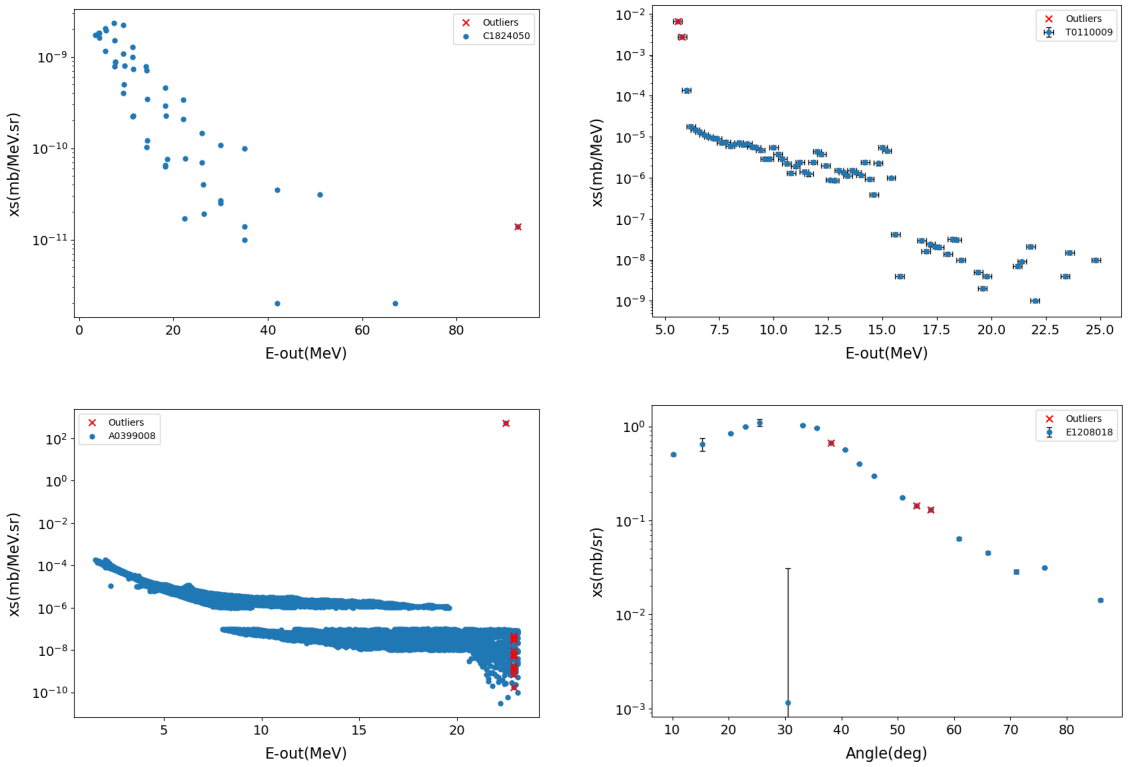

Figure 0.5: Detección de valores atípicos utilizando el método LOF.

SVM (Support Vector Machine): En el contexto de detección de outliers, se suele utilizar una variante de SVM conocida como One-class SVM. Este algoritmo intenta encontrar el hiperplano en un espacio de características de alta dimensión que mejor separa los puntos de datos "normales" de cualquier potencial outlier. La idea es maximizar el margen entre los puntos de datos y el origen en este espacio transformado. Los puntos que queden del lado incorrecto del hiperplano se consideran outliers. Este método es particularmente útil para conjuntos de datos de alta dimensión y es eficaz incluso cuando los datos son no lineales.

Conforme a los datos presentados en la Figura 0.6, el algoritmo SVM resultó ser efectivo en

la detección de valores atípicos en una variedad de conjuntos de datos. Al usar parámetros predeterminados como punto de partida, el método demostró su robustez en diferentes escenarios. Aunque estos parámetros son un buen inicio, podrían optimizarse más mediante técnicas específicas o conocimiento del dominio.

Similar a lo que ocurrió con el DBSCAN, el algoritmo SVM demanda una alta capacidad computacional, especialmente cuando se trata de grandes conjuntos de datos. Este reto fue palpable en nuestro estudio, donde fue necesario segmentar el conjunto inicial de datos en subsets más manejables para cumplir con las demandas computacionales.

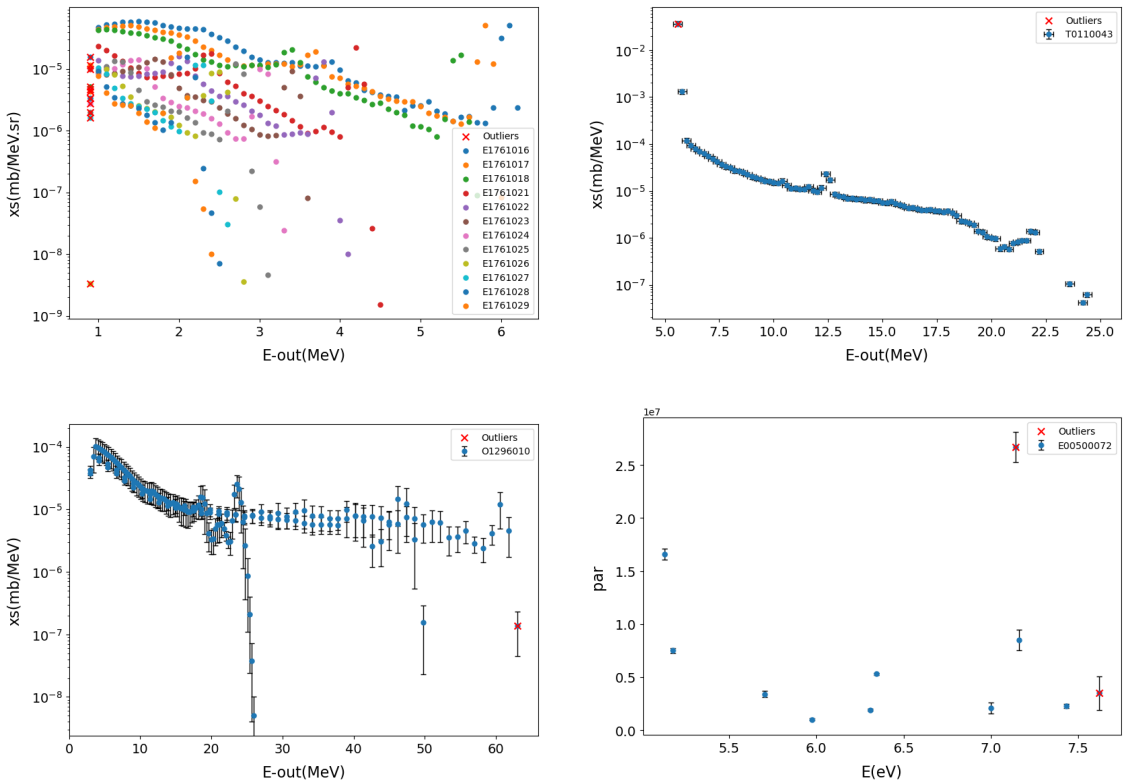

Figure 0.6: Detección de valores atípicos utilizando el método SVM.

Autoencoder: Los Autoencoders son una clase de redes neuronales utilizadas generalmente para la reducción de dimensionalidad o para el aprendizaje de representaciones. La red está compuesta típicamente por una capa de entrada, una o más capas ocultas, y una capa de salida. La red se entrena para reconstruir su entrada, forzando así a la red a aprender representaciones internas eficientes de los datos de entrada.

Arquitectura y Creación del Modelo: La red consta de dos partes principales: el codificador y el decodificador. El codificador comprime la entrada y la representa en un espacio latente de menor dimensionalidad. El decodificador toma esta representación y la reconstruye para producir una salida lo más cercana posible a la entrada original. Se entrena el modelo minimizando la diferencia entre la entrada y la salida, comúnmente usando una función de pérdida como el error cuadrático medio.

Aplicación para la Detección de Outliers: Una vez entrenado el autoencoder en datos "normales", se puede utilizar para detectar outliers al medir cuán bien el modelo puede reconstruir nuevas instancias de datos. Si un punto de datos es una observación atípica, la red tendrá dificultades para reconstruirlo correctamente, resultando en un alto valor de error de reconstrucción. Este

valor se puede utilizar como un indicador para identificar outliers en el conjunto de datos.

Según lo mostrado en la Figura 0.7, los autoencoders, que son algoritmos basados en redes neuronales, se diferencian significativamente de los métodos de aprendizaje automático tradicionales tratados en esta tesis. A pesar de requerir un nivel más profundo de entendimiento para su implementación efectiva, los beneficios potenciales de usar estas redes son considerables. Futuros trabajos podrían focalizarse en ajustar diversos parámetros, tales como el número de capas, funciones de activación y tasas de aprendizaje.

En términos de eficacia para la detección de valores atípicos, el autoencoder muestra resultados prometedores. Logró identificar diversas instancias de posibles valores atípicos con menos falsos positivos que otros métodos, aunque su rendimiento es altamente dependiente de la configuración de parámetros.

Un factor crítico para el desempeño del modelo es la cantidad de datos disponibles para entrenamiento. En ciertos casos, el tamaño limitado de los conjuntos de datos afectó negativamente el rendimiento del modelo. Una táctica alternativa podría ser el uso de un conjunto de datos más amplio, aunque esta estrategia no garantiza necesariamente una mejora en el rendimiento.

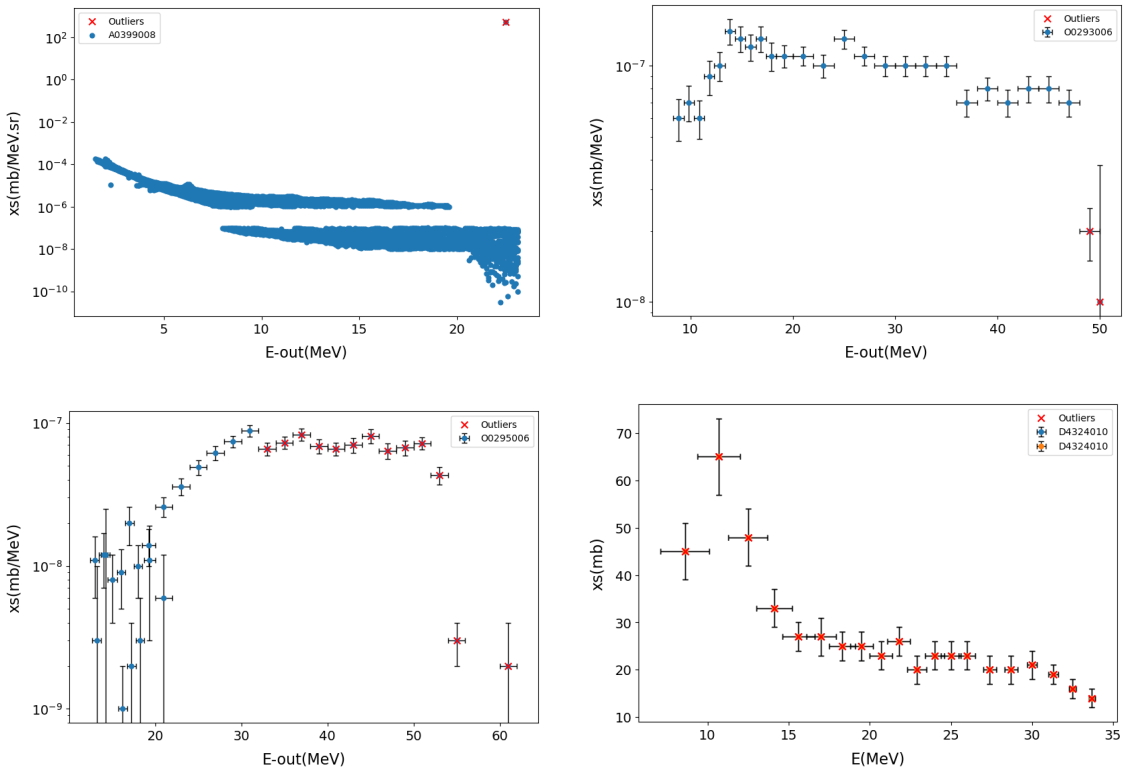

Figure 0.7: Detección de valores atípicos utilizando el método Autoencoder.

En conclusión, cada método para la detección de valores atípicos examinado en este estudio tiene sus propias ventajas y limitaciones. La efectividad de estos métodos varía según las características específicas de los conjuntos de datos analizados; algunos sobresalen en áreas donde otros pueden tener deficiencias. En el ámbito de las técnicas de aprendizaje automático, es difícil determinar un método definitivamente superior. Aunque DBSCAN mostró resultados ligeramente mejores, su eficacia depende en gran medida de la naturaleza de los datos y de los ajustes específicos elegidos para cada análisis. Los métodos basados en redes neuronales, como los autoencoders, identificaron menos valores atípicos, lo que podría sugerir una mayor

precisión. No obstante, los mayores requisitos computacionales y la complejidad en la configuración de parámetros de las redes neuronales plantean interrogantes sobre su idoneidad práctica para la tarea. Por lo tanto, cada técnica aporta su propia perspectiva para la identificación de valores atípicos, y su éxito relativo está vinculado a las complejidades de los conjuntos de datos a los que se aplican y a los objetivos específicos que buscan cumplir.

Es fundamental entender que los algoritmos de aprendizaje automático están diseñados para identificar un porcentaje fijo de valores atípicos, lo que puede llevar a falsos positivos. Por ello, la intervención de un experto en el dominio es crucial para interpretar adecuadamente los resultados. Dada la gran cantidad de datos en la biblioteca EXFOR, un análisis manual no sería práctico, resaltando la importancia de las técnicas de aprendizaje automático en este estudio. Este trabajo aporta una herramienta gratuita y fácil de usar para revisar datos, accesible para cualquiera con un ordenador con Python.

En definitiva, esta tesis de máster se ha centrado en la incorporación de técnicas de aprendizaje automático para la detección de valores atípicos en la biblioteca de datos nucleares EXFOR para reacciones inducidas por protones. El estudio ha desarrollado una metodología robusta específicamente adaptada para la complejidad de los datos de EXFOR. No solo se crearon estrategias eficientes para la extracción y preparación de datos, sino que también se estableció un marco de trabajo que sirve como modelo para futuras investigaciones en este campo.

Se exploraron diversas técnicas para la detección de valores atípicos, desde métodos estadísticos tradicionales como el rango intercuartílico (IQR) hasta algoritmos de aprendizaje automático más sofisticados como Isolation Forest, DBSCAN y el Local Outlier Factor (LOF). Además, se incorporó un modelo de red neuronal, específicamente un Autoencoder, en nuestro conjunto de algoritmos.

El estudio ha cumplido con todos los objetivos iniciales y ha establecido una metodología sólida para aplicar el aprendizaje automático en la detección de valores atípicos en la biblioteca EXFOR. La metodología desarrollada es lo suficientemente versátil como para extenderse fácilmente a otras secciones de la biblioteca EXFOR, lo que abre la puerta a mejoras integrales. Para fomentar el desarrollo y la participación comunitaria, todos los scripts y códigos generados están disponibles de forma pública en GitHub, lo que no solo cumple con el imperativo académico de compartir conocimiento, sino que también prepara el terreno para esfuerzos colaborativos futuros para refinar y ampliar el trabajo presentado.

Códigos UNESCO

- Campo:
  - 33 - Ciencias de la Tecnología
  - 12 - Ciencias de la Computación e Información
  - 22 - Física
- Disciplina:
  - 3304 - Tecnología
  - 3307 - Tecnología de los reactores nucleares
  - 3320 - Ingeniería y Tecnología Nuclear
  - 1203 - Ciencias de la Computación
  - 2202 - Física Nuclear
- Subdisciplina:
  - 330708 - Ingeniería de reactores
  - 332099 - Otras
  - 120326 - Simulación
  - 120317 - Redes Neuronales

Palabras Clave

- EXFOR Nuclear Data Library
- Machine Learning
- Proton-Induced Reactions
- Outlier Detection
- Isolation Forest
- DBSCAN
- Local Outlier Factor (LOF)
- Support Vector Machines (SVM)
- Interquartile Range (IQR)
- Neural Network Autoencoder
- Data Preprocessing
- Data Mining

ABSTRACT

The EXFOR nuclear data library is an indispensable resource for nuclear physicists and engineers. However, the database is susceptible to outliers and inconsistencies that can significantly impair its utility. This thesis focuses on the application of machine learning techniques for identifying potential outliers in proton-induced reactions within the EXFOR library. A comprehensive literature review on machine learning techniques was conducted to identify appropriate methods for outlier detection. Several algorithms, including Isolation Forest, DBSCAN, Local Outlier Factor (LOF), Support Vector Machines (SVM), along with a statistical method based on the Interquartile Range (IQR) and a neural network-based autoencoder, were rigorously tested on their capability to flag outliers effectively.

Extensive preprocessing work was carried out to prepare the data for machine learning applications. This involved creating a user-friendly tool that facilitates easy data mining of the EXFOR library, thereby serving as an invaluable resource for other researchers. Our results demonstrate that machine learning techniques can significantly improve the reliability and accuracy of the EXFOR database, with implications for a wide range of applications in nuclear science and engineering.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Background and Motivation

Outliers are observations that deviate markedly from other observations in a sample. They may indicate bad data, incorrect coding, or experimental errors. Outliers can also affect the accuracy and reliability of statistical analyses, especially when dealing with small sample sizes or non-normal distributions. Therefore, it is important to detect and treat outliers appropriately in experimental physics [2].

One area of experimental physics that requires high-quality data is nuclear reaction data, which is used for various applications such as nuclear energy, astrophysics, medical physics, and radiation protection. Nuclear reaction data are compiled in the EXFOR library [3], which contains more than 24,000 experiments involving neutrons, charged particles, and photons.

A specific type of nuclear reaction data that has gained increasing interest and necessity is incident proton data. One of the main reasons for this is that in recent years, the development of proton therapy medical treatments has gained significant interest and importance in the field of cancer treatment. This method uses proton beams to deliver radiation to cancer cells with precision, causing less damage to surrounding healthy tissues, fewer side effects, and improved outcomes than traditional radiotherapy techniques. Due to its high accuracy and effectiveness, proton therapy has become an increasingly common treatment option for certain types of cancer [4][5][6].

However, incident proton data are also subject to outliers due to various sources of uncertainty such as measurement errors, calibration errors, model uncertainties, or physical phenomena. Outliers can affect the accuracy and precision of incident proton data and consequently affect the quality and safety of proton therapy treatments. Precise knowledge of the physical properties of the proton beam and its interaction with biological tissues is necessary [7]. This knowledge is often obtained through the analysis of experimental data from proton incident particle experiments, which provide information on the behavior of the proton beam under different conditions. Therefore, the detection and removal of outliers from experimental data is an important step in obtaining reliable and accurate results.

1.2 Objectives

The primary aim of this master's thesis is to assess the implementation of various machine learning techniques for outlier detection in the EXFOR nuclear data library. In pursuit of this aim, a key objective is to develop a robust methodology that can effectively tackle this complex task. The overarching goal extends beyond mere implementation; it also includes evaluating the feasibility of applying machine learning to this specialized data set. The intent is not just to apply machine learning algorithms, but to investigate how well these algorithms can be tailored to suit the data and task at hand, and to identify which techniques emerge as most effective.

A focal point of the investigation lies in the initial stages of data mining and preprocessing, which are critical for the successful application of any machine learning algorithm. Given the complexity of EXFOR's data structures, significant efforts will be invested in researching various methods for efficiently extracting, transforming, and preparing the data for subsequent machine learning techniques.

Building upon this foundational work, the study will implement a range of outlier detection algorithms. These include traditional statistical methods like the Interquartile Range (IQR), tree-based methods such as Isolation Forest, density-based techniques like DBSCAN, and distance-based metrics like the Local Outlier Factor (LOF). Furthermore, to explore the frontiers of what machine learning can offer, a neural network model, specifically an Autoencoder, will also be deployed. This involves intricate steps from model building and parameter optimization to performance evaluation in the context of outlier detection.

The various methods and their results will then be critically compared through visual and quantitative analyses. Through this comparative lens, the study aims to gauge the reliability, advantages, and limitations inherent in each approach. Finally, the findings will be contextualized within the broader scientific literature, providing insights into the future potential and challenges of leveraging machine learning for outlier detection in the EXFOR dataset.

1.3 Thesis Structure

The structure of this thesis is organized into distinct chapters, each serving a specific role and contributing to the overall research objectives:

- **State of Art:** This opening chapter lays the groundwork by reviewing the existing literature and key concepts pivotal to this thesis. Topics like EXFOR, outlier detection, and machine learning are discussed to provide a well-rounded understanding of the research context.
- **Methodology:** This chapter is dedicated to outlining the tools, libraries, and theoretical foundations used in the research. It explains the preprocessing and data mining techniques employed, as well as the machine learning methods that are used for outlier detection.
- **Results:** The findings from the application of various methodologies for outlier detection are presented in this chapter. The results are analyzed to assess the reliability, strengths, and weaknesses of each method applied.
- **Conclusions:** This chapter serves as a repository for the key findings, implications, and future research avenues. It draws together the entire scope of the thesis to offer a consolidated view of the research carried out.
- **Social and Professional Responsibility:** This section delves into the ethical, social, and professional implications tied to the study. It highlights how these aspects have been conscientiously considered and addressed in the course of the research.
- **Project Management:** In this chapter, the managerial aspects of the research project are discussed. It covers elements such as planning, resource allocation, time management, and risk assessment.

The thesis is supplemented by two final sections: 'Glossary' and 'Annexes'. The Glossary offers explanations for key terms and concepts, while the Annexes provide additional materials and data sets that support the core research.

2 STATE OF THE ART

In the world of nuclear research, the coming together of real-world data, machine learning, and ways to spot unusual data points is a new and exciting area. At the center of this is the Exchange Format (EXFOR), a detailed collection created to help with worldwide nuclear studies and uses. But there is more to the story, with many methods, partnerships, and new technologies being part of it. In today's data-driven world, machine learning plays a big role. It helps us look at, understand, and use huge amounts of nuclear data in new ways, making research better and faster. This is not just a standalone field but something that links different areas together, allowing for new ideas and insights.

However, handling the data properly is also important. Finding and dealing with unusual or 'outlier' data points makes sure that the information is trustworthy, which means the results we get from it are more reliable. This chapter on the state of the art opens up a broad view of what's happening now, looking at:

- **EXFOR:** What it is, why it is important, and how it is part of global efforts around nuclear data.
- **Machine Learning:** A look at the latest ways it is being used, what it can do, and the challenges it faces in the nuclear field.
- **Outlier Detection:** How we find and deal with unusual data points, the tools we use, and best practices.
- **Additional Sections:** Other connected topics that add to the understanding of nuclear data, machine learning, and keeping the data of high quality.

This chapter serves as a foundational introduction to the vital elements at the intersection of nuclear research, machine learning, and data quality. It offers a starting point for those new to these fields, providing clear explanations of key concepts and terms. It also creates a connection by linking different areas together, showing how they relate to each other and why they matter. Moreover, the chapter provides insight into how these parts fit into the bigger picture of nuclear research, setting the stage for deeper exploration in subsequent chapters. Together, these elements form a comprehensive yet accessible overview, aiming to engage both experts and newcomers. By laying the groundwork, this chapter helps readers understand the core ideas, opening the door to further study and appreciation of a multifaceted and rapidly evolving discipline.

2.1 EXFOR

The EXchange FORmat, or EXFOR, is a comprehensive library of experimental nuclear reaction data, meticulously compiled to facilitate nuclear research and applications. Constructed with a clear goal, it provides a reliable and interpretive data source built around the context of each experiment [8]. The format, designed for flexibility, serves to ease the exchange of data among centers. This platform was developed to facilitate the transmission of nuclear reaction data among the members of the International Network of Nuclear Reaction Data Centres (NRDC), an international collaboration aimed at coordinating the collection, compilation, and global dissemination of nuclear data. The NRDC, overseen by the IAEA Nuclear Data Section, is responsible for maintaining and managing the EXFOR nuclear data library [9].

EXFOR, as a product of worldwide cooperation, offers a system designed for the collection, storage, exchange, and retrieval of experimental nuclear reaction data. The database, as of October 2022, holds 24,382 experiments [10]. It aims to compile a comprehensive collection of low-energy experimental neutron-induced reaction data, a less complete compilation of charged-particle-induced reaction data, and selectively compiles photon-induced, heavy-ion-induced, and high-energy neutron-induced reaction data [8].

The target audience for EXFOR encompasses researchers and professionals in nuclear science and engineering. This includes individuals in academia, national laboratories, and the industry, who are engaged in the design and analysis of nuclear reactors, nuclear fuel cycles, and other nuclear applications.

The organization of data within EXFOR follows a clear, structured format (see Figure 2.1). The basic unit, known as an "ENTRY," corresponds to a single experiment, while divisions within an ENTRY, termed "SUBENTRIES", contain the results of the experiment. The initial SUBENTRY typically lacks numerical data but includes information relevant to the entire ENTRY, such as bibliographic references and details about the experimental setup. The SUBENTRY is further divided into sections [10]:

- **BIB:** This section contains bibliographic, descriptive, and bookkeeping information.
- **COMMON:** This section holds data relevant to all data throughout the SUBENTRY.
- **DATA:** This section contains the experimental data tables.

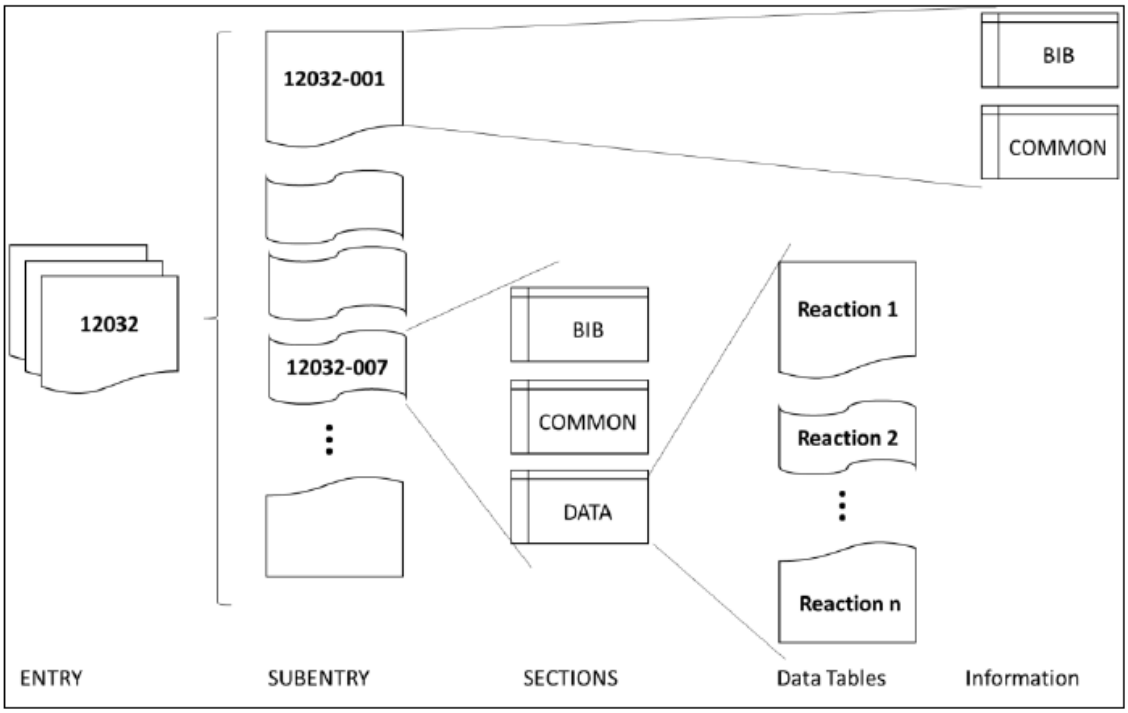

Figure 2.1: Structure of information in EXFOR [10].

Additionally, the EXFOR library incorporates "TRANS" files. These files, which are exchanged among data centers, encompass numerous ENTRIES (work). Each file is formatted as an 80 character ASCII record, wherein columns 1-66 are designated for EXFOR data. For unique identification of a record within the file, columns 67-79 are utilized [10].

EXFOR operates as a supplementary publication medium to conventional, formal publications, ensuring that any revisions made by authors post-publication are reflected in the data files. The numerical data housed within EXFOR are supplemented by explanatory text that delivers essential information about the data's meaning and quality. An EXFOR "entry" may not necessarily correspond to the information found in a single publication. Often, a "work" is reported across multiple formal or informal publications, such as progress reports, conference papers with preliminary results, lab reports, and final articles in refereed international journals. It can also incorporate preliminary data or pre-publication data with the author's consent. Once the final data becomes available, preliminary data are routinely replaced and any new bibliographic references describing the results are added. Importantly, EXFOR is a compilation of the author's original published experimental data and not a collection of recommended values for each reaction. Evaluators systematically review the experimental works and recommend best evaluated cross-section data based on the experimental data from EXFOR. Their results are compiled in various evaluated data files such as ENDF/B, JEFF, JENDL, and more, most of which are also accessible from the IAEA Nuclear Data Section [9].

In the worldwide push to make the EXFOR Nuclear Data Library easier to use, a group known as the Working Party on International Nuclear Data Evaluation Co-operation Subgroup 50 (WPEC SG50), part of the Organisation for Economic Co-operation and Development's (OECD) Nuclear Energy Agency (NEA), has been doing some interesting work. This group is building an advanced database for experimental nuclear reaction data called the Machine-readable Experimental Data User Application & Library (MEDUSAL), as explained in reference [11].

MEDUSAL is an ambitious project that aims to create a large, automatically readable database filled with experimental nuclear reaction data. It has been designed to support a wide array of users, including nuclear data evaluators, experimentalists, model builders, and those engaged in machine learning and artificial intelligence. What sets MEDUSAL apart is its ability to store not only the core data but also additional information contributed by users in areas such as model development, evaluation, and validation.

The architecture of MEDUSAL is thoughtfully planned and segmented into four levels or 'layers.' Layer 0 forms the base where data from EXFOR is translated into MEDUSAL's format, standardizing the units. Layer 1, the main access point for users, consists of information from Layer 0, augmented with machine-readable metadata and supplementary details from EXFOR and related literature. Layer 2 introduces changes to the original authors' data, such as updates to new standards and flagging inconsistent measurements. Layer 3, the most flexible layer, can hold multiple versions of the same dataset, enabling users to apply their corrections and judgments. Importantly, this layer documents precisely how a dataset was employed in an evaluation, essential for ensuring reproducibility [11].

Creating MEDUSAL has not been without its challenges, particularly in assuring the accuracy and currency of the data. However, the Lewis-led team is optimistic about MEDUSAL's future applications in nuclear data evaluations, model development, and validation. They envision expanding MEDUSAL to encompass more data types, such as covariance data, and are developing an Application Programming Interface (API) for easier database access. The potential use of MEDUSAL in machine learning and artificial intelligence amplifies the need for precise and reliable data [11].

In conjunction with the MEDUSAL project, there are ongoing developments in Python codes to parse the EXFOR nuclear data library. These codes, publicly available on the IAEA-NDS GitHub Repository [12], include tools to read the entire EXFOR data into nested dictionary structures for Python handling or .json format conversion. Since these projects are still under development and many features are pending, users should approach them with caution. A different approach, known as "EXFORTABLES", developed by Arjan Koning, will be discussed in more detail in the following subsection.

2.2 EXFORTABLES

EXFORTABLES is an innovative software tool developed by Arjan Koning and designed to translate the extensive and intricate EXFOR experimental nuclear reaction database into a more accessible and user-friendly format. This transformation enhances the ease of accessing, plotting, and utilizing nuclear data for evaluation purposes. Essentially, EXFORTABLES serves as a bridge for researchers to interact with the wealth of data within EXFOR more efficiently.

The capabilities of EXFORTABLES extend beyond mere data translation. The software's directory structure enables users to quickly locate the required data, saving valuable time in research. Moreover, it automatically compares the data against global nuclear model calculations, identifying potential errors in the database. This dual functionality simultaneously tests nuclear models, allowing researchers to assess their accuracy and deepen their comprehension of nuclear reactions.

The open-source nature of EXFORTABLES presents an added benefit. Researchers can modify and adapt the software according to their specific needs, promoting a tailored research experience. It's noteworthy that EXFORTABLES is assumed to be installed on a Unix/Linux operating system, and the installation process requires approximately 3 GB of free disk space.

Upon installation, the EXFORTABLES package will be stored in the `exfortables/` directory. It consists of various directories and files, each serving a distinct purpose. These include:

- `source/`: containing the Fortran code responsible for database creation and data checking.
- `files/`: housing the entire EXFOR database in different formats, alongside other related information.
- Specific directories like `'n/'`, `'p/'`, `'d/'`, etc., representing different reaction types, and containing the entire structured database.
- `doc/`: with documentation.
- `input/`: for the input file used to create the database.
- `quality`: for quality scores.
- `stat/`: with statistical information.
- `special/`: for specific data files.
- `README`: outlining the package content.

The hierarchical directory structure of EXFORTABLES is categorized by projectile type (such as neutron, proton, deuteron, etc.), element, mass, and reaction. Most users can directly utilize the database without additional installation or configuration, offering a seamless research experience.

While EXFORTABLES offers comparison with evaluated nuclear data libraries using statistical methods, it is worth acknowledging this aspect even though it falls outside the scope of this thesis, focused more on Machine Learning techniques. The underlying methodology in EXFORTABLES could also be insightful for the detection or comparison of outlier results, thus contributing to the broader field of nuclear data analysis.

2.3 Evaluated Nuclear Data Libraries

In the domain of nuclear science and technology, Evaluated Nuclear Data Libraries (ENDL) play a critical role. These libraries consist of systematic collections of evaluated nuclear reaction data, where 'evaluated' refers to data that has been critically assessed and validated through rigorous analysis of experimental data, theoretical models, and phenomenological trends. ENDLs are essentially a bridge between raw experimental data and practical application in nuclear technology [13].

The primary purpose of ENDLs is to serve as reliable sources of nuclear data for various applications in nuclear science and technology, which include but are not limited to nuclear energy production, medical applications, nuclear safeguards, and nuclear astrophysics. The quality and reliability of these libraries are paramount, as they directly impact the safety, efficiency, and effectiveness of these applications. In the process of obtaining the evaluated data, experimental databases such as EXFOR play an integral part. EXFOR, with its extensive collection of nuclear reaction data from a wide range of experiments conducted worldwide, serves as a primary input in the evaluation process. The raw data from EXFOR is analyzed and validated through a meticulous process of evaluation, involving critical review, data adjustment, model calculations, and uncertainty quantification. The resulting evaluated data is then compiled into the ENDLs.

To illustrate the global efforts in maintaining and developing evaluated nuclear data libraries, a range of key libraries have been listed below in Table 2.1. Each library is managed by a specific organisation or institution, demonstrating the cooperative nature of this significant scientific endeavor [13]. The libraries span across multiple continents, from the United States to Europe and Asia, each contributing a unique set of expertise and focus.

Table 2.1: Major Evaluated Nuclear Data Libraries and their respective Organisations [13]

Evaluated Nuclear Data Library	**Organisation**
ENDF/B-VIII (Evaluated Nuclear Data File/B Version VIII)	The National Nuclear Data Center (NNDC), Brookhaven National Laboratory, United States
JEFF (Joint Evaluated Fission and Fusion)	The Organisation for Economic Co-operation and Development (OECD) Nuclear Energy Agency (NEA) Data Bank, France
JENDL (Japanese Evaluated Nuclear Data Library)	Japan Atomic Energy Agency (JAEA)
CENDL (China Evaluated Nuclear Data Library)	China Institute of Atomic Energy (CIAE)
BROND (Russian Evaluated Nuclear Data Library)	National Nuclear Data Center of Russia, Institute for Physics and Power Engineering (IPPE), Obninsk, Russia
TENDL (TALYS Evaluated Nuclear Data Library)	Nuclear Research and Consultancy Group (NRG), Netherlands

2.4 Machine Learning

Machine Learning (ML) is a subfield of artificial intelligence (AI) that focuses on the design of systems, enabling them to learn and make decisions or predictions based on data. Rather than manually programming specific tasks, these systems improve and adapt their performance over time with exposure to new data. The roots of machine learning date back to the mid-20th century, with the earliest concepts revolving around the idea of computers or machines that can simulate aspects of human intelligence. However, the term "machine learning" was officially coined by Arthur Samuel in 1959 [14].

Machine learning is used extensively in various fields including healthcare, finance, transportation, and energy. In healthcare, machine learning algorithms can analyze large datasets to predict disease outcomes, thereby enhancing disease diagnosis and treatment [15]. In finance, machine learning aids in detecting fraudulent transactions [16]. In transportation, machine learning algorithms underpin the technologies in autonomous vehicles [17]. Similarly, in the energy sector, machine learning models can forecast energy demand and optimize energy distribution [18].

Machine learning algorithms can handle vast, complex datasets, often identifying hidden patterns that may not be immediately noticeable to human analysts. This attribute has been central to its broad application in today's increasingly data-intensive world. However, the methodologies and approaches to machine learning can vary, with three principal types often recognized: supervised learning, unsupervised learning, and reinforcement learning [19].

- **Supervised Learning:** is a type of machine learning where the model is trained on labeled data. It learns a mapping from features to labels and applies this to accurately predict the labels for new, unseen instances.A commonly cited example of supervised learning is predicting house prices based on features such as location, size, and condition of the property. The model would be trained on historical data where the prices of sold houses (the labels) are known, and then it could predict the price of a new house on the market (an unseen instance) based on its features.
- **Unsupervised Learning:** involves training the model using unlabeled data, and the model identifies patterns and structures within this data on its own. An example of this can be seen in customer segmentation based on shopping habits, where the company does not initially know what these segments should be, so unsupervised learning is used to identify them.
- **Reinforcement Learning:** is a type of machine learning where an agent learns to make decisions by performing actions and observing the results, essentially learning through trial and error. Google's AlphaGo, which learned to play the board game Go by playing millions of games against itself, is a notable example of reinforcement learning.

While the three types of machine learning may seem distinct, they are united by a common goal: to learn from data and make intelligent decisions or predictions. The principal difference lies in the type of feedback they are provided and the learning context, which can range from having a clear mapping of inputs to outputs (supervised learning), no specific output targets but an aim to understand data structure (unsupervised learning), or learning from a series of actions and rewards in a dynamic context (reinforcement learning) [1]. Figure 2.2 shows a diagram depicting the taxonomy of Machine Learning inside AI.

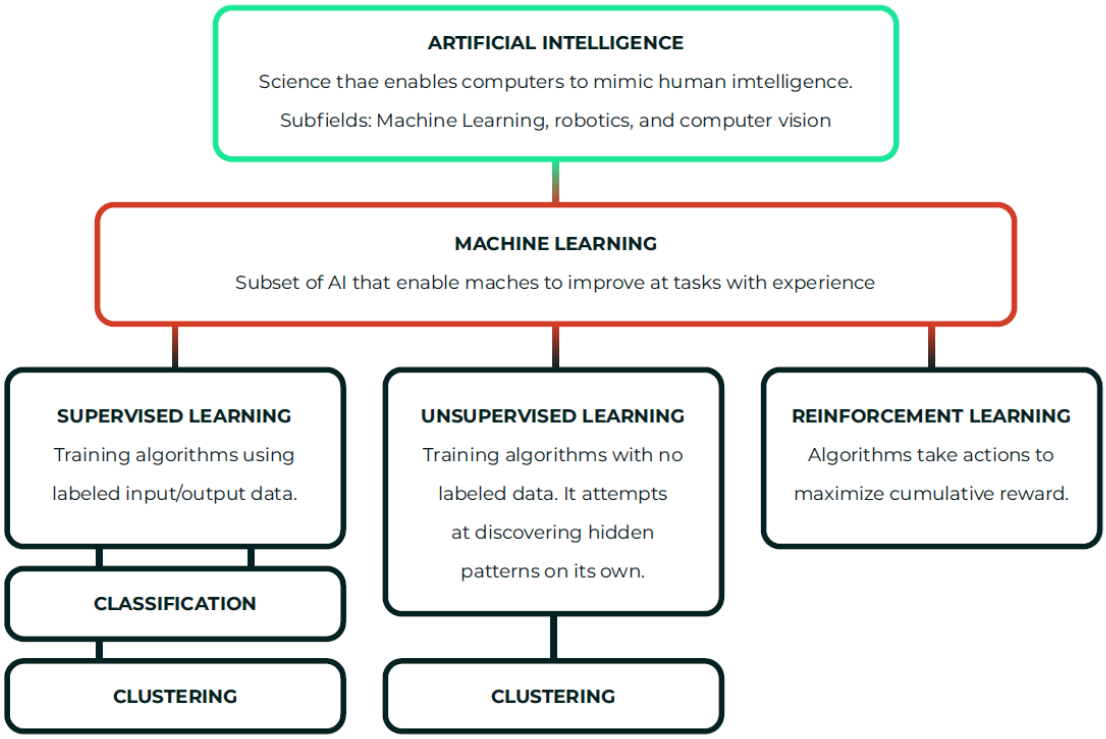

Figure 2.2: Artificial Intelligence and Machine Learning taxonomy [1].

Following this, it's essential to recognize the relationships between machine learning, deep learning, and artificial intelligence, since these are terms that are typically confused or used in an imprecise manner [1].

- **Artificial Intelligence:** is the broadest term, encompassing any technique that enables machines to mimic human intelligence, using logic, if-then rules, decision trees, and machine learning, including deep learning. For instance, chess-playing computers that examine millions of potential outcomes are using a form of AI.
- **Machine Learning:** as explained, is a subset of AI that involves the use of statistical techniques to enable machines to improve at tasks with experience. An example is email filtering systems. They learn to classify emails as 'spam' or 'not spam' based on previous examples that they have seen.
- **Deep Learning:** is a further subset of Machine Learning, inspired by the structure and function of the human brain—namely neural networks. Deep Learning (DL) models learn to perform classification tasks directly from images, text, or sound using large neural networks and a significant amount of data. For example, Deep Learning is used in driverless car technologies to identify objects, such as pedestrians and other vehicles.

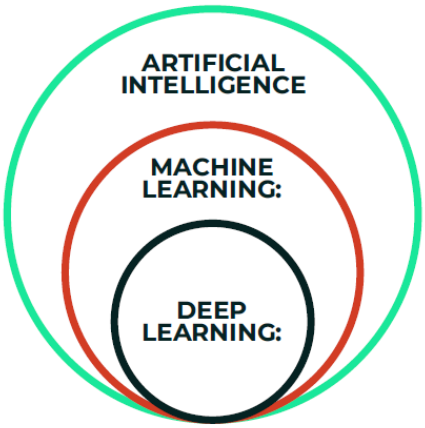

Figure 2.3: Nested structure of AI [1].

While these definitions provide a basic understanding, it's worth noting that the boundaries and interactions between AI, ML, and DL are complex and continually evolving, driven by innovations in theory and increasing computational power. In addition to the aforementioned, machine learning is generally more interpretable and easier to understand than deep learning. It allows for feature importance analysis and provides insights into the data and the underlying processes generating it. On the other hand, deep learning models, especially larger ones, often act as black boxes with little interpretability. However, they often excel at handling unstructured data and can achieve impressive performance on tasks such as image or speech recognition [1].

Python has emerged as one of the most popular programming languages for machine learning, owing to its simple syntax, extensive collection of libraries, and compatibility with a wide range of machine learning frameworks. Libraries such as scikit-learn for general machine learning tasks, TensorFlow for deep learning, and Pandas for data manipulation and analysis have made Python a preferred choice for both beginners and experts in the field. Moreover, the active online community and abundance of educational resources further contribute to its popularity, added to the fact that it is a great option for end-to-end machine learning applications due to its compatibility with diverse data sources and tools for data visualization. As a result, it acts as a complete tool for effectively developing, validating, and deploying machine learning models [1].

It is important to note that Python's role and usage in machine learning will be explored in more detail in Chapter 3 of this thesis. There, we will delve deeper into the specific libraries and techniques employed in the project, providing a better understanding of the practical application of Python in machine learning for enhancing the EXFOR Nuclear Data Library.

2.5 Outliers

Outliers are observations that fall significantly outside the norm within a dataset, typically characterized by their distance from other values in the data distribution [20]. In data analysis, these are critical to consider as they can significantly skew results, leading to inaccurate interpretations and predictions [21].

Outliers can arise for a variety of reasons, such as measurement errors, data entry errors, or genuine extreme observations. Regardless of their source, their importance in data analysis cannot be overstated. Outliers can have a significant impact on our understanding of the data: they can either provide valuable insights into anomalies and trends, or, if they are errors, they can distort the underlying structure and relationships in the data [21]. Thus, outlier detection and management form a critical step in any data analysis pipeline.

Outlier detection techniques can be categorized broadly into statistical methods and machine learning methods [22].

2.5.1 Statistical Techniques

Statistical techniques for outlier detection primarily focus on the identification of data points that deviate from a distribution's expected characteristics. These methods often assume specific statistical properties of the underlying data and are typically best suited for univariate outlier detection [23]. Some of the widely used statistical techniques include:

1. **Z-Score or Standard Score:** This method calculates the Z-score of each data point, which reflects how many standard deviations it is from the mean. Data points with a Z-score above a certain threshold (commonly 3 or -3 for a normal distribution) are considered outliers [24].
2. **Modified Z-Score Method:** This approach, based on the median absolute deviation (MAD), is more robust to outliers than the traditional Z-score method, making it suitable for datasets with heavy-tailed distributions [25].
3. **IQR Method:** This method utilizes the interquartile range (IQR), which is the range between the first quartile (25th percentile) and the third quartile (75th percentile). Any data point that falls below the first quartile minus the IQR times a factor (often 1.5) or above the third quartile plus the IQR times a factor is considered an outlier [26].

2.5.2 Machine Learning Techniques

In contrast to traditional statistical methods, machine learning techniques can handle multivariate outliers and do not require any assumptions about the distribution of the data [27]. Some popular machine learning techniques for outlier detection include:

1. **Clustering Algorithms:** Techniques such as K-Means or DBSCAN create groups (clusters) of data points based on their similarity. Data points that do not belong to any cluster, or belong to small, sparsely populated clusters, can be considered outliers [28].

2. **Anomaly Detection Algorithms:** Algorithms such as One-class SVM or Isolation Forest are specifically designed for outlier detection. They work by learning the "normal" data's distribution and then identifying points that deviate from this norm [29].

3. **Neural Networks:** Methods such as autoencoders can be used for outlier detection. These networks are trained to reconstruct the input data, and those instances that the network has the most difficulty reconstructing are considered outliers [30].

Overall, the choice of method depends on the specific characteristics of the dataset and the task at hand. It is important to note that not all outliers are "bad" or undesirable. In some cases, outliers can represent valuable information that is worth further investigation. However, if these outliers are the result of errors or noise, it is crucial to detect and, if necessary, remove them to prevent the model from learning from these misleading instances.

3 METHODOLOGY

The Methodology chapter serves as a comprehensive guide to the procedures and tools employed in this research, detailing the process from data collection to analysis. It starts with an explanation of the programming language Python, and the specialized libraries and tools used, such as NumPy, Pandas, Scikit-learn, TensorFlow, Keras, and Matplotlib. These tools play a critical role in handling numerical computations, data manipulation, machine learning algorithms, deep learning tasks, and data visualization, essential elements of this study.

Following the discussion of tools, we address the data collection and the general approach to preprocessing the data, ensuring that it is suitable for analysis. This preprocessing is a significant step and will be explored further in chapter 4. In conjunction with preprocessing, we introduce the methods employed for outlier detection, which include traditional statistical techniques and machine learning approaches, as well as the use of a neural network model for data prediction.

Lastly, we outline the criteria by which the success and performance of the models were evaluated, focusing on the specific metrics for both outlier detection and neural networks. This chapter strives to present the methodology in a clear and understandable manner, laying out the groundwork for the work done in this research. By detailing the methods in a straightforward way, we provide a solid foundation for the implementation and evaluation stages that will be addressed in the Results chapter. The subsequent sections of this chapter will delve into these aspects in greater detail, providing readers with a clear understanding of the technical aspects of this research project.

3.1 Tools and Libraries Used

In the context of this research, Python serves as our primary language for implementing machine learning models, leveraging its powerful suite of libraries. Our methodology focuses on several key Python libraries and frameworks that are widely used in the field of machine learning:

- **NumPy:** The foundational library for numerical computing in Python, NumPy (Numerical Python), provides support for arrays, matrices, and high-level mathematical functions to operate on these. NumPy is indispensable for handling numerical data and performing mathematical operations, which are fundamental in machine learning.
- **Pandas:** This library is used for data manipulation and analysis. It provides data structures and functions needed to manipulate structured data. Features like handling missing data, merging, reshaping, selecting, replacing, data filtration, and others make it perfect for preprocessing the data for machine learning tasks.
- **Scikit-learn:** This is one of the most widely used machine learning libraries in Python. It provides a range of supervised and unsupervised learning algorithms. Scikit-learn comes with standard datasets, various machine learning algorithms (from linear regression to clustering), and also offers tools for model fitting, data preprocessing, model selection, and evaluation.
- **TensorFlow:** Originally developed by Google Brain, TensorFlow is an open-source library for numerical computation and large-scale machine learning. TensorFlow bundles together a slew of machine learning and deep learning (aka neural networking) models and algorithms and makes them useful by way of a common metaphor. It's known for its capabilities in handling deep learning tasks.

- **Keras:** Is a high-level neural networks API, capable of running on top of TensorFlow, and is used for building and training deep learning models. It's user-friendly and easy to use, making it a popular choice for beginners and experts in deep learning.
- **Matplotlib:** Data visualization is a crucial part of machine learning. Matplotlib is a popular library for creating static, animated, and interactive visualizations in Python, aiding in the analysis and interpretation of the computational results.

In terms of workflow, we first prepare the data using Pandas and NumPy, carry out preprocessing tasks, and then apply appropriate machine learning algorithms from Scikit-learn, TensorFlow, or Keras as required. The data and the performance of the models are visualized using Matplotlib.

3.2 Preprocessing and Data Mining

The preprocessing stage represents a vital phase in our methodology, dedicated to preparing the raw data for subsequent analysis. This complex and multifaceted process involves numerous tasks such as cleaning and transforming the data, selecting appropriate variables, and engineering the features that will be used in the models. These efforts set the stage for both the outlier detection and neural network modeling that follow. In this section, we will delve into the specifics of this comprehensive preprocessing effort, detailing the techniques and approaches applied to ensure that the data is in the optimal format for the analysis techniques utilized in subsequent chapters.

Preprocessing and data mining are essential components in machine learning applications. The success of algorithms relies heavily on the structure and quality of the dataset. The EXFOR nuclear data library, comprising more than 24,000 experiments, each with unique characteristics, measurements, and features, presents a significant challenge for several reasons.

First, the sheer volume of data in EXFOR is immense, demanding careful registration and management. Second, the library contains a heterogeneous mix of data, reflecting the wide array of target elements, incident particles, reactions, and quantities measured. Third, the lack of a consistent structure for storing data further complicates matters. Although there is a general structure followed within EXFOR, as mentioned in section 2.1, the earliest experiments from the 1950s and 60s lacked digitalization and common structure. When these experiments were digitized, some information might have been lost. Even in more modern experiments, not all details might be shared, resulting in a heterogeneous and complex library.

Efforts like MEDUSAL and EXFORTABLES, also referenced in 2.1, have been made over the past decade to mitigate these challenges.

EXFORTABLES, in particular, serves as a valuable starting point. It parses the entire EXFOR library, creating a logically structured directory on your computer, categorized by projectile, element, mass, and reaction. Moreover, it formats the information in a more computationally friendly way. Figure 3.1 illustrates an example of a file located at:

`exfortables/p/Ca044/xs/004/p-Ca044-MT004-Krajewski-O2136002.2013`

As seen, the experiment-related information is neatly structured, beginning with a '#', while the data points are presented in table format without the hashtag. This approach makes the data more accessible to computer code, such as Python, compared to the original ENDF format used in EXFOR.

```
# Target Z    :  20
# Target A    :  44
# Target state:
# Projectile  : p
# Reaction    : (p,n')
# Final state :
# Quantity    : Cross section
# Frame       : L
# MF          :    3
# MT          :    4
# X4 ID       : O2136002
# X4 code     : 20-CA-44(P,N)21-SC-44,,SIG
# Author      : Krajewski
# Year        : 2013
# Data points :      8
#    E(MeV)       xs(mb)      dxs(mb)     dE(MeV)
  5.53000E+00  8.42000E+00  0.00000E+00  0.00000E+00
  7.28000E+00  1.44530E+02  0.00000E+00  0.00000E+00
  7.86000E+00  2.77330E+02  0.00000E+00  0.00000E+00
  8.84000E+00  6.95480E+02  0.00000E+00  0.00000E+00
  9.83000E+00  5.64270E+02  0.00000E+00  0.00000E+00
  1.06300E+01  7.28140E+02  0.00000E+00  0.00000E+00
  1.31400E+01  6.33630E+02  0.00000E+00  0.00000E+00
  1.74800E+01  4.13510E+02  0.00000E+00  0.00000E+00
# Reference:
#S.Krajewski, I.Cydzik, K.Abbas, A.Bulgheroni, F.Simonelli, U.Holzwarth, A.Bilewicz
#Cyclotron production of 44Sc for clinical application.
#Jour. Radiochimica Acta Vol.101, p.333, 2013
```

Figure 3.1: Example entry from EXFORTABLES.

Being able to access information in a well-structured way is valuable but only the first step in the path to obtain usable data for machine learning. It becomes essential at this point to define clearly what type of information we need to apply machine learning. This involves understanding our objective and how the information should be ideally structured to use it efficiently.

Before moving on to machine learning analysis, an essential preprocessing step is the scaling of feature values. Given that most machine learning algorithms use distance-based metrics, the scale of each feature can affect the model's performance. For instance, a feature with a large range could dominate the objective function and make the estimator unable to learn from other features correctly. In this study, we utilize the StandardScaler provided by the Scikit-learn library to standardize the dataset's features by removing the mean and scaling to unit variance. The formula for standard scaling is:

$$z = \frac{(x - u)}{s} \tag{3.1}$$

Where x is the original feature vector, u is the mean of that feature vector, and s is its standard deviation. This scaling ensures that each feature contributes equally to the distance computations during machine learning model training.

In general terms, when it comes to machine learning and Python, the preferred way to structure your dataset is in a dataframe. A dataframe in Python is a two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes (rows and columns). It's like a

spreadsheet or SQL table and is generally understood to be the most common Pandas object, allowing for the easy manipulation, analysis, and visualization of data.

But structuring the data in a dataframe is not the only requirement. Most machine learning algorithms rely on numerical values, meaning that categorical values may need to be converted into numerical ones. There are different methods to achieve this, but a common one is the One-hot encoding technique. One-hot encoding is a process that transforms categorical data variables so they can be provided to machine learning algorithms to improve predictions. It involves converting each categorical value into a new categorical column and assigns a binary value of 1 or 0. Each integer is represented as a binary vector, with all the values zero except the index, marked with a 1.

At the end of this process, we should have a dataframe, akin to a table, where each row represents a data point from an experiment, and each column corresponds to a feature that helps identify and characterize it. All the data points for the same experiment will have identical values in the columns that characterize the experiment they belong to. And the categorical values are transformed through the one-hot encoding into columns where the values are 0s or 1s, depending on whether that column represents the data point.

So first, all experiments parsed with EXFORTABLES must be read from the files and stored in a way that makes them easily accessible at any point. For this, it was decided to use Python's object-oriented coding, so a class named Experiment was defined with different attributes representing all the characteristics of the experiment. The utilization of object-oriented coding in this context offers several advantages:

1. **Encapsulation**: By bundling the data with the methods that operate on it, the implementation details are hidden from the user, ensuring consistency.
2. **Modularity**: Code is organized into objects, making it easier to maintain, modify, and understand.
3. **Reusability**: Objects and classes can be reused across different parts of the project or in different projects, promoting code efficiency.
4. **Inheritance**: It enables the creation of new classes that are built upon existing classes allowing code reuse and the addition of new features.

This approach also allows us to create different methods, such as a function for automatically plotting the results of an experiment.

Once the class is defined, we can code a function to read each experiment, parsing the information line by line from EXFORTABLES, as shown in Figure 3.1. Although saving each experiment in a Python list is not the most practical way to handle the data, our goal is to read all the experiments we are interested in (in our case, those using protons as incident particles) and save them in a file. This file can be easily read and transferred, allowing quick and straightforward access to the list of experiments without requiring each user to have the entire EXFORTABLES files on their computer.

With the data stored in this manner, we have a valuable format that is suitable for various applications such as data analysis and information retrieval. By scripting the experiments through objects and storing the information in a list, we can create functions to perform different tasks, such as:

- Getting all unique values of each attribute.
- Filtering experiments based on the value of a specific attribute (example: query all experiments in which MT=102).
- Classifying experiments by attributes (example: group experiments by MT).
- Plotting one or more experiments at once.

Indeed, one of these tasks, classifying, is crucial for the next step. While it would be feasible to create a dataframe with all the data points for every proton-induced experiment, it doesn't necessarily align with our analysis objectives. Since our main goal is to detect outliers, grouping data based on the value being measured seems more practical. Fortunately, uncertainties in the measured value or energy used are already treated, ensuring consistent column appearances. This uniformity simplifies comparison and enhances classification.

Lastly, the data within each group must be converted into a dataframe, following the specifications mentioned earlier. By meticulously organizing and preparing the data in this manner, we set the stage for accurate and efficient machine learning analysis, aligning with our primary focus on outlier detection.

To sum up this section, we could divide the preprocessing and data mining into the following key points:

1. **Utilizing EXFORTABLES**: Initial parsing of the EXFOR nuclear data library using EXFORTABLES to structure it in a computationally friendly manner.
2. **Defining Objectives**: Clarifying the information requirements and the ideal data structure for machine learning tasks.
3. **Feature Scaling**: Utilizing StandardScaler to standardize the features, thereby enhancing the algorithm's ability to learn effectively from all features.
4. **DataFrames as Data Structure**: Converting the cleaned and parsed data into Python dataframes for easy manipulation and analysis.
5. **Categorical to Numerical**: Applying One-hot encoding to transform categorical values into a format suitable for machine learning algorithms.
6. **Object-Oriented Design**: Creating a Python class named `Experiment` to encapsulate the attributes and methods associated with individual experiments.
7. **Data Reading and Storage**: Developing a function to read experiments one by one from EXFORTABLES and storing them in an easily accessible file.
8. **Data Manipulation Functions**: Implementing functionalities to filter, classify, and plot experiments based on various attributes.
9. **Grouping for Analysis**: Classifying experiments into groups based on the value being measured to facilitate outlier detection.
10. **Final DataFrame Creation**: Converting the grouped data into a final dataframe, ready for machine learning applications.

By following these steps, the data is prepared in a structured manner that is not only conducive to machine learning but also efficient for specific tasks such as outlier detection. The application and results of these preprocessing and data mining steps will be further explored in the chapter 4.

3.3 Interquartile Range (IQR)

The Interquartile Range (IQR) is a statistical measure widely used for identifying potential outliers in a dataset. Originating from the field of descriptive statistics, it provides a straightforward yet robust way to understand data variability around the median. Unlike methods that use mean and standard deviation, the IQR focuses on the middle 50% of data points, making it less sensitive to extreme values or outliers.

The IQR is calculated as the difference between the third quartile ($Q3$) and the first quartile ($Q1$) of the dataset. These quartiles divide a rank-ordered data set into four equal parts. Specifically, $Q1$ is the median of the lower half, not including the overall median if the number of elements in the dataset is odd, and $Q3$ is the median of the upper half. Mathematically, the IQR is represented as:

$$\text{IQR} = Q3 - Q1 \tag{3.2}$$

To identify outliers using the IQR, the following steps are generally followed:

- Calculate $Q1$ and $Q3$ for the dataset.
- Compute the IQR using Equation 3.2.
- Define the inner fences:

$$\text{Lower Inner Fence} = Q1 - 1.5 \times \text{IQR} \tag{3.3}$$

$$\text{Upper Inner Fence} = Q3 + 1.5 \times \text{IQR} \tag{3.4}$$

- Data points outside these inner fences are considered as outliers.

A graphical representation of how the IQR method works is shown in Figure 3.2 below.

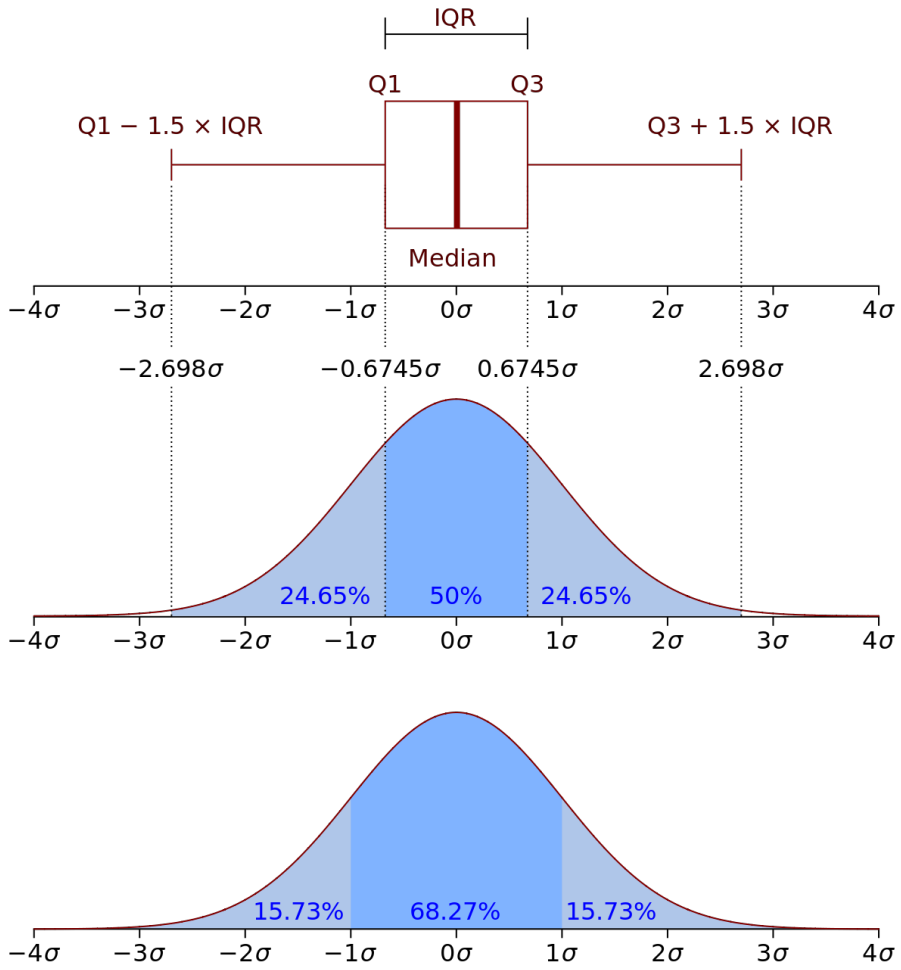

Figure 3.2: Graphical representation of the IQR method [31].

While the IQR method's simplicity and robustness are appealing, it has its limitations. For instance, the IQR may be less sensitive to outliers that are close to the inner fences. Moreover, the risk of data loss is another concern, as some genuine data points may be labeled as outliers, especially if the multiplier for calculating the inner fences is not appropriately chosen.

Additionally, the effectiveness of the IQR method is influenced by the distribution of the data points in the dataset. For skewed or heavy-tailed distributions, the IQR might not effectively identify outliers, making it crucial to select the multiplier for the inner fences carefully.

3.4 Isolation Forest

Isolation Forest is an ensemble-based machine learning algorithm for anomaly detection. Unlike traditional methods which seek to construct a profile of normal instances, Isolation Forest isolates anomalies by exploiting the inherent properties of tree-based partitioning. The basic idea is to recursively partition the feature space by randomly selecting a feature and a random value between the feature's minimum and maximum values.

The Isolation Forest algorithm has the following key steps:

- Randomly sample the data and construct an isolation tree by recursively partitioning the

feature space.

- Continue this process until a predefined number of trees are generated, forming the Isolation Forest.

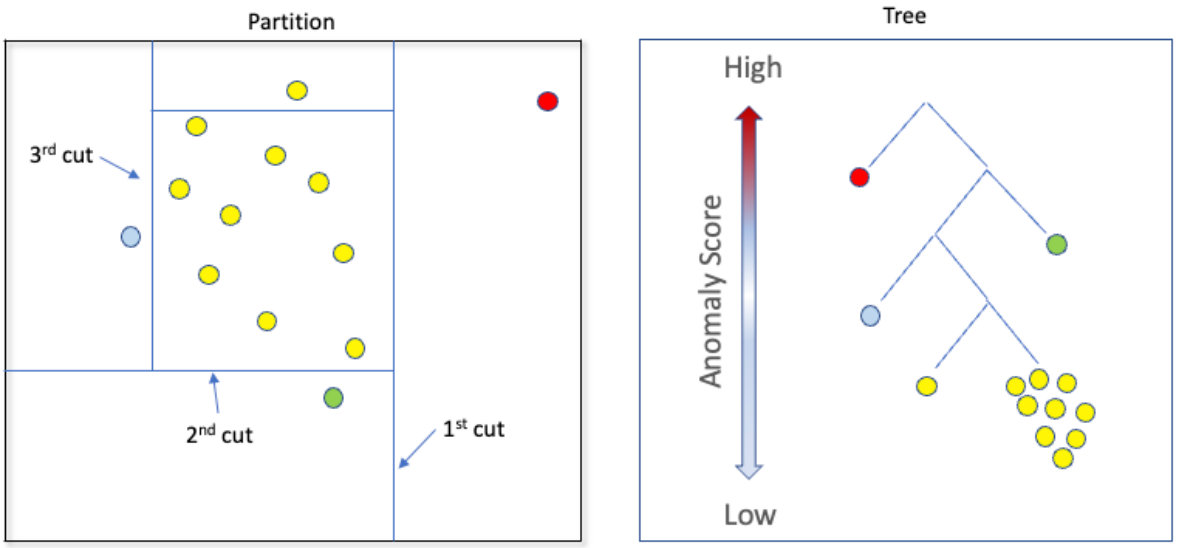

Figure 3.3: Graphical representation of the Isolation Forest method [32].

The anomaly score is calculated using the equation shown in Equation 3.5:

$$s(x,n) = 2^{-\frac{E(h(x))}{c(n)}} \tag{3.5}$$

Where $s(x,n)$ is the anomaly score of sample x, $E(h(x))$ is the average path length of x, n is the number of external nodes, and $c(n)$ is the average path length of unsuccessful search in a binary search tree.

Isolation Forest excels in its computational efficiency and its ability to handle high-dimensional datasets. It also has the advantage of not making any assumptions about the underlying distribution of the data. However, Isolation Forest may have difficulties in distinguishing between noise and actual anomalies, especially in datasets with strong overlap between the two categories. Moreover, because it uses random partitioning, the results are not always deterministic, which may be a downside in certain applications.

The distribution of points in the dataset can influence the performance of Isolation Forest. In dense clusters where anomalies are close to normal instances, the algorithm may fail to distinguish them adequately. Conversely, in well-separated data spaces, the algorithm performs remarkably well.

By understanding and implementing the Isolation Forest algorithm following these guidelines, one can effectively detect outliers in complex datasets where traditional statistical methods may fall short.

3.5 DBSCAN

DBSCAN is a density-based clustering algorithm that identifies clusters in a dataset and treats points that do not belong to any cluster as outliers. Unlike partitioning methods like K-means, DBSCAN does not require the number of clusters to be specified in advance. It works by defining a neighborhood around each data point and finding clusters based on the density of these neighborhoods.

The DBSCAN algorithm consists of the following key steps:

1. Choose two parameters: Eps, the radius of the neighborhood, and $MinPts$, the minimum number of points required to form a dense region.
2. For each unvisited point, identify its Eps-neighborhood. If it contains at least $MinPts$, a cluster is formed.
3. Expand the cluster by including points within the Eps-neighborhood of each point in the cluster.
4. Iterate over each unvisited point, repeating steps 2 and 3 until all points have been visited.

To formalize the concept of density and clustering, DBSCAN utilizes the following equation to define a neighborhood:

$$N_{Eps}(p) = \{q \in D : \text{dist}(p, q) \leq Eps\} \tag{3.6}$$

Where $N_{Eps}(p)$ is the Eps-neighborhood of point p, D is the dataset, and $\text{dist}(p, q)$ is a distance metric (usually Euclidean distance).

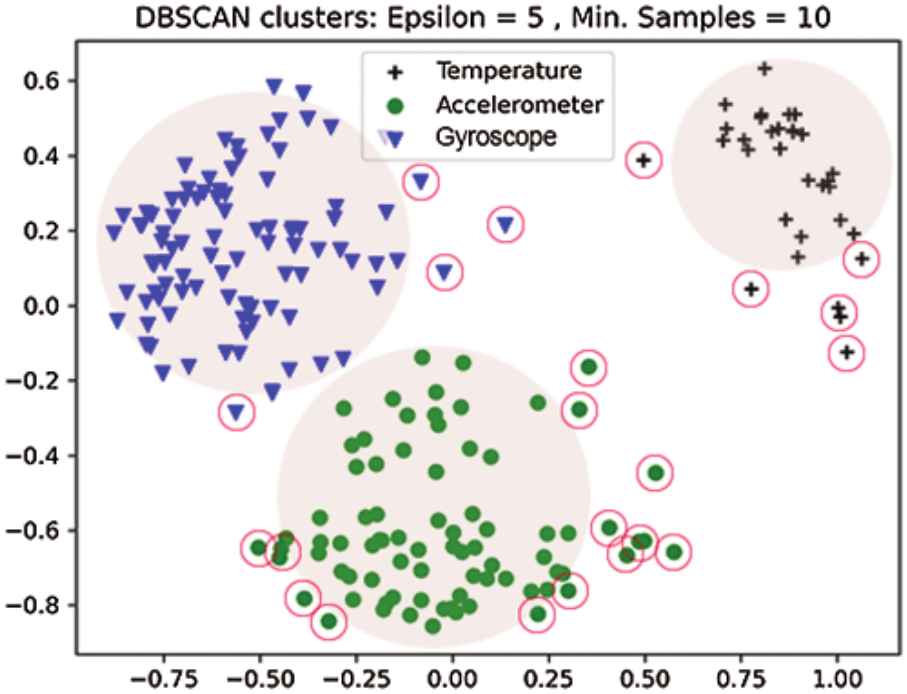

Figure 3.4: Graphical representation of the DBSCAN method [33].

DBSCAN has several strengths and weaknesses. One of its primary advantages is that it can find arbitrarily shaped clusters, unlike spherical clustering methods like K-means. Moreover, it inherently handles outliers by classifying them as points that do not belong to any cluster. On the downside, DBSCAN is sensitive to the choice of Eps and $MinPts$, and its performance can degrade with high-dimensional data. The algorithm can also struggle with clusters of varying densities.

The effectiveness of DBSCAN in outlier detection is directly influenced by the distribution of data points in the dataset. If points are uniformly distributed across various densities, DBSCAN may find it challenging to identify the boundaries between clusters and outliers accurately. However, in datasets where the differences in density between normal instances and outliers are distinct, DBSCAN can be incredibly effective.

3.6 Local Outlier Factor (LOF)

The Local Outlier Factor (LOF) algorithm is a method used for outlier detection that focuses on the local properties of data points rather than global characteristics. Unlike traditional methods that define outliers based on some global property of the dataset, LOF considers each data point within the context of its neighbors.

The core concept in LOF is 'locality', which is evaluated based on the density deviation of a data point in relation to its neighbors. The algorithm uses several steps to assess the local outlier-ness of each data point. First, for each data point, it calculates the distance to its k^{th} nearest neighbor, commonly referred to as *k-distance*. Mathematically, this is represented as:

$$\text{k-distance}(A) = \min_{B \in N_k(A)} d(A, B) \tag{3.7}$$

Second, the algorithm computes the Reachability Distance between a data point and its neighbors. This distance is defined as the maximum of the k-distance of the neighbor and the actual distance to the neighbor, formulated as:

$$\text{ReachabilityDistance}_k(A, B) = \max\{\text{k-distance}(B), d(A, B)\} \tag{3.8}$$

Subsequently, the Local Reachability Density (LRD) of each data point is calculated. LRD is the inverse of the average reachability distance of the k neighbors of the data point:

$$\text{LRD}_k(A) = \left(\frac{\sum_{B \in N_k(A)} \text{ReachabilityDistance}_k(A, B)}{|N_k(A)|} \right)^{-1} \tag{3.9}$$

Finally, the LOF score of each data point is computed. It is the average ratio of the LRDs between the data point and its k neighbors:

$$\text{LOF}_k(A) = \frac{\sum_{B \in N_k(A)} \frac{\text{LRD}_k(B)}{\text{LRD}_k(A)}}{|N_k(A)|} \tag{3.10}$$

LOF identifies outliers by scrutinizing the LOF score (3.10) of each data point. Specifically, a data point with a LOF score significantly greater than 1 is considered an outlier, as it is less dense than its neighbors.

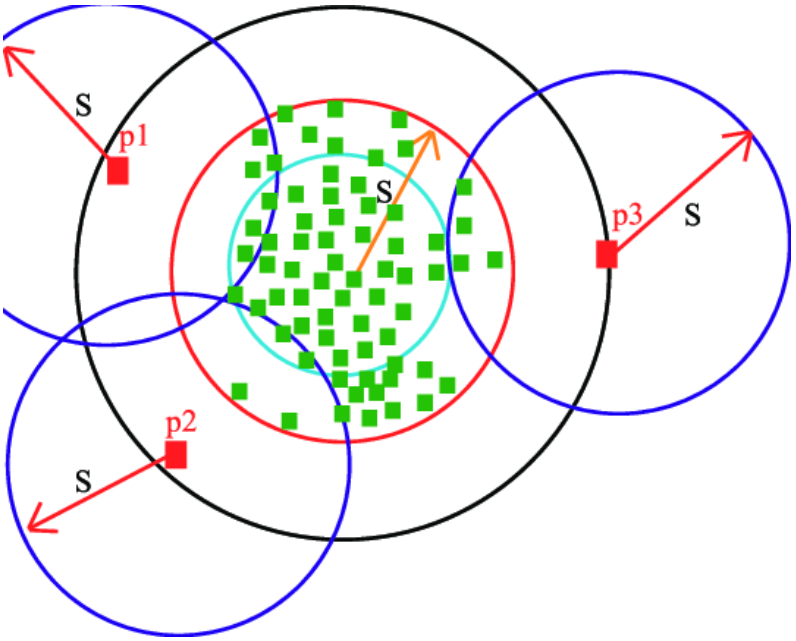

Figure 3.5: Graphical representation of the LOF method [34].

LOF has distinct advantages and disadvantages. One of its strengths is its adaptability to varying densities within a dataset, providing a nuanced method for outlier detection. It does not require prior knowledge about the number of outliers, which is an advantage for dynamically changing data. On the downside, the algorithm is computationally expensive, especially for high-dimensional datasets. The choice of the parameter k can also greatly impact the results. Most pertinent to our research, the distribution of data points within the dataset can affect the performance of LOF. Specifically, if the dataset is comprised of large regions of sparse data, LOF may have difficulties in accurately distinguishing between outliers and noise.

3.7 Support Vector Machines (SVM)

Support Vector Machines (SVM) are supervised learning models primarily used for classification and regression tasks. The main idea behind SVM is to find a hyperplane that best separates the data into different classes. In a two-dimensional space, this hyperplane is a line. For higher dimensions, it becomes a plane or a set of planes that aim to maximize the margin between different classes of data points.

The hyperplane is represented by the equation $wx - b = 0$, where w is the weight vector and b is the bias. To find the optimal hyperplane, SVM aims to maximize the margin M, which is the distance between the nearest data point from any class to the hyperplane. Mathematically, the margin is expressed as:

$$M = \frac{2}{\|w\|} \tag{3.11}$$

The algorithm operates by minimizing the following objective function with respect to w and b:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \tag{3.12}$$

subject to the constraint $y_i(wx_i - b) \geq 1$ for each data point x_i and its corresponding label y_i.

For non-linearly separable data, SVM employs the kernel trick to implicitly map the input space to a higher-dimensional feature space. Common kernels include polynomial, radial basis function (RBF), and sigmoid kernels. The choice of kernel can significantly affect the performance of the model.

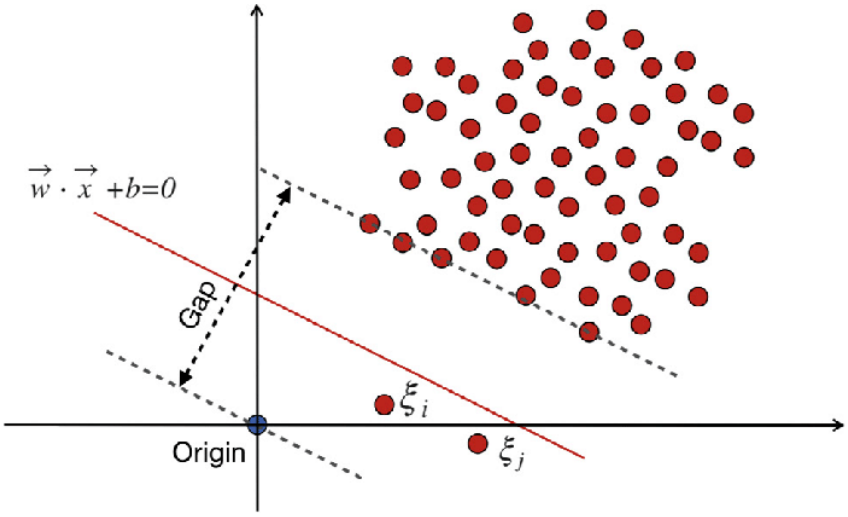

Figure 3.6: Graphical representation of the SVM method [35].

SVM offers several advantages. It is effective in high-dimensional spaces and is versatile, thanks to the various kernel functions. Furthermore, only a subset of training points, known as support vectors, are used in the decision function, making the algorithm efficient. However, SVMs have their downsides. They are sensitive to the choice of the kernel and its parameters, and they can be computationally intensive for large datasets. Additionally, SVMs are not well-suited for multi-class problems and may require complex schemes like one-vs-one or one-vs-all for such tasks.

3.8 Neural Network Models: Autoencoders

Neural networks are a subclass of machine learning algorithms modeled after the human brain. They are composed of layers of interconnected nodes or neurons. Among various types of neural network architectures, autoencoders are unsupervised neural networks employed for anomaly detection, dimensionality reduction, and feature learning, among other tasks.

An autoencoder network is typically symmetrical about its central hidden layer, known as the encoding layer. The network can be divided into two primary components: the encoder and the decoder. The encoder compresses the input into a latent-space representation and encodes the input as an internal fixed-size representation in reduced dimensionality. The decoder, on the other hand, works to reconstruct the input data from this internal representation.

Mathematically, an encoder function f maps an input x to a hidden representation h:

$$h = f(x; W) \tag{3.13}$$

where W are the weights of the encoder.

The decoder function g then maps h back to the reconstructed input $\hat{x}$:

$$\hat{x} = g(h; W') \tag{3.14}$$

where W' are the weights of the decoder.

The primary objective of an autoencoder is to minimize the reconstruction error, usually represented by a loss function L:

$$L(x, \hat{x}) = ||x - \hat{x}|| \tag{3.15}$$

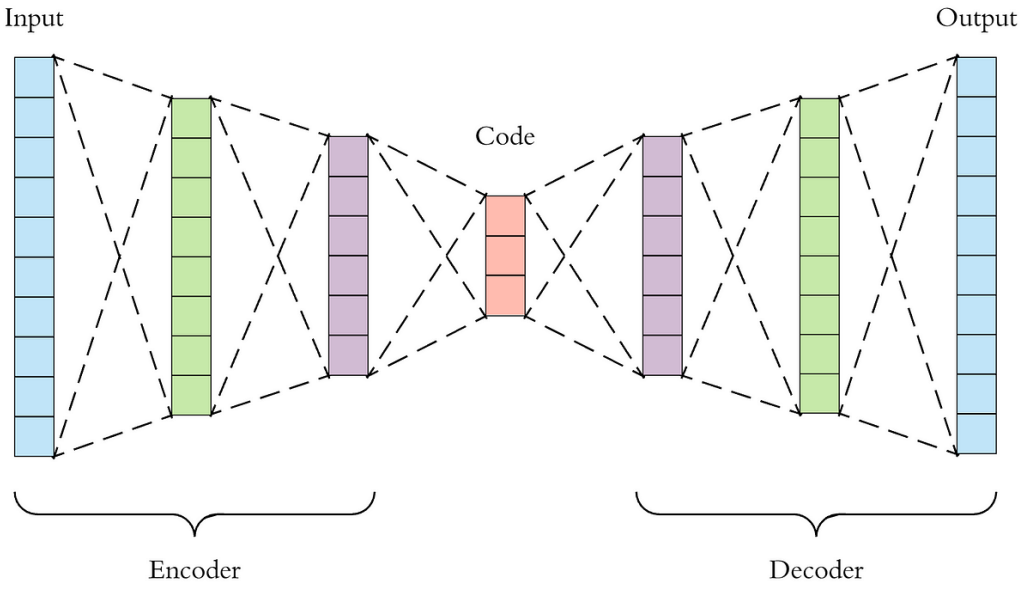

Figure 3.7: Graphical representation of an autoencoder neural network [36].

In an autoencoder network, the architecture often starts with an input layer that matches the dimensionality of the feature space. This is followed by one or more hidden layers that progressively reduce the dimensionality, leading to the central encoding layer. The decoder starts from the encoding layer and gradually increases the dimensions through its hidden layers, culminating in an output layer of the same size as the input layer.

The choice of the number of layers and neurons within those layers is often problem-dependent and influenced by the complexity of the input data. More complex data might require deeper architectures with more neurons, but care should be taken to avoid overfitting, where the model learns the training data too well but performs poorly on unseen data.

Hyperparameters are parameters whose values are set before the learning process begins. In the context of autoencoders, important hyperparameters include:

- *Learning Rate*: Controls how much to update the model parameters during training. A too-high rate can cause the model to converge too quickly and possibly overshoot the optimal values. A too-low rate could result in a very long training process.
- *Batch Size*: The number of samples processed before updating the model parameters. Smaller batch sizes often provide a regularizing effect and lower generalization error.
- *Number of Epochs*: An epoch is one complete forward and backward pass of all the training examples. The training process is often repeated for a certain number of epochs to ensure the model learns the representation effectively.
- *Activation Functions*: Functions like ReLU (Rectified Linear Unit), sigmoid, or tanh, applied to the neurons' output.

- *Regularization Techniques*: Such as dropout or L1/L2 regularization, these techniques can help prevent overfitting.

Hyperparameter optimization often involves techniques like grid search, random search, or more sophisticated methods like Bayesian optimization. The aim is to find the hyperparameter values that yield the most accurate model in terms of minimizing the loss function (3.15).

The advantage of autoencoders is their ability to learn non-linear transformations with a simple architecture, making them suitable for complex tasks such as feature learning and anomaly detection. Additionally, they can automatically learn features necessary for various tasks, thereby reducing the need for manual feature engineering. However, they have disadvantages too. Autoencoders are sensitive to the architecture and hyperparameters; a poor choice can result in overfitting or underfitting. Furthermore, training autoencoders can be computationally intensive and may require a lot of data for the model to generalize well.

In summary, autoencoders are powerful tools in the neural network arsenal that offer both sophistication and depth in solving complex machine learning problems. However, a nuanced understanding and careful selection of architecture and hyperparameters are crucial for optimal performance.

3.9 Evaluation Metrics

Evaluating the efficiency of outlier detection methods is an essential step, especially when employing a range of techniques. To aid in this evaluation, custom software scripts will be developed, serving the specific purpose of facilitating the visual inspection process. These tailored tools will enable the graphical representation of detected outliers, providing a contextual view within the dataset for more accurate assessment.

Visual inspection serves as an immediate and qualitative method to assess how well each algorithm is performing. This approach becomes particularly valuable when the dataset lacks a well-defined set of known outliers, making it challenging to establish a ground truth for comparison.

A significant point to consider is that many of the algorithms employed in this research are configured to identify a specific number of outliers. It's essential to understand that this predetermined number may not align with the actual number of outliers present in the dataset. Therefore, it's possible for these algorithms to label points as outliers even when they might not be. Given this context, thorough visual inspection becomes indispensable for distinguishing between true and false positives.

4 RESULTS

This chapter is dedicated to the comprehensive presentation and analysis of the results obtained from various outlier detection methods applied to the EXFOR Proton Reactions dataset. The purpose is to evaluate the effectiveness of these methods in identifying outliers.

The chapter is organized into several sections, each focusing on a specific outlier detection method. We begin with data preprocessing, setting the groundwork for the subsequent analyses. Then, traditional statistical and machine learning methods such as the Interquartile Range (IQR), Isolation Forest, DBSCAN, and Local Outlier Factor (LOF) are discussed. Each of these sections covers the implementation, graphical analysis, and concludes with an evaluation of the method's efficacy. This is followed by a foray into neural network models, specifically focusing on Autoencoders, which offers a more complex but potentially more effective approach for outlier detection.

Due to the large volume of data and the nature of the algorithms implemented, it is impractical to display all the results from the execution of the various scripts created. As such, only a subset of the results will be presented in the 'Results and Graphical Analysis' subsection of each method. However, all the necessary scripts for reproducing these results are included in this work and can also be accessed at GitHub EXFOR-ProtonReactions-Analysis Repository.

4.1 Data Preprocessing

Data preprocessing is a crucial and often time-consuming stage in both machine learning and various data analytics initiatives. While the core framework for this process was outlined in Section 3.2 of Chapter 3, this section provides a more in-depth look. We will focus on the results from each preprocessing step, discuss how these results are useful in machine learning and data analytics, and consider their potential applications.

To facilitate the replication or adaptation of our work, a Python Jupyter Notebook named `EXFOR_ProtonReactions_Data_Preprocessing.ipynb` has been developed. This notebook encompasses all steps required for data preprocessing. We will provide a line-by-line walkthrough of the script to clarify the logic and mechanics behind each procedure. The notebook is accessible via the GitHub repository managed by monleon96, which can be found at GitHub EXFOR-ProtonReactions-Analysis Repository. Additionally, all custom Python functions created specifically for this project are documented in the Annexes of this thesis.

Our initial assumptions are that `EXFORTABLES` is pre-installed on your computer with its default folder structure, as outlined in reference [37]. We also presume that you have successfully imported the Experiment Python class from Annex A as well as the utility functions from Annex B. Both are available in the aforementioned GitHub repository under the filenames:

- `EXFOR_ProtonReactions_Experiment_Class.py`
- `EXFOR_ProtonReactions_UtilityFunctions.py`

With these prerequisites met, you can initiate a new Jupyter notebook and access the experiments from `EXFORTABLES` by specifying the computer path to the proton-induced reactions, termed as 'p'.

```
path = "path_to_proton-induced_reactions"
experiments = read_proton_experiments_from_exfortables(path)
```

The function `read_proton_experiments_from_exfortables()` (see Annex B.6) reads each experiment's files from `EXFORTABLES`. It puts the important details into an `Experiment` object (see Annex A). When you run this, the variable `experiments` will hold a list of `Experiment` objects, which contain all the needed data from EXFOR's proton experiments.

To make things easier and avoid reading the `EXFORTABLES` files every time, we have two ways to save this data: as a binary file or as a text file (see Annex B.2 and B.4).

```
write_experiments_to_binary(experiments, "EXFOR_ProtonReactions_Database
    .bin")
write_experiments_to_txt(experiments, "EXFOR_ProtonReactions_Database.
    txt")
```

Using these functions, you can create either a binary or a text file. Later on, you only need to read one of these files to get back the saved data. This can be done with the functions `read_experiments_from_binary()` and `read_experiments_from_txt()` (see Annex B.3 and B.5).

```
exp_from_bin = read_experiments_from_binary("
    EXFOR_ProtonReactions_Database.bin")
exp_from_txt = read_experiments_from_txt("EXFOR_ProtonReactions_Database
    .txt")
```

Now, you might wonder why we offer two types of files. Binary files are quicker to load but rely on Python's pickle module. This can sometimes lead to issues with serialization—essentially, converting the data back and forth between file and program. Text files are more straightforward but might take longer to read into the program.

Once the experiments are loaded into a Python list, you can see what is inside by just printing one item from the list. This will show you all the details for that specific experiment.

```
print(exp_from_bin[3])
```

Output:

```
Title:  p-Ag000-MT002-Heiberg-004550051-ang-E0424.000.1957
Target Z:  47
Target A:  0
Projectile:  p
Reaction:  (p, el)
Incident energy:  424.000 MeV
Quantity:  Angular distribution
Frame:  L
MF:  4
MT:  2
X4 ID:  004550051
X4 code:  47-AG-0(P,EL)47-AG-0,,DA
Author:  Heiberg
```

```
Year:  1957
Data points:  15
Data:       Angle(deg)  xs(mb/sr)  dxs(mb/sr)  dAngle(deg)
0       5.69977     4710.0       500.0           NaN
1       6.39997     2520.0       110.0           NaN
2       7.59980      379.0        48.0           NaN
3       8.29993      425.0        28.0           NaN
4       9.00012      400.0        27.0           NaN
5       9.79996      397.0        40.0           NaN
6      10.40020      539.0        33.0           NaN
7      11.60010      263.0        14.0           NaN
8      12.29990      237.0        15.0           NaN
9      12.80010      118.0         6.0           NaN
10     13.90010       47.2         5.9           NaN
11     14.80010       51.9        11.8           NaN
12     15.90010       42.5         5.9           NaN
13     18.50010       29.5         3.5           NaN
14     19.20010       16.5         2.4           NaN
Reference:
 E.Heiberg
Angular Distribution of 424-MeV Polarized Protons Elastically Scattered
   from Various Nuclei.
Jour. Physical Review Vol.106, p.1271, 195
```

You can also focus on just the data points by using a dataframe. This makes it easy to handle the data for further analysis.

```
exp_from_bin[3].data
```

Output:

```
    Angle(deg)  xs(mb/sr)  dxs(mb/sr)  dAngle(deg)
0       5.69977     4710.0       500.0           NaN
1       6.39997     2520.0       110.0           NaN
2       7.59980      379.0        48.0           NaN
3       8.29993      425.0        28.0           NaN
4       9.00012      400.0        27.0           NaN
..          ...        ...         ...           ...
10     13.90010       47.2         5.9           NaN
11     14.80010       51.9        11.8           NaN
12     15.90010       42.5         5.9           NaN
13     18.50010       29.5         3.5           NaN
14     19.20010       16.5         2.4           NaN

[15 rows x 4 columns]
```

Another helpful feature is the ability to convert an entire experiment's information into a dataframe. Here, each row represents a data point from the experiment, and the experiment's attributes become the columns of the dataframe.

```
exp_from_bin[3].to_dataframe()
```

Output:

```
    Angle(deg)  xs(mb/sr)  dxs(mb/sr)  ...  MT   Author  Year
0      5.69977     4710.0       500.0  ...   2  Heiberg  1957
1      6.39997     2520.0       110.0  ...   2  Heiberg  1957
2      7.59980      379.0        48.0  ...   2  Heiberg  1957
3      8.29993      425.0        28.0  ...   2  Heiberg  1957
4      9.00012      400.0        27.0  ...   2  Heiberg  1957
..         ...        ...         ...  ...  ..      ...   ...
10    13.90010       47.2         5.9  ...   2  Heiberg  1957
11    14.80010       51.9        11.8  ...   2  Heiberg  1957
12    15.90010       42.5         5.9  ...   2  Heiberg  1957
13    18.50010       29.5         3.5  ...   2  Heiberg  1957
14    19.20010       16.5         2.4  ...   2  Heiberg  1957

[15 rows x 21 columns]
```

You can also make plots directly using a method from the class. This method offers optional settings to tweak the graph's appearance, like its size or the scales of the x and y axes. If the experiment includes uncertainty data, the plot will also display error bars, giving you a full view of the data's quality.

```
exp_from_bin[0].plot(ylog=True)
```

Output:

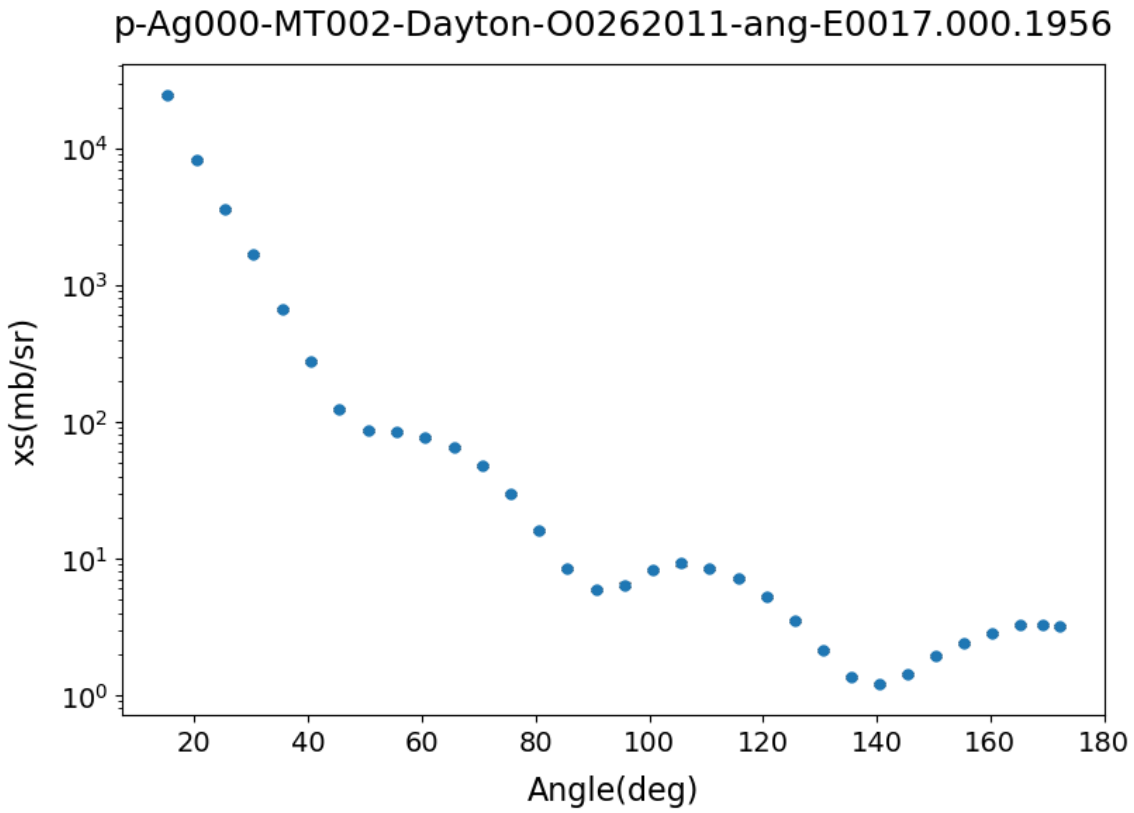

Figure 4.1: Experimental results obtained using the plot method of the `Experiment` class.

```
exp_from_bin[2135].plot(ylog=True)
```

Output:

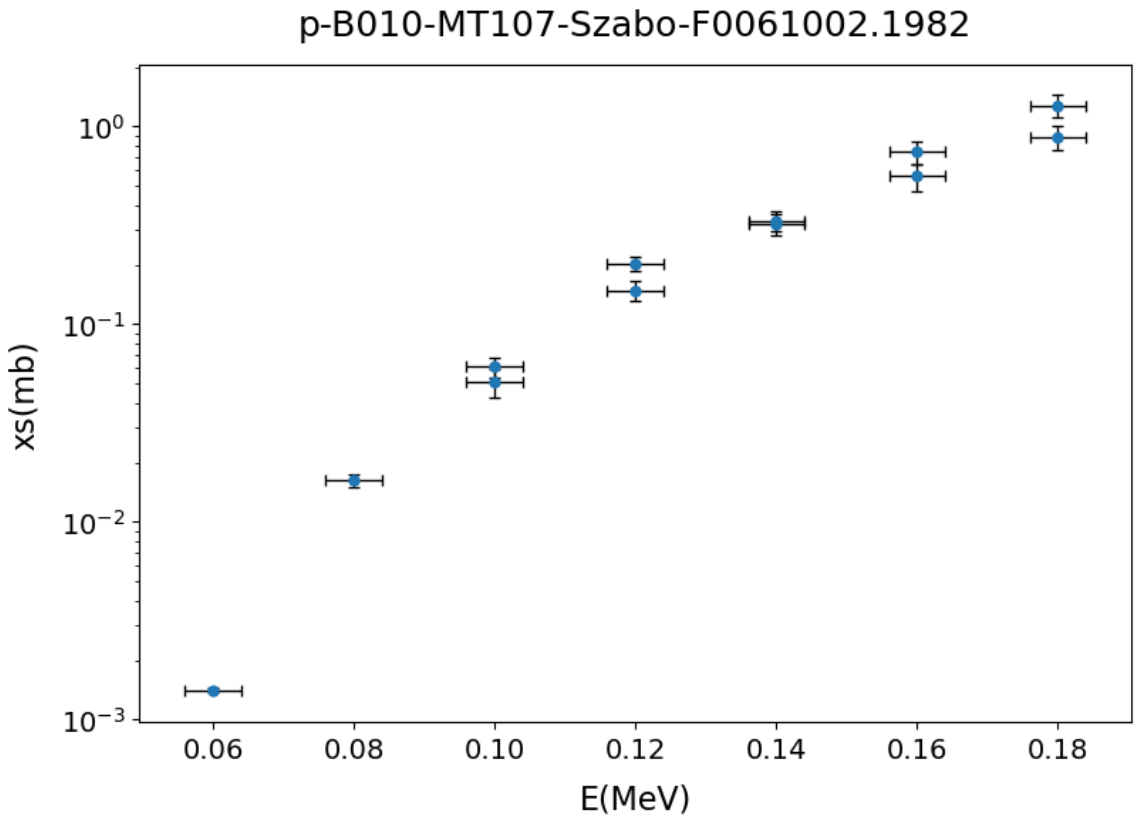

Figure 4.2: Experimental results with uncertainties obtained using the plot method of the `Experiment` class.

When it comes to analyzing or sorting through the experiments, there are specialized functions to make things easier. For example, the function `get_unique_values()` takes a list of experiments and an attribute name as inputs. It then returns all the unique values that appear for that attribute among the listed experiments (see Annex B.8).

```
MT_values = get_unique_values(exp_from_bin, "MT")
```

If you are interested in experiments that meet a specific condition, you can filter them out. The function `filter_experiments()` does this for you. For example, you can filter out experiments where $MT = 42$ from a larger list (see Annex B.7).

```
MT42_exp = filter_experiments(exp_from_bin, "MT", 42)
```

Output:

```
14 experiments with MT = 42
```

You can also utilize the `plot_experiments()` function (see Annex B.12) to display multiple similar experiments—those with the same X and Y axes—on a single graph. This is helpful

for comparing data, like in an example where 14 experiments with $MT = 42$ are showcased together.

```
plot_experiments(MT42_exp)
```

Output:

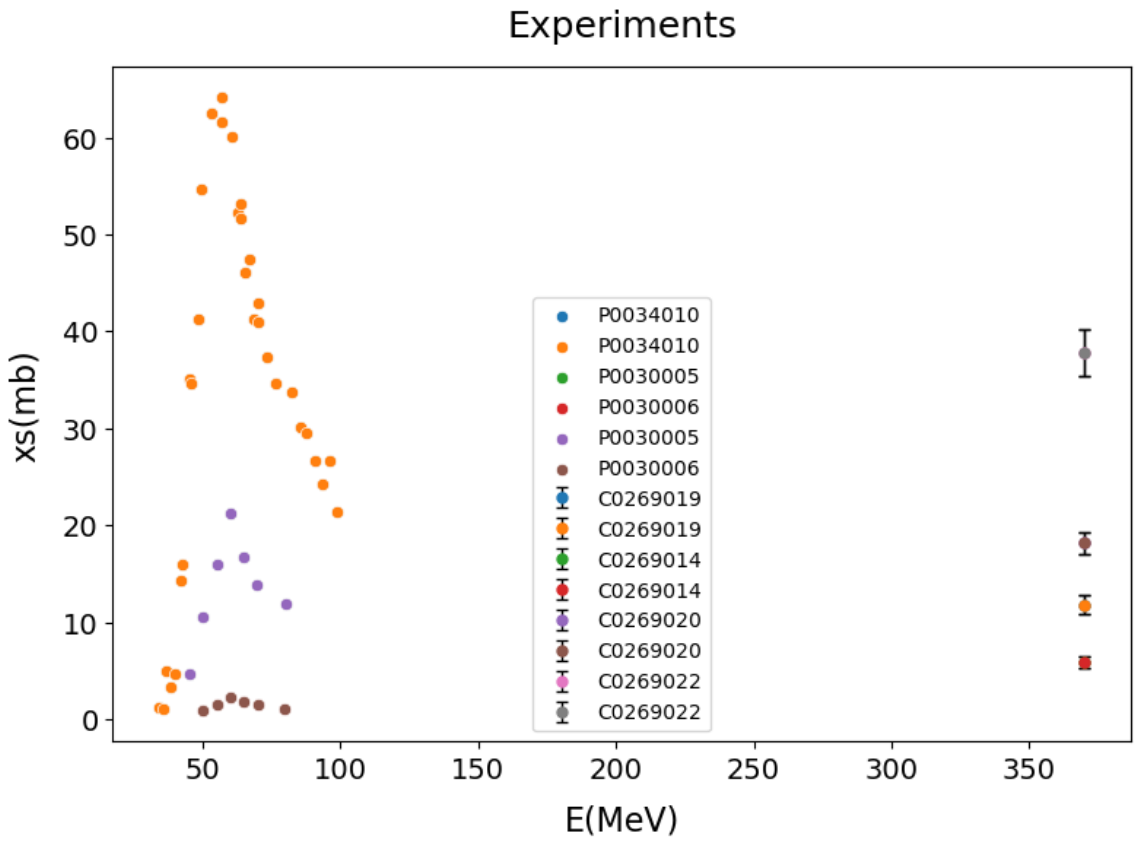

Figure 4.3: Experimental results of all proton-induced experiments with $MT = 42$.

TWith over 27,000 subentries for proton-induced reactions in EXFOR, not every experiment measures the same things. For machine learning applications, it is useful to sort experiments based on what is actually being measured. Essentially, this involves grouping experiments by their data. Each experiment object has a `.data`. attribute that holds a table with four columns: X values, Y values, uncertainties in Y, and uncertainties in X. If there are no uncertainties, these will be marked as `NaN`. To organize the experiments in this manner, the function `classify_experiments_by_data()` is used (see Annex B.11).

```
df = classify_experiments_by_data(exp_from_bin)
```

This function creates a CSV file for each of the each group of similar experiments, focusing only on the details that matter for analysis. This means that certain attributes like the title or the author aren not included. Be aware that this process could take a long time since it deals with millions of data points. Once the files are created, you can easily load them into a Python dataframe using the pandas library. For additional details regarding the classification of experimental groups, please refer to Annex D.

When turning this data into CSV files, we use a technique called "one-hot encoding" to turn everything into numbers. This makes it easier to use machine learning tools on the data. Here are some additional things to consider:

1. Some columns may have the same value for every row in each group. In machine learning, these uniform columns are not useful. You can remove them using the `clean_dataframe()` function (see Annex B.9).

2. Machine learning often does not need uncertainty data. You can also remove these columns using the `clean_dataframe()` function.

3. Many machine learning tools can not handle `NaN` values. The `clean_dataframe()` function can usually remove these, but for Group 4, you will need an extra step. Here, the code divides Group 4 into two sub-groups to ensure there are no `NaN` values. The information is then saved into two different CSV files.

```
df = pd.read_csv("EXFOR_ProtonReactions_Classified_Group_4.csv")
df_nan = df[df["final_A"].isna()]
df_not_nan = df[df["final_A"].notna()]
df_nan.to_csv("EXFOR_ProtonReactions_Classified_Group_4_1.csv",
    index=False)
df_not_nan.to_csv("EXFOR_ProtonReactions_Classified_Group_4_2.csv",
     index=False)
```

4. A last column named `X4_ID` is added to each row in order to identify the EXFOR experiments that corresponds to each value and be able to address it correctly. This column must be removed before applying machine learning techniques as it contains string values that can not be handle by algorithms.

There is an additional function named `classify_experiments()` (see Annex B.10). This function offers a different way to group the experiments, although it is not as well-suited for machine learning applications as the one described earlier. Specifically, it organizes the experiments based on a chosen attribute. For example, using this function with the attribute MT will create separate CSV files, where each file contains experiments that share the same MT value. Unlike the previous function, this one does not apply one-hot encoding and merely converts the information into dataframes. While it might not be the best choice for machine learning, it can still be useful for other types of data analysis.

```
classify_experiments(exp_from_bin, MT)
```

Finally, the function `plot_outliers()` (see Annex B.13) is designed to visualize outliers identified through various detection methods. To optimize data visualization, experiments sharing identical attributes are consolidated into a single graph. The legend serves as an intuitive guide for identifying the EXFOR experiments associated with each data point, with outliers being denoted by a red cross. Examples of this will be shown in the following sections along with the results of the outlier detection.

```
plot_outliers(outliers_df, exp_from_bin)
```

4.2 Interquartile Range (IQR)

4.2.1 Implementation of IQR Method

The outlier detection using the Interquartile Range (IQR) method is implemented through Python code, which applies the theoretical framework described in Section 3.3. The mathematical formulations for IQR and the lower and upper bounds used for detecting outliers are given by Equations 3.2, 3.3, and 3.4 respectively. This methodology is implemented through two functions `detect_outliers_IQR` and `IQR_method` that can be found in Annex C.

The `detect_outliers_IQR` function takes a grouped dataframe (`grouped_df`), an `Experiment` object (`experiment`), a limit factor (`limit`), and a Boolean flag to consider uncertainties (`uncertainties`).

The function calculates the first ($Q1$) and third ($Q3$) quartiles for the specified columns in the grouped dataframe:

```
Q1 = grouped_df[[first_column, second_column]].quantile(0.25)
Q3 = grouped_df[[first_column, second_column]].quantile(0.75)
```

Following this, the IQR is computed according to equation 3.2 and the conditions for determining outliers are then applied:

```
outlier_condition = ((grouped_df[[first_column, second_column]] < (Q1 -
    limit * IQR)) | (grouped_df[[first_column, second_column]] > (Q3 +
    limit * IQR)))
```

Finally, the function filters and returns all rows that satisfy the outlier condition.

The `IQR_method` function serves as a wrapper for `detect_outliers_IQR`. It first identifies columns by which to group the DataFrame and then applies `detect_outliers_IQR` to each valid group, ensuring the group meets the minimum number of observations (`min_observations`):

```
grouped = df.groupby(groupby_columns).size().reset_index(name="count")
valid_groups = grouped[grouped["count"] >= min_observations][
    groupby_columns].to_dict("records")
```

If uncertainties are to be considered (`uncertainties=True`), the function alters the conditions for outlier identification accordingly. The uncertainty values are used to define lower and upper values for each data point:

```
lower_values = grouped_df[second_column] - grouped_df[third_column]
upper_values = grouped_df[second_column] + grouped_df[third_column]
```

Both ends of the uncertainty range for a data point must lie outside the IQR-defined bounds for it to be considered an outlier.

4.2.2 Results and Graphical Analysis

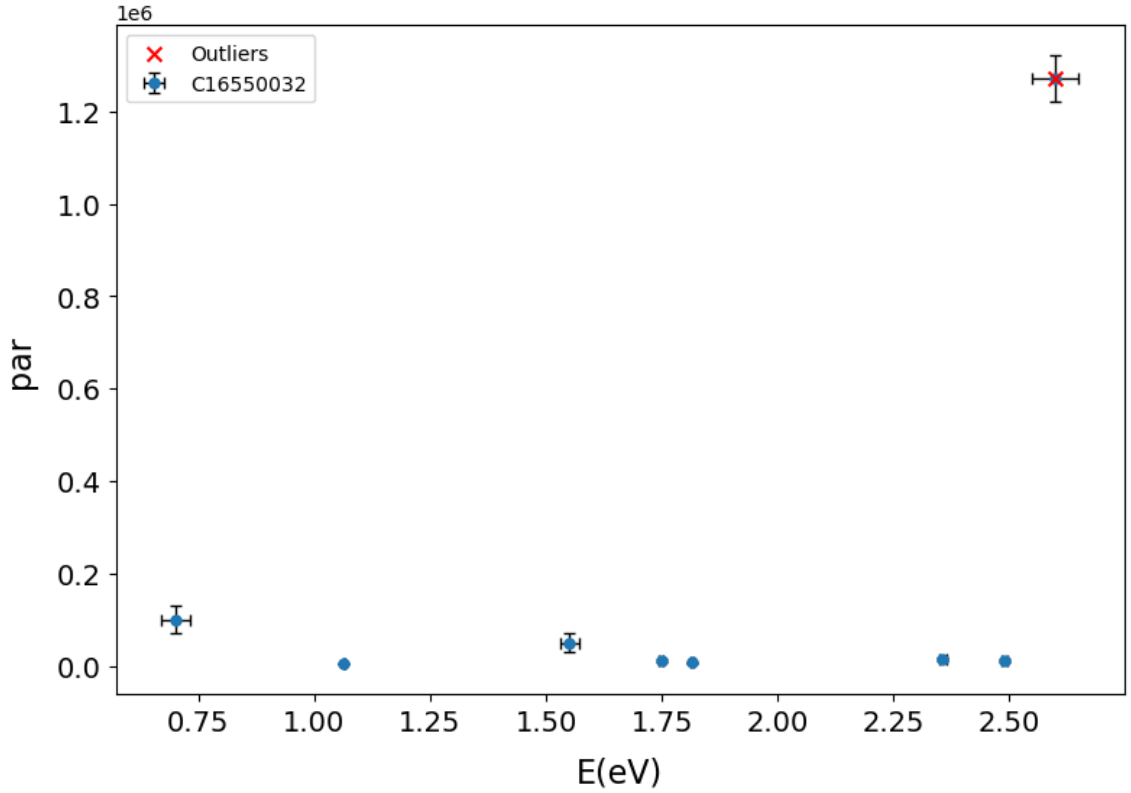

Figure 4.4: Outlier Detection in Group 6 using IQR Method.

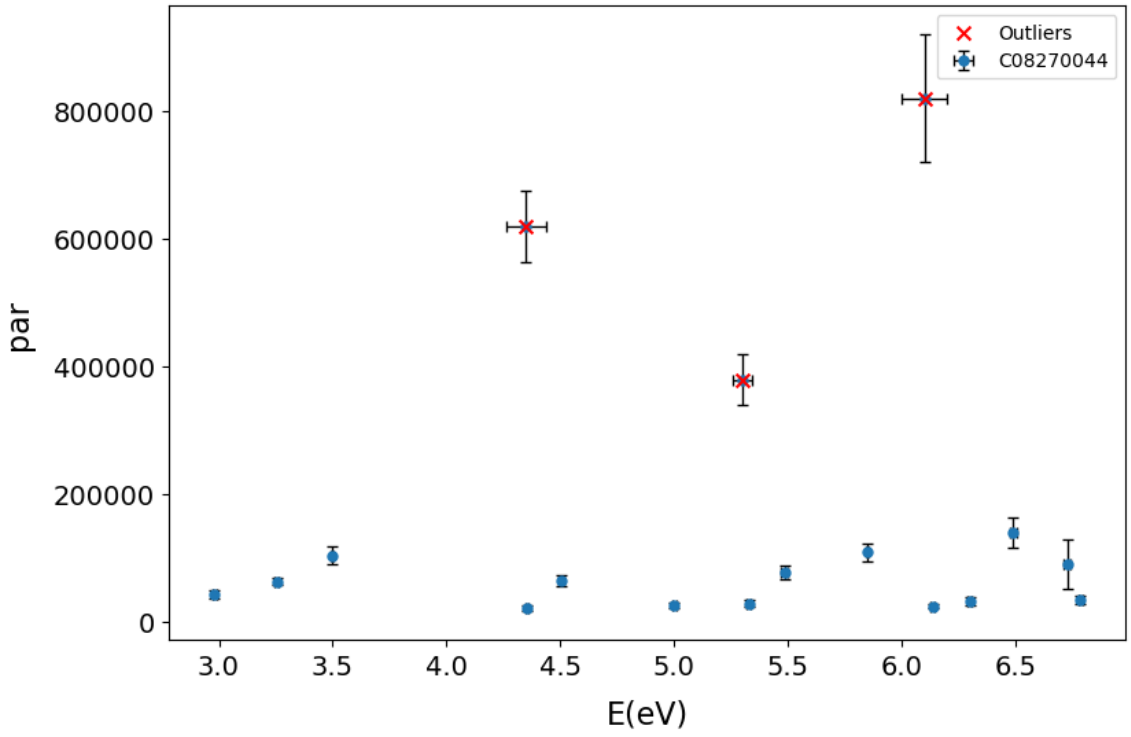

Figure 4.5: Outlier Detection in Group 6 using IQR Method.

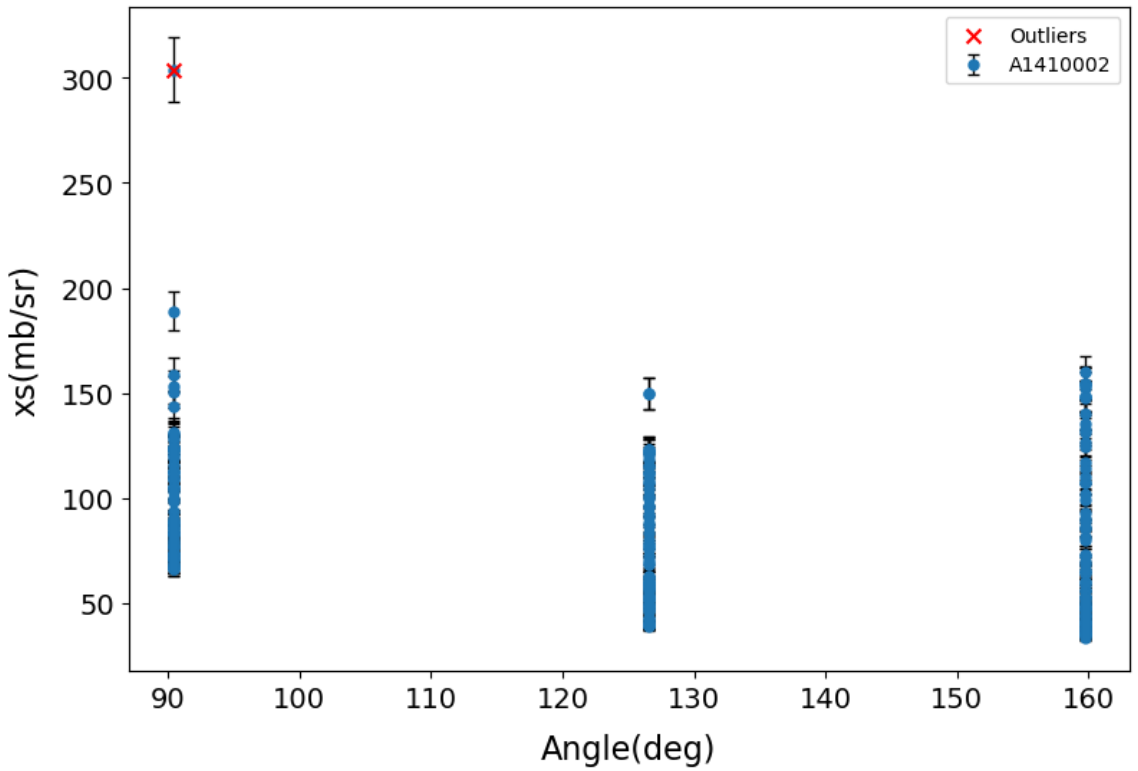

Figure 4.6: Outlier Detection in Group 1 using IQR Method.

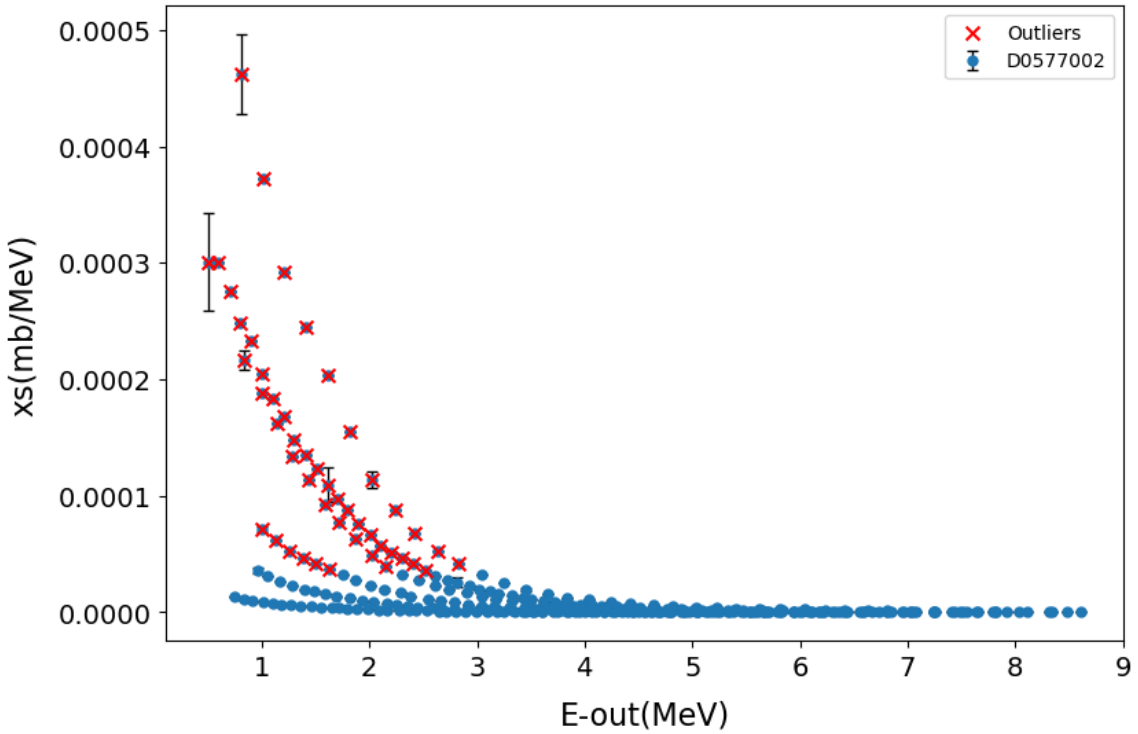

Figure 4.7: Outlier Detection in Group 5 using IQR Method.

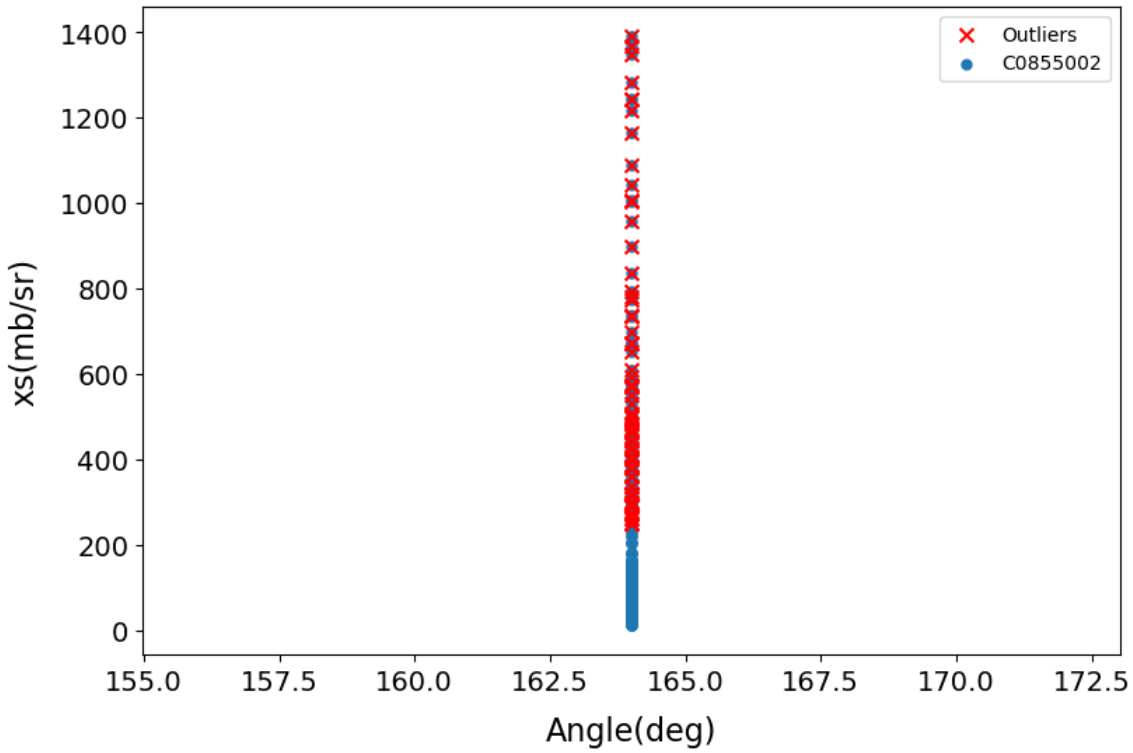

Figure 4.8: Outlier Detection in Group 1 using IQR Method.

While graphs 4.4, 4.5, and 4.6 demonstrate success in identifying potential outliers across different groups, graphs 4.7 and 4.8 highlight the limitations of the IQR method. Specifically, these latter figures exhibit a high rate of false positives. This is largely attributable to the non-uniform distributions present within the data sets, causing the IQR method to incorrectly classify a significant number of inliers as outliers.

4.2.3 Conclusion on IQR Method

The IQR method is fundamentally a statistical technique that allows for the consideration of uncertainties in its calculations. This feature can be advantageous for reducing false positives. However, our study reveals limitations in the applicability of the IQR method to our specific data sets. The nature of the data, which follows varying types of distributions, is not particularly well-suited for IQR, a method that performs better on uniformly distributed data.

In particular, any peak in the data, resulting from its underlying distribution, could be mistakenly identified as an outlier. Consequently, this makes the IQR method ill-suited for our purposes, as it cannot reliably distinguish between genuine outliers and data points that are naturally part of a non-uniform distribution.

The visualizations and analyses thus indicate that while the IQR method has its merits, it is not the most appropriate technique for outlier detection in this specific context.

4.3 Isolation Forest

4.3.1 Implementation of Isolation Forest Method

The script for this implementation can be found in the GitHub repository under the name `OutlierDetection_IsolationForest_Method.ipynb`. We utilize the `IsolationForest` class from the `sklearn.ensemble` package. Below is a snippet of the code outlining the algorithm's application to our dataset:

```
# Create the model
clf = IsolationForest(contamination=0.001)  # contamination = % of
    outliers
# Train the model
clf.fit(df.iloc[:, :-1])
# Predict outliers
outliers = clf.predict(df.iloc[:, :-1])
```

After training the model, the `predict` method is used to identify the outliers in the dataset. These are then flagged and added to the dataframe:

```
# Add the prediction to the dataframe
df["is_outlier"] = np.where(outliers == -1, "Yes", "No")
# Get the outliers dataframe
outliers_df = df[df["is_outlier"] == "Yes"]
outliers_df = outliers_df.drop(columns=["is_outlier"])
```

The code first adds a new column to the dataframe, labeling each data point as an outlier or an inlier. Then, the proportion of detected outliers is calculated and printed.

4.3.2 Results and Graphical Analysis

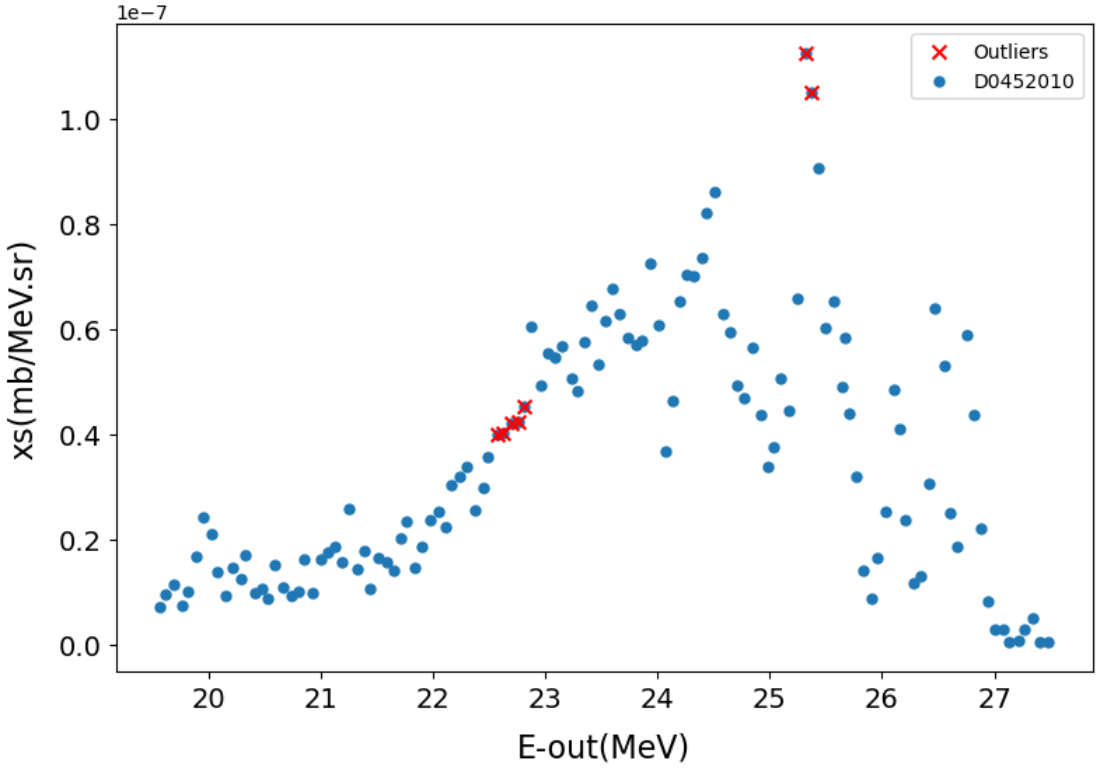

Figure 4.9: Outlier Detection in Group 2 using Isolation Forest Method.

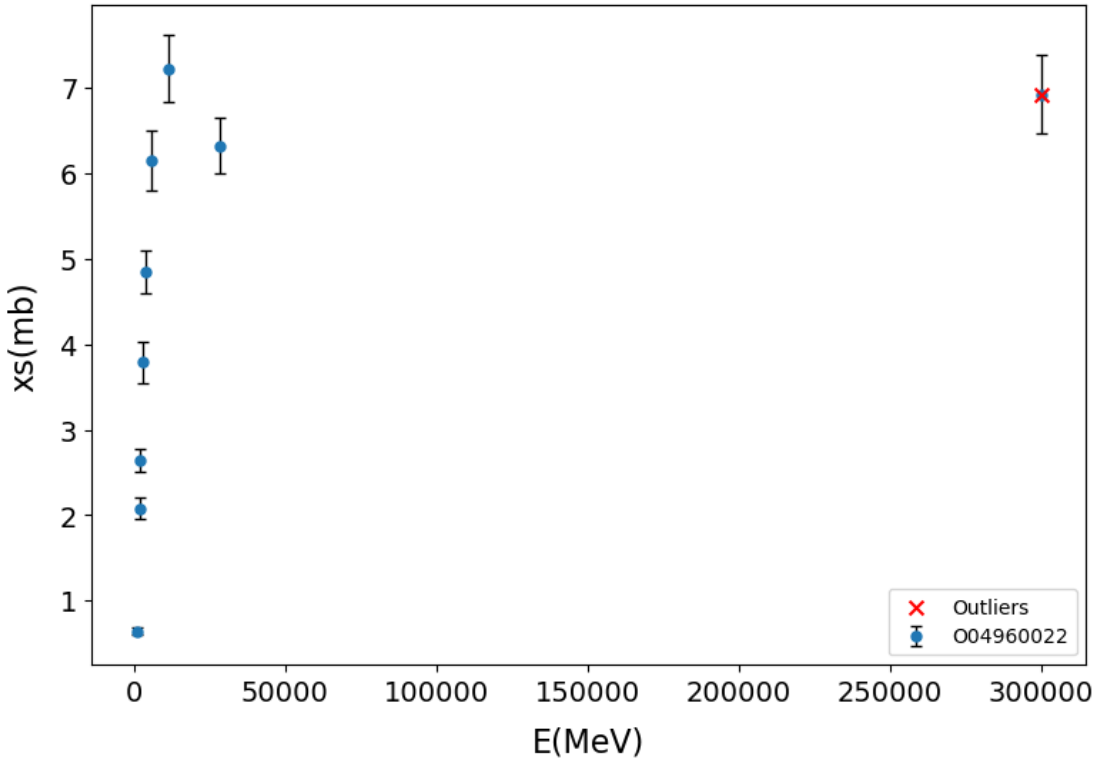

Figure 4.10: Outlier Detection in Group 4.2 using Isolation Forest Method.

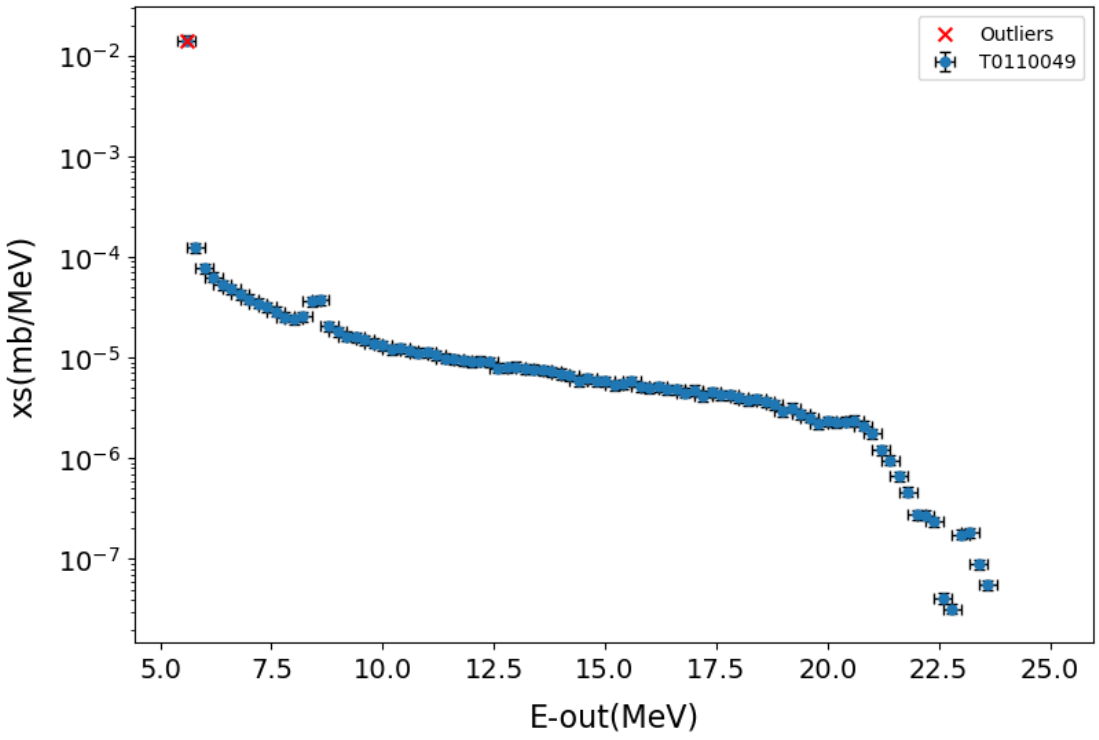

Figure 4.11: Outlier Detection in Group 5 using Isolation Forest Method.

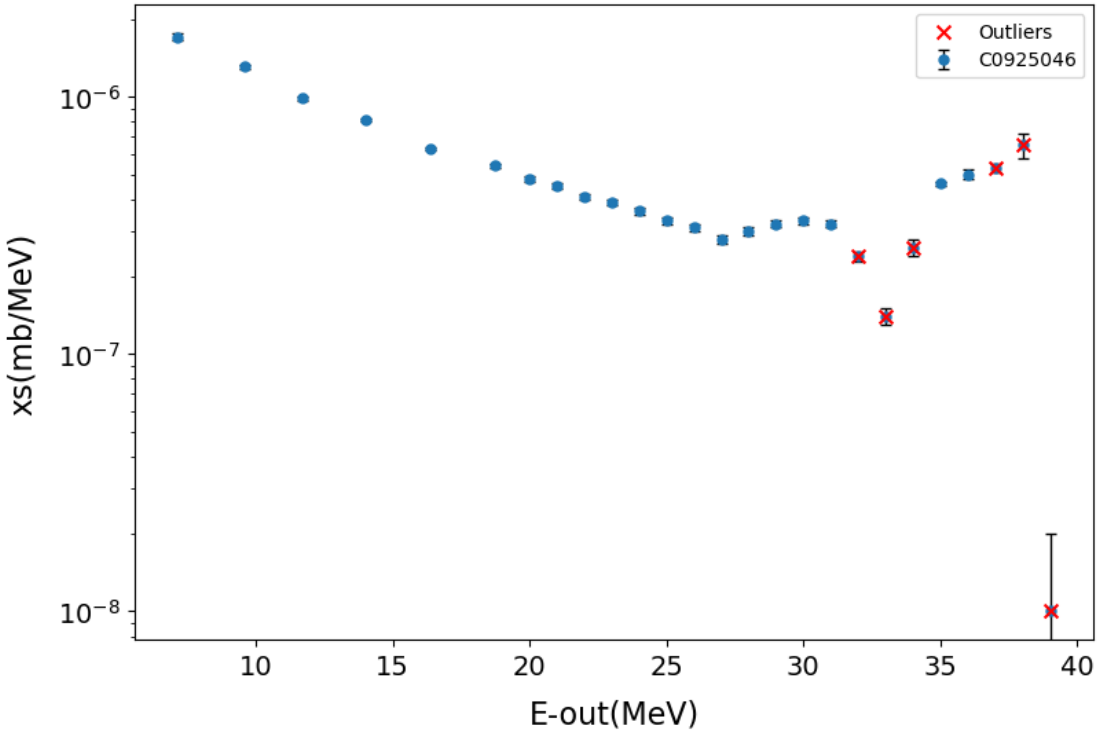

Figure 4.12: Outlier Detection in Group 5 using Isolation Forest Method.

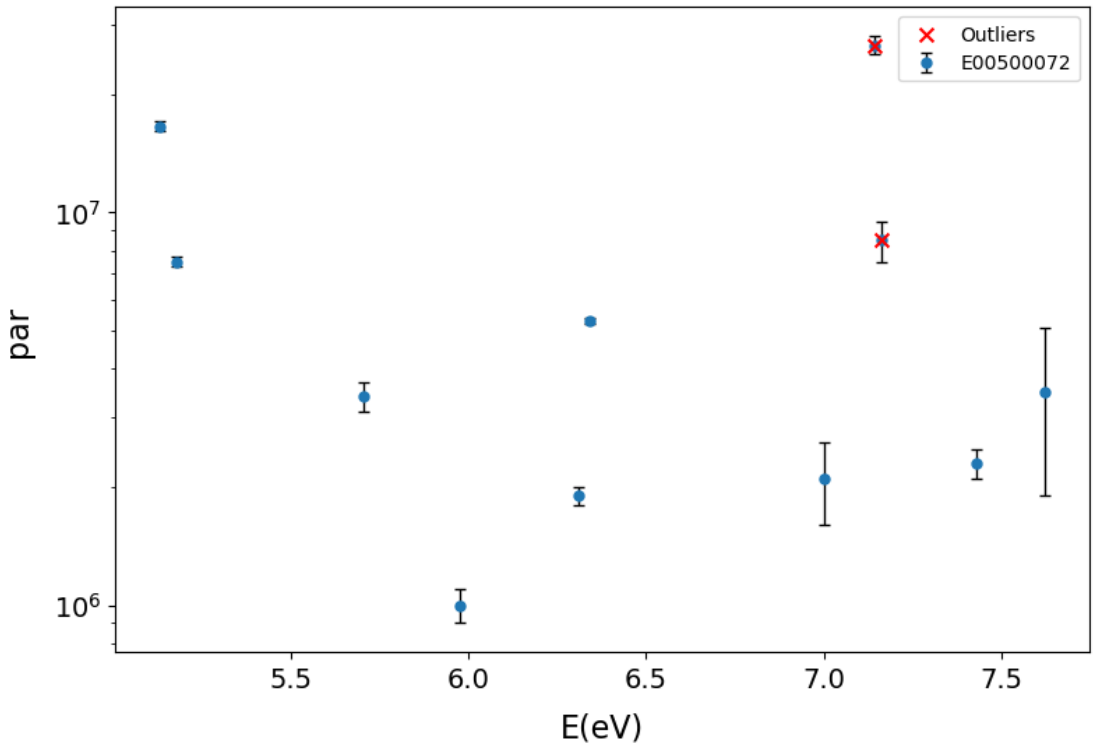

Figure 4.13: Outlier Detection in Group 6 using Isolation Forest Method.

Through graphs 4.9 to 4.13, a range of potential outliers are displayed. In some cases, the detected points are clearly outliers, supporting the efficacy of the Isolation Forest method. However, in other cases, it becomes challenging to ascertain whether the detected points are indeed outliers or simply inliers that appear isolated due to the sparsity of data points in those regions.

4.3.3 Conclusion on Isolation Forest Method

The Isolation Forest method is an effective technique for outlier detection, generally surpassing the performance of statistical methods such as the IQR method. However, the algorithm is highly sensitive to the specific distribution of the dataset. One significant limitation lies in how the method handles sparse regions of the data. When data points are lacking in a specific range—due to insufficient sampling rather than because the data points are genuine outliers—these isolated data points may be incorrectly flagged as outliers. This false positive detection represents a significant drawback in applying the Isolation Forest method in this study's context.

4.4 DBSCAN

4.4.1 Implementation of DBSCAN Method

The script for the DBSCAN implementation is available in the GitHub repository, named as `OutlierDetection_DBSCAN_Method.ipynb`. We use the `DBSCAN` class from the `sklearn.cluster` package for this purpose. Below is a simplified version of the code to illustrate how the algorithm is applied to our dataset:

1. The data is first scaled using the `StandardScaler` from the `sklearn.preprocessing` package. This standardization is crucial as DBSCAN is sensitive to the scale of the data.

```
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df_without_ids)
```

2. After scaling, the DBSCAN algorithm is applied to the data.

```
clustering = DBSCAN().fit(df_scaled)
```

3. Outliers are identified by finding where the clustering labels equal -1.

```
outlier_positions = np.where(clustering.labels_ == -1)  # Positions of the outliers
```

4. Finally, the data is then descaled to revert it back to its original state.

```
df_descaled = pd.DataFrame(scaler.inverse_transform(df_scaled), columns=df_without_ids.columns)
```

4.4.2 Results and Graphical Analysis

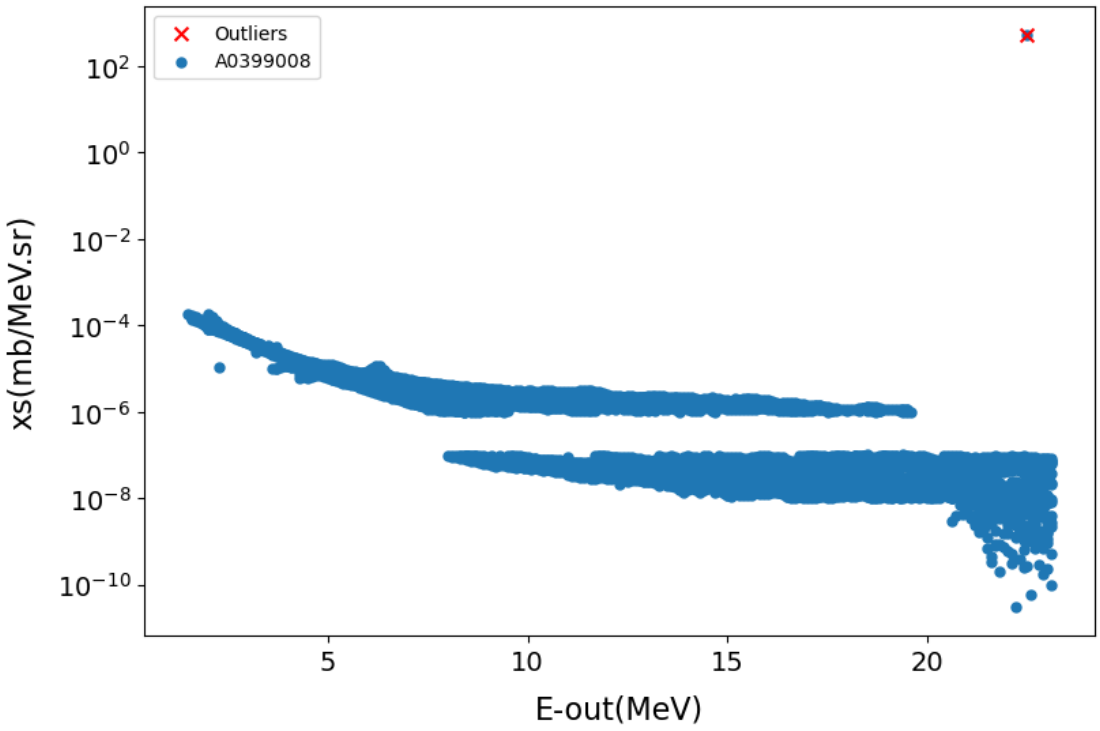

Figure 4.14: Outlier Detection in Group 2 using DBSCAN Method.

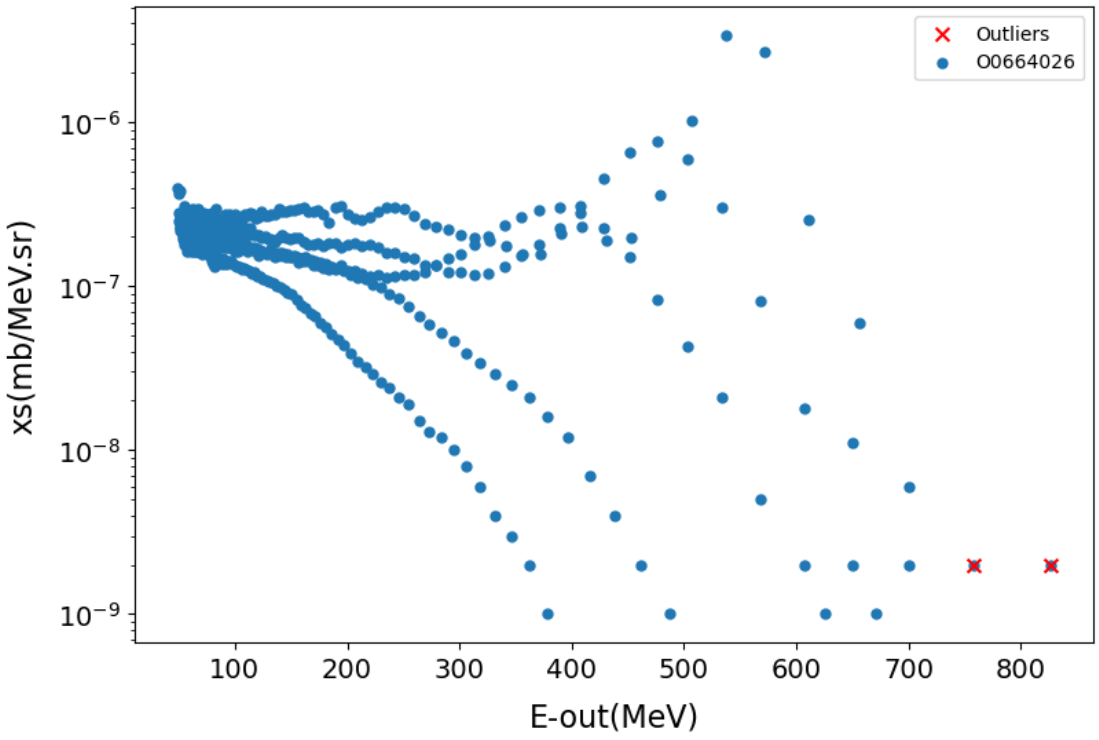

Figure 4.15: Outlier Detection in Group 2 using DBSCAN Method.

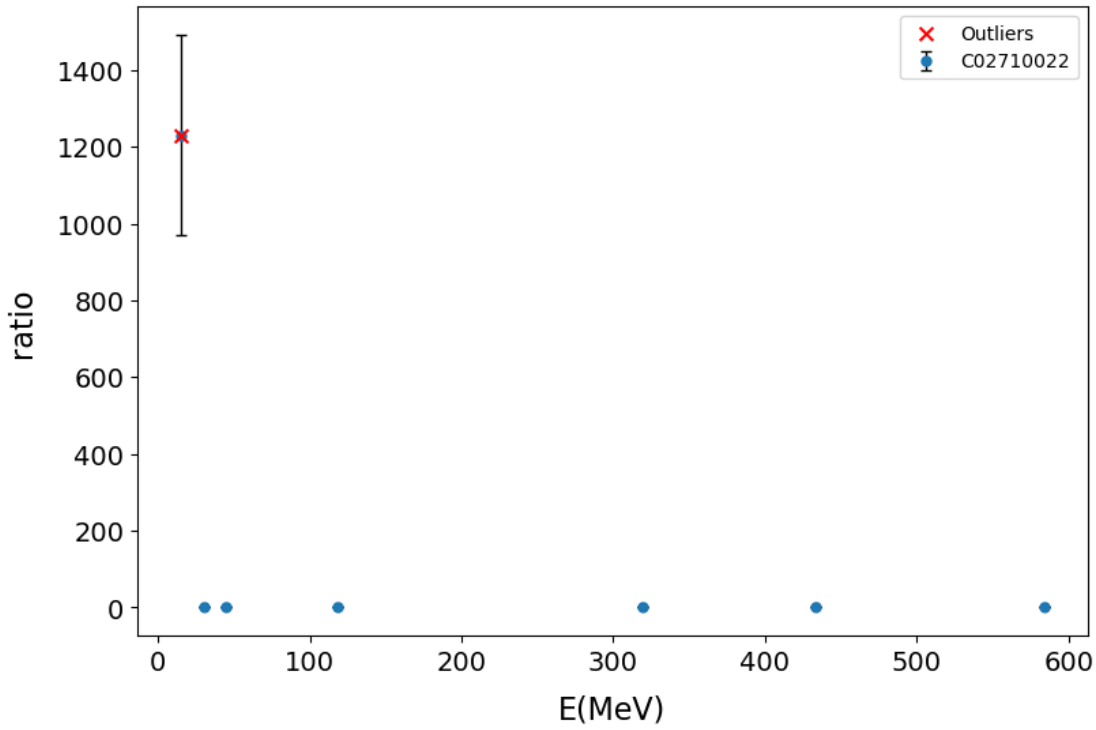

Figure 4.16: Outlier Detection in Group 3 using DBSCAN Method.

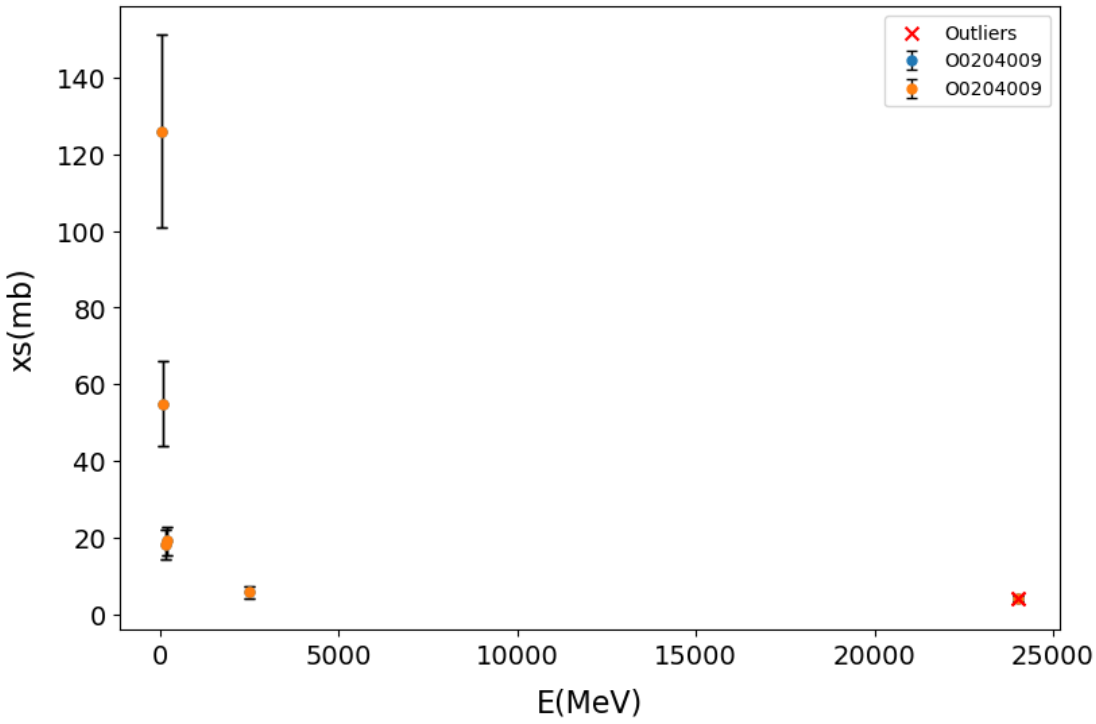

Figure 4.17: Outlier Detection in Group 4.2 using DBSCAN Method.

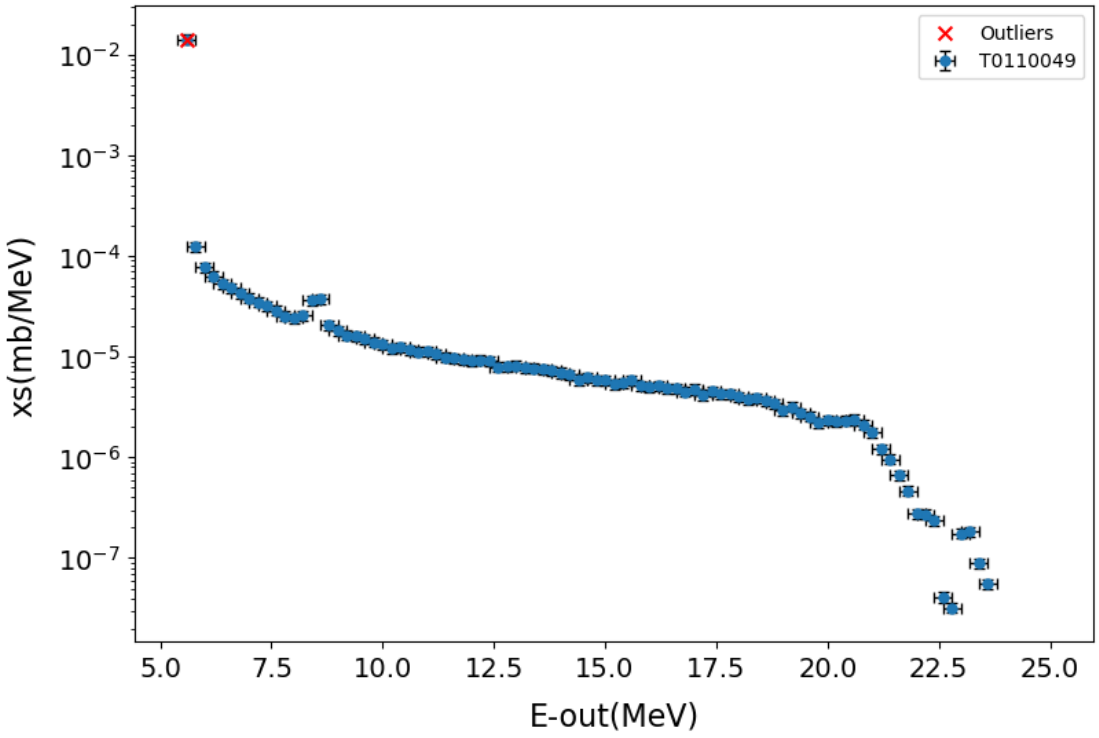

Figure 4.18: Outlier Detection in Group 5 using DBSCAN Method.

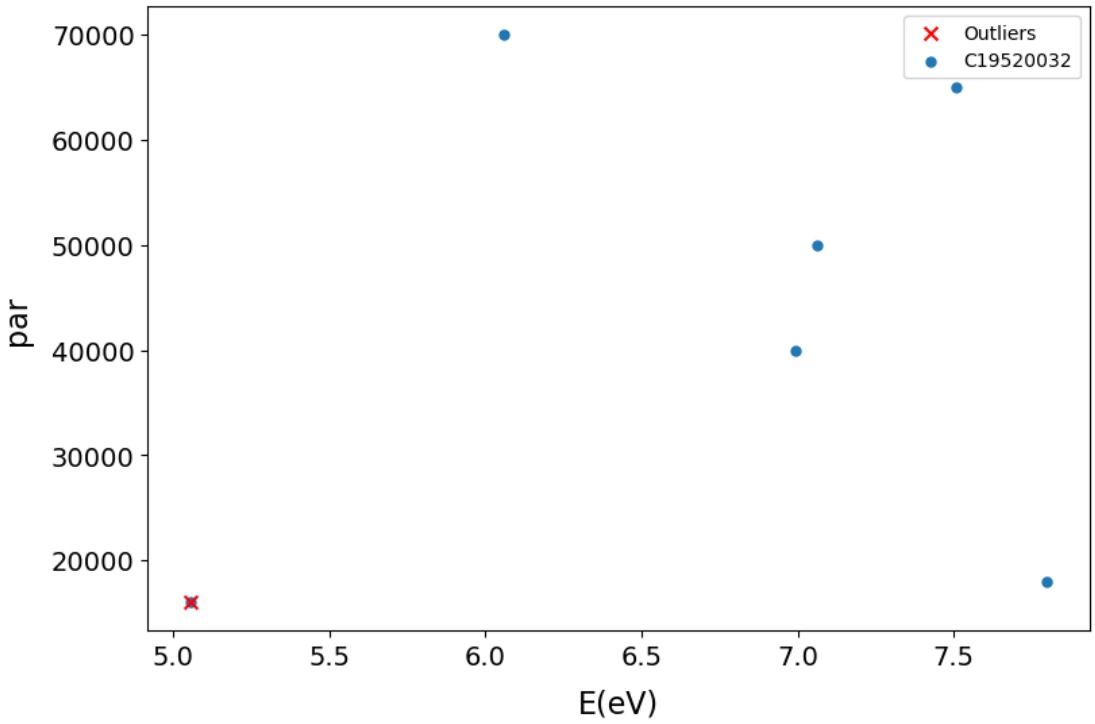

Figure 4.19: Outlier Detection in Group 6 using DBSCAN Method.

Figures 4.14 to 4.19 illustrate the outcomes of applying the DBSCAN algorithm to various data groups. Generally speaking, the results appear promising, although some instances of false positives can be observed, along with some outliers that are debatable. Nevertheless, a portion of the detected points clearly display characteristics consistent with being potential outliers.

4.4.3 Conclusion on DBSCAN Method

DBSCAN is another machine learning-based method for outlier detection that offers a unique approach. Unlike methods that are primarily statistical, DBSCAN considers the density of points in a data space to identify outliers. This can be advantageous as it might better handle data with varying densities.

For the purpose of this study, the default hyperparameters are used in the DBSCAN implementation. While the method is known to be highly sensitive to its hyperparameters, such as the radius of the neighborhood and the minimum number of points needed to form a dense region, these parameters are kept at their default settings as an initial exploration. Detailed explanations for these hyperparameters can be found in Section 3.5. Future work may involve hyperparameter tuning to optimize the method for the specific characteristics of the dataset. However, even with default settings, DBSCAN provides a distinct and insightful approach to outlier detection compared to purely statistical methods.

Moreover, it is important to note that DBSCAN can be computationally intensive, particularly when dealing with large datasets. In this study, for example, the data for Group 1 had to be subdivided into smaller datasets to make the algorithm computationally feasible. This is a limitation to consider, especially when aiming to apply the method on a larger scale.

4.5 Local Outlier Factor (LOF)

4.5.1 Implementation of Local Outlier Factor (LOF) Method

The script for the LOF implementation is accessible in the GitHub repository, named as `OutlierDetection_LOF`. We make use of the `LocalOutlierFactor` class available in the `sklearn.neighbors` package. Below is a segment of code demonstrating how the algorithm is applied:

```
# Scaling the Data
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df_without_id)

# Applying the LOF algorithm
lof = LocalOutlierFactor(n_neighbors=20, contamination=0.01)
outliers = lof.fit_predict(df_scaled)

# Identifying Outliers
is_outlier = outliers == -1

# Reverting the Scaling
df_descaled = pd.DataFrame(scaler.inverse_transform(df_scaled), columns=
    df_without_id.columns)

# Adding the IDs and extracting the outliers
df_descaled["X4_ID"] = X4_ID
df_descaled["is_outlier"] = is_outlier
```

In the implementation of the LOF (Local Outlier Factor) method, the first step involves standardizing the dataset using the `StandardScaler` class from the `sklearn.preprocessing` package. This ensures that each feature has a mean of zero and a standard deviation of one. After scaling, the outlier detection is carried out by utilizing the `LocalOutlierFactor` class. The algorithm is parameterized with `n_neighbors=20`, denoting the number of neighbors to consider for each data point, and a `contamination` factor, which is suitably selected to indicate the expected proportion of outliers. The `fit_predict` method is subsequently invoked to classify each data point as an outlier or inlier, with outliers receiving a label of `-1`. The data is then reverted to its original scale for interpretability. Lastly, identification labels (`X4_ID`) and outlier statuses (`is_outlier`) are appended to the dataframe for visual verification.

4.5.2 Results and Graphical Analysis

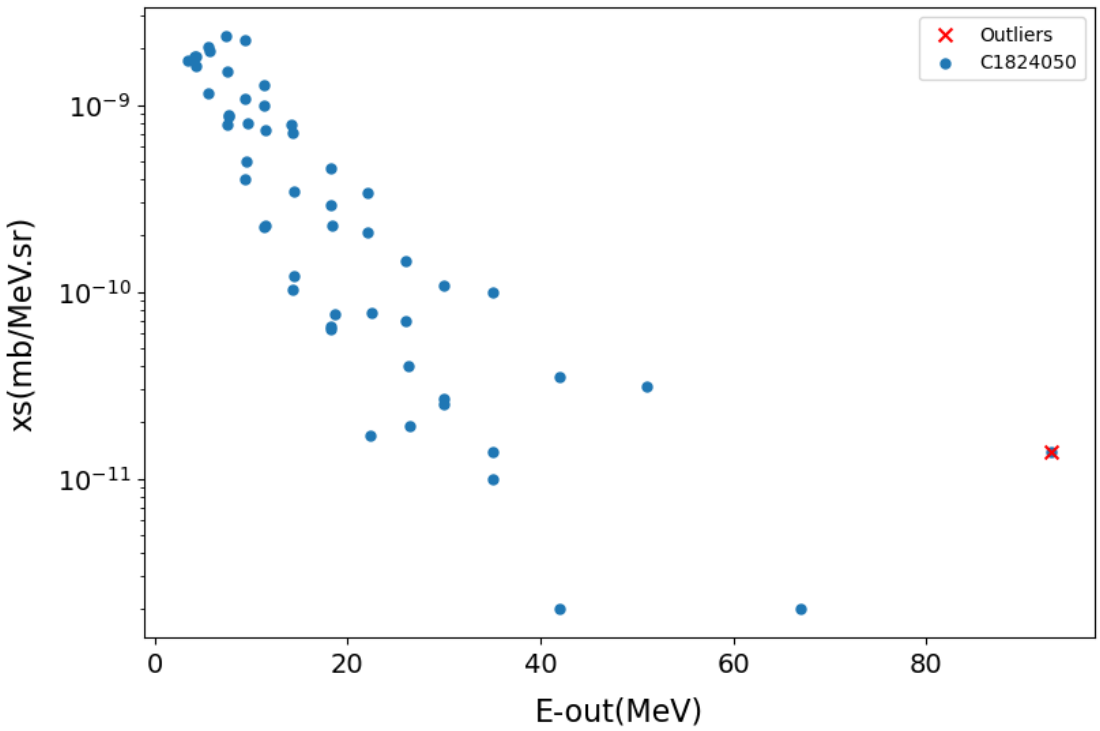

Figure 4.20: Outlier Detection in Group 2 using LOF Method.

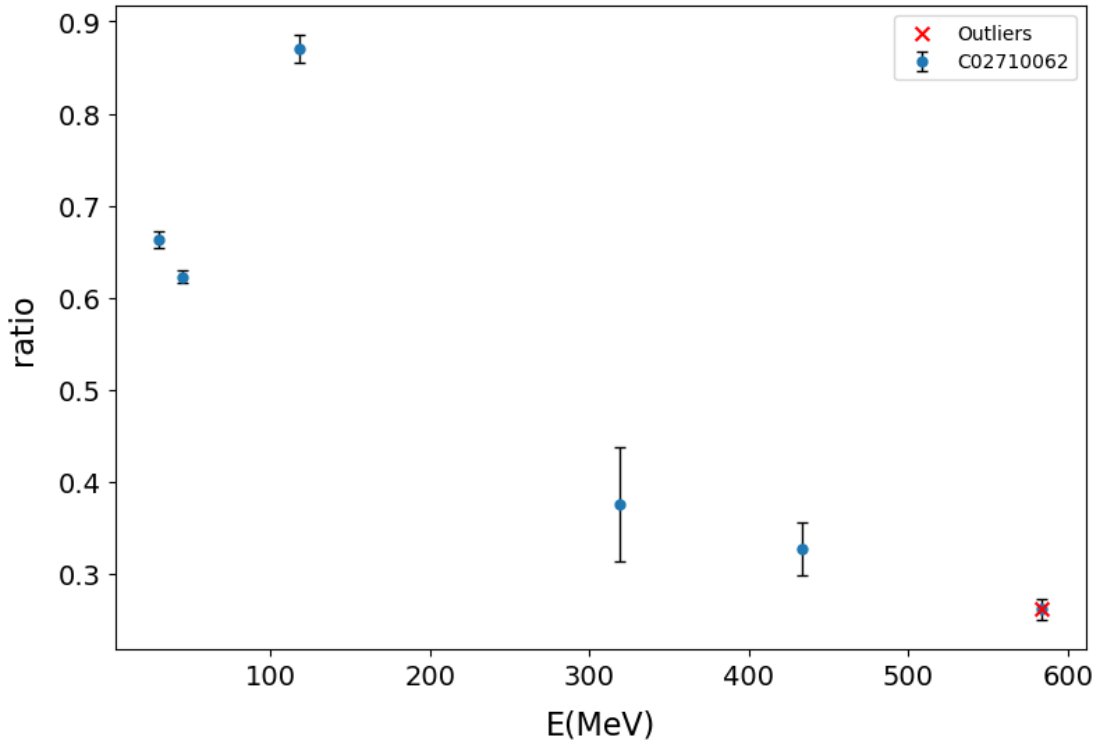

Figure 4.21: Outlier Detection in Group 3 using LOF Method.

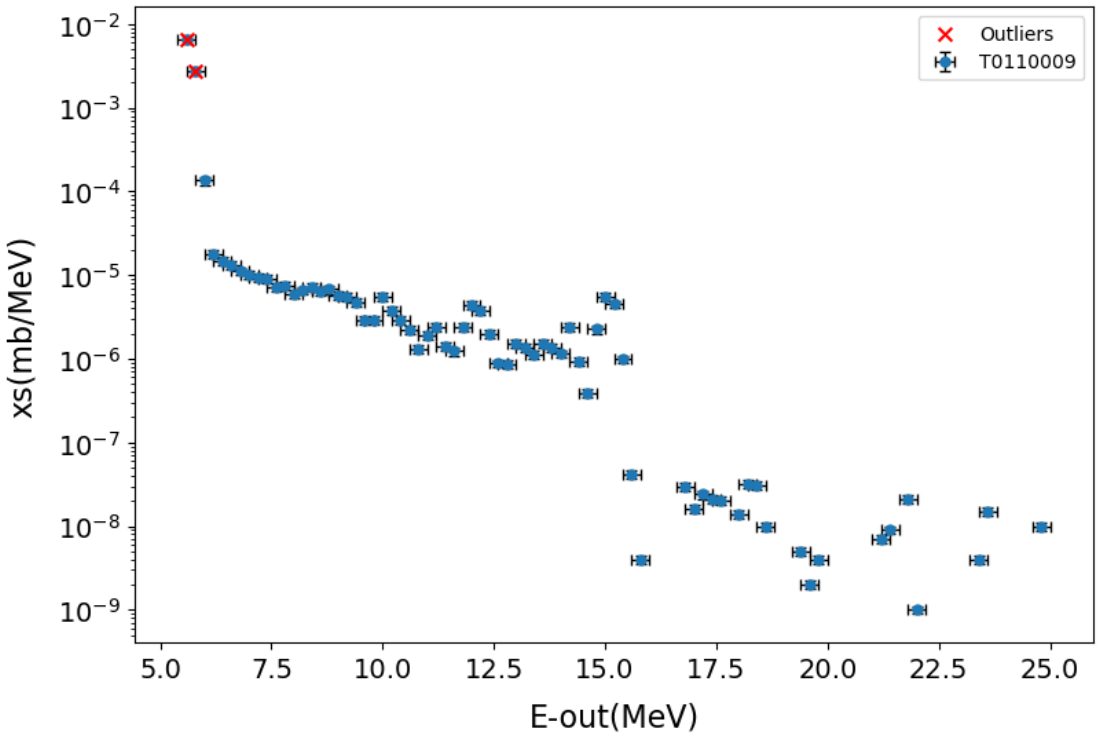

Figure 4.22: Outlier Detection in Group 5 using LOF Method.

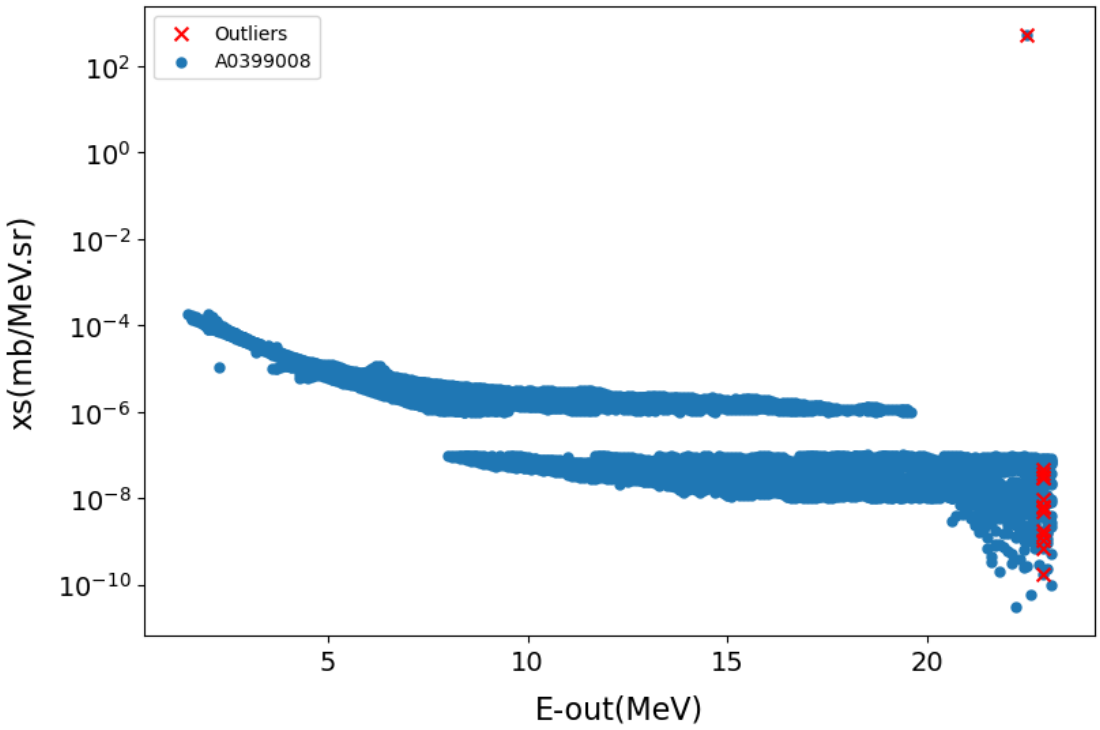

Figure 4.23: Outlier Detection in Group 2 using LOF Method.

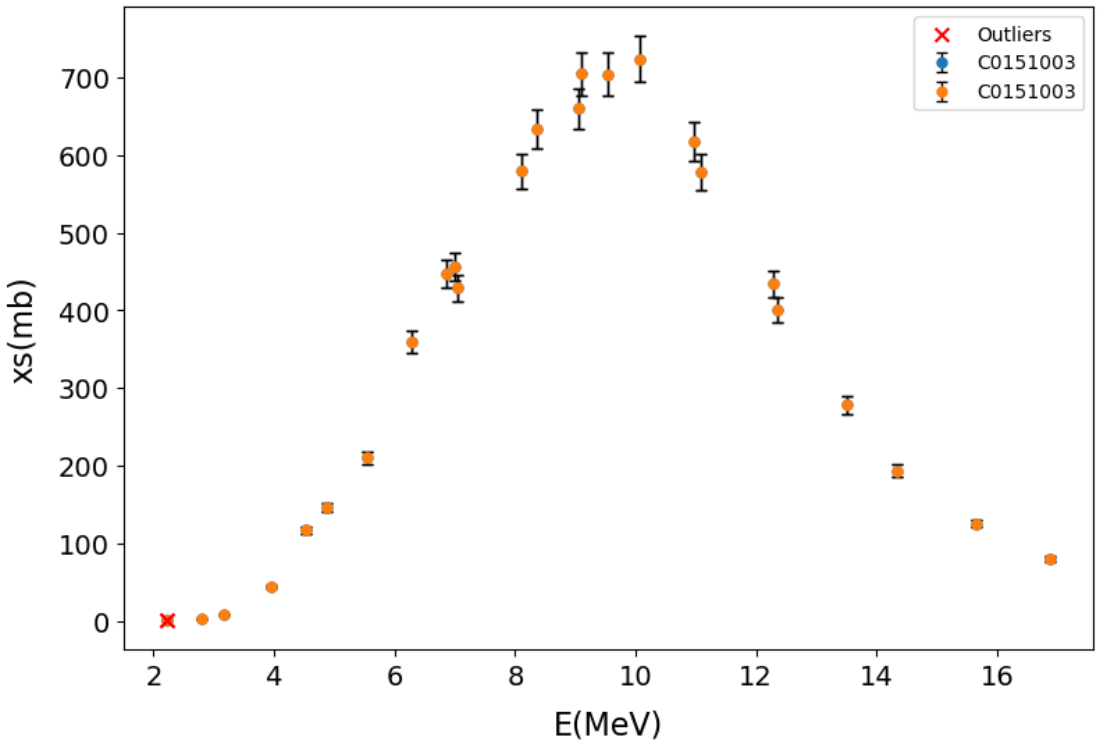

Figure 4.24: Outlier Detection in Group 4.2 using LOF Method.

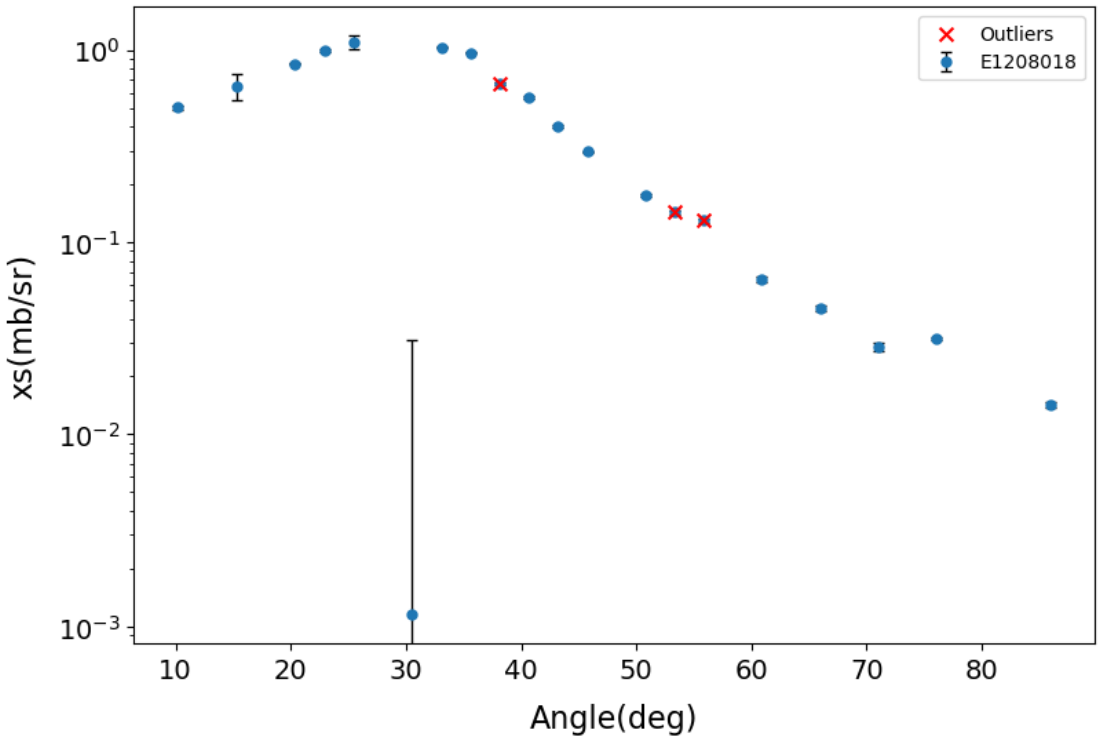

Figure 4.25: Outlier Detection in Group 1 using LOF Method.

The performance of the LOF method exhibits notable variability across different scenarios. In certain instances, such as demonstrated in Figures 4.20 to 4.22, the algorithm adeptly pinpoints data points that could be considered legitimate outliers. However, there are instances where

the detection is less reliable. For example, Figure 4.23 highlights a case where the algorithm successfully identifies a potential outlier but also erroneously flags other points that are more likely to be inliers. In more extreme situations, such as depicted in Figures 4.24 and 4.25, the algorithm falters entirely, failing to correctly identify obvious outliers within the dataset.

4.5.3 Conclusion on LOF Method

The Local Outlier Factor (LOF) method is lauded for its adaptability in handling various types of data distributions. However, it also grapples with challenges common to many outlier detection algorithms. One such issue is its sensitivity to the specific characteristics of the dataset. This becomes particularly evident when the dataset lacks adequate data points in certain ranges, thereby increasing the likelihood of identifying false positives.

Moreover, the necessity for fine-tuning hyperparameters cannot be understated. The effectiveness of LOF in outlier detection is significantly influenced by the choice of hyperparameters, such as the number of neighbors and the contamination factor. Unfortunately, these settings may require different configurations for each group of data, making the process somewhat time-consuming to arrive at satisfactory results.

Despite these challenges, LOF exhibits strengths where other methods may falter. It is particularly robust in avoiding false positives in datasets that feature sharp peaks or clusters. Many algorithms struggle in these areas and tend to misclassify inliers as outliers, whereas LOF demonstrates superior performance in such conditions.

4.6 Support Vector Machines (SVM)

4.6.1 Implementation of SVM Method

The script for the SVM-based outlier detection is available in the GitHub repository, named as `OutlierDetection_SVM_Method.ipynb`. For this approach, we leverage the `OneClassSVM` class from the `sklearn.svm` package. The corresponding segment of the code detailing the algorithm's application is presented below:

```
# Scaling the Data
scaler = StandardScaler()
scaled_df = scaler.fit_transform(df_without_id)

# Applying OneClassSVM
ocsvm = OneClassSVM(kernel="rbf", nu=0.01)
ocsvm.fit(scaled_df)

# Identifying Outliers
pred = ocsvm.predict(scaled_df)
outliers = (pred == -1)

# Reverting the Scaling
descaled_df = scaler.inverse_transform(scaled_df)

# Adding the IDs and extracting the outliers
result_df = pd.DataFrame(descaled_df, columns=df_without_id.columns)
result_df["X4_ID"] = x4_id_column
```

```
result_df["outliers"] = outliers
outliers_df = result_df[result_df["outliers"] == 1].iloc[:, :-1]
print("Percentage of outliers: {:.2f}%".format(len(outliers_df)/len(df)
    *100))
outliers_df
```

In the SVM-based outlier detection methodology, the initial phase entails standardizing the dataset using the `StandardScaler` class from the `sklearn.preprocessing` package. By doing so, every feature is normalized to have zero mean and unit variance. The `OneClassSVM` class is then applied, with the kernel set to `rbf` (Radial Basis Function). The `nu` parameter is set to its default value, which serves as both an upper bound on the fraction of margin errors and a lower bound on the fraction of support vectors. Once the SVM is trained, predictions are made on the scaled dataset to identify outliers, which are labeled as `-1`. After identifying outliers, the dataset is transformed back to its original scale. Lastly, ID labels (`X4_ID`) and outlier indicators are appended to the dataframe, which is then filtered to extract the detected outliers for further analysis.

4.6.2 Results and Graphical Analysis

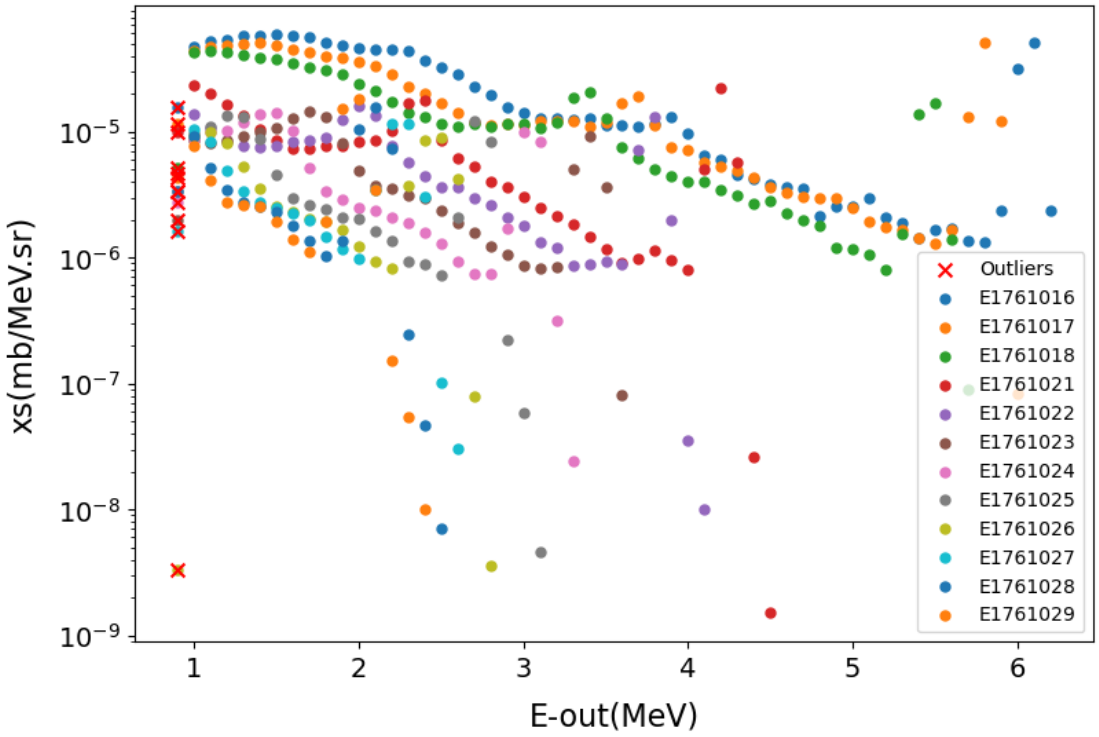

Figure 4.26: Outlier Detection in Group 2 using SVM Method.

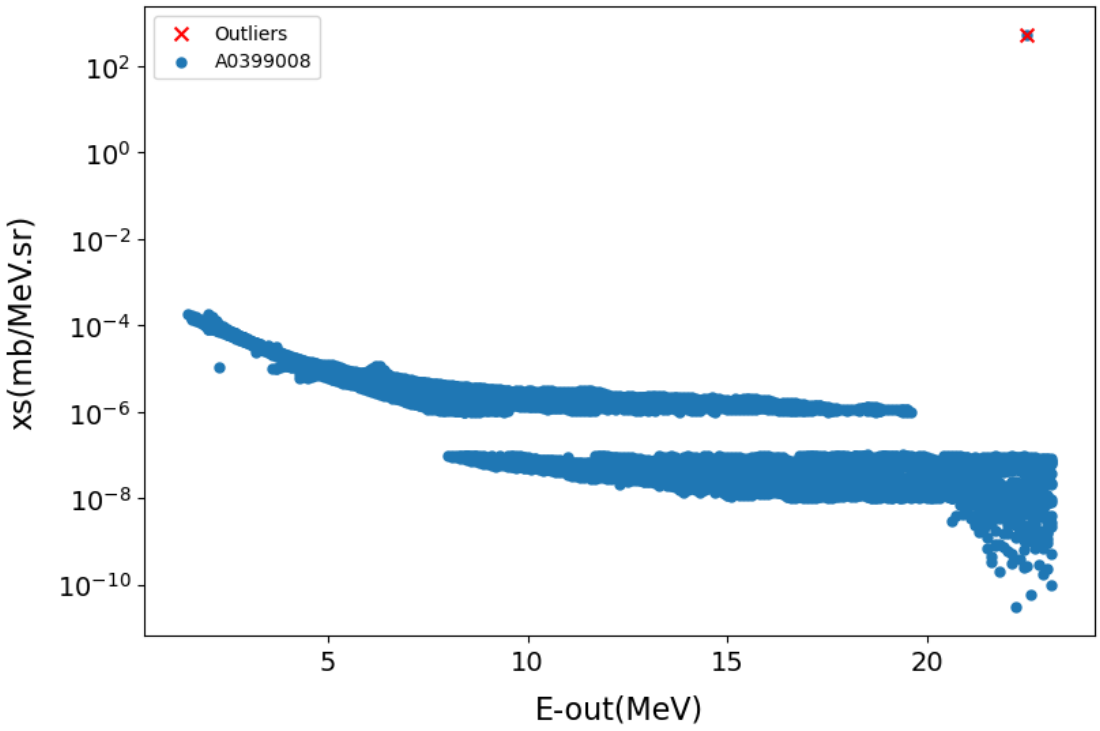

Figure 4.27: Outlier Detection in Group 2 using SVM Method.

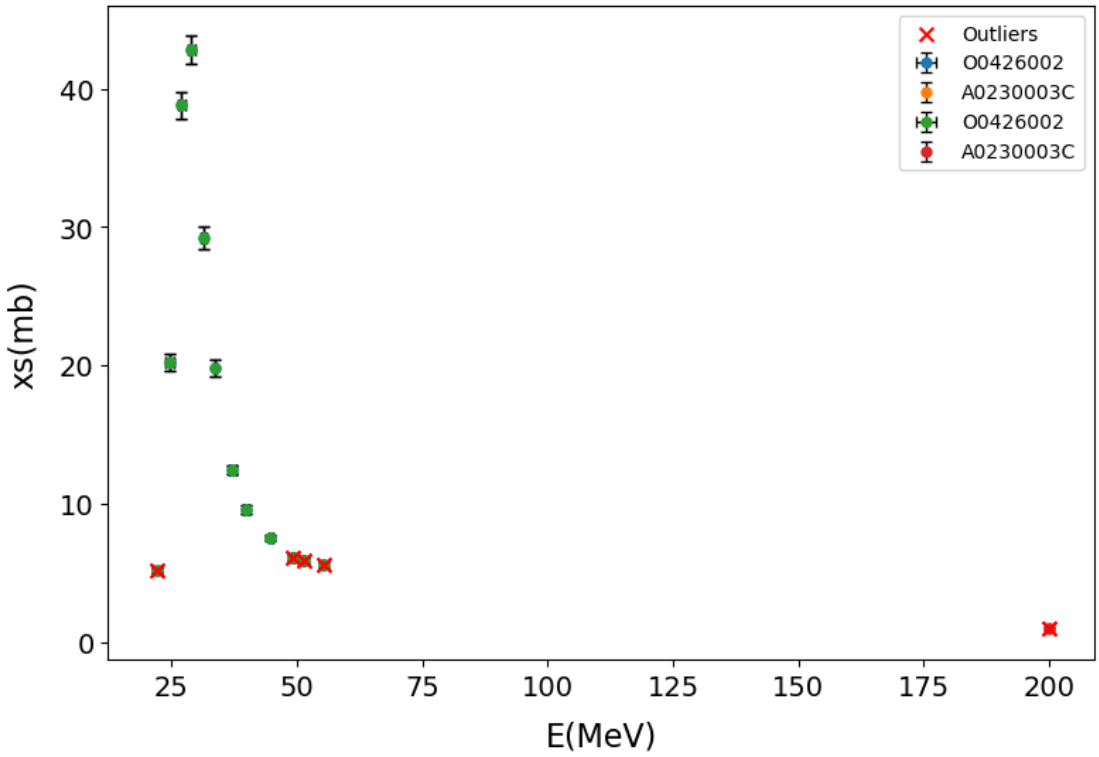

Figure 4.28: Outlier Detection in Group 4.1 using SVM Method.

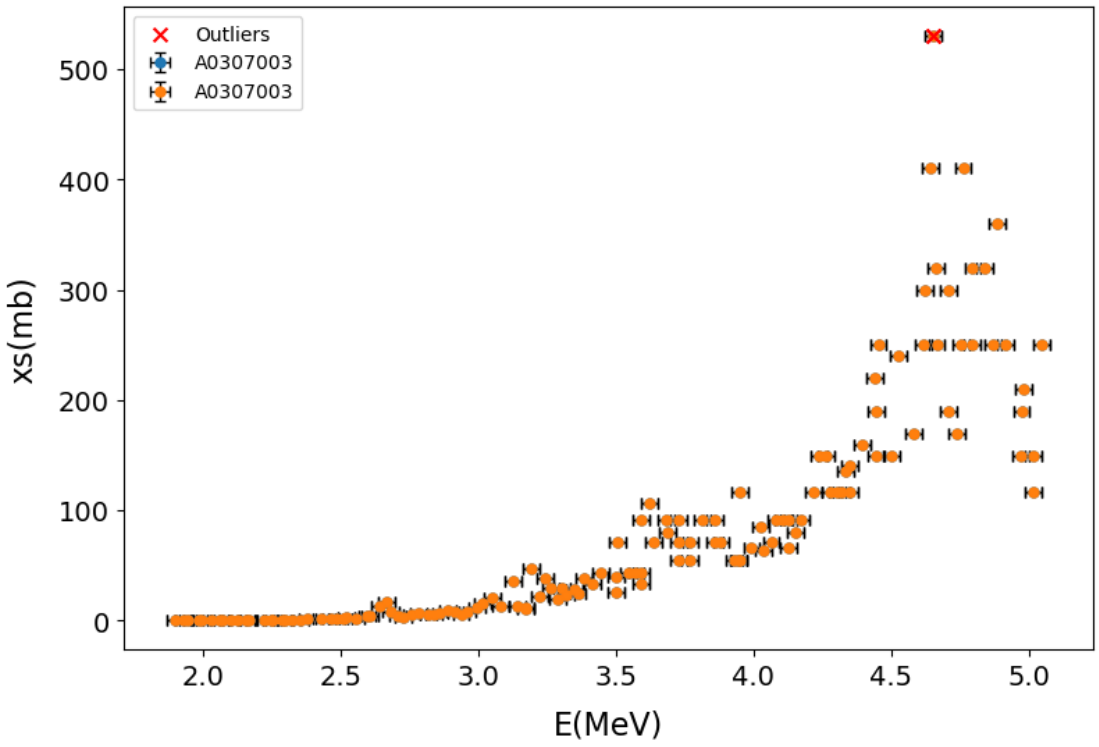

Figure 4.29: Outlier Detection in Group 4.2 using SVM Method.

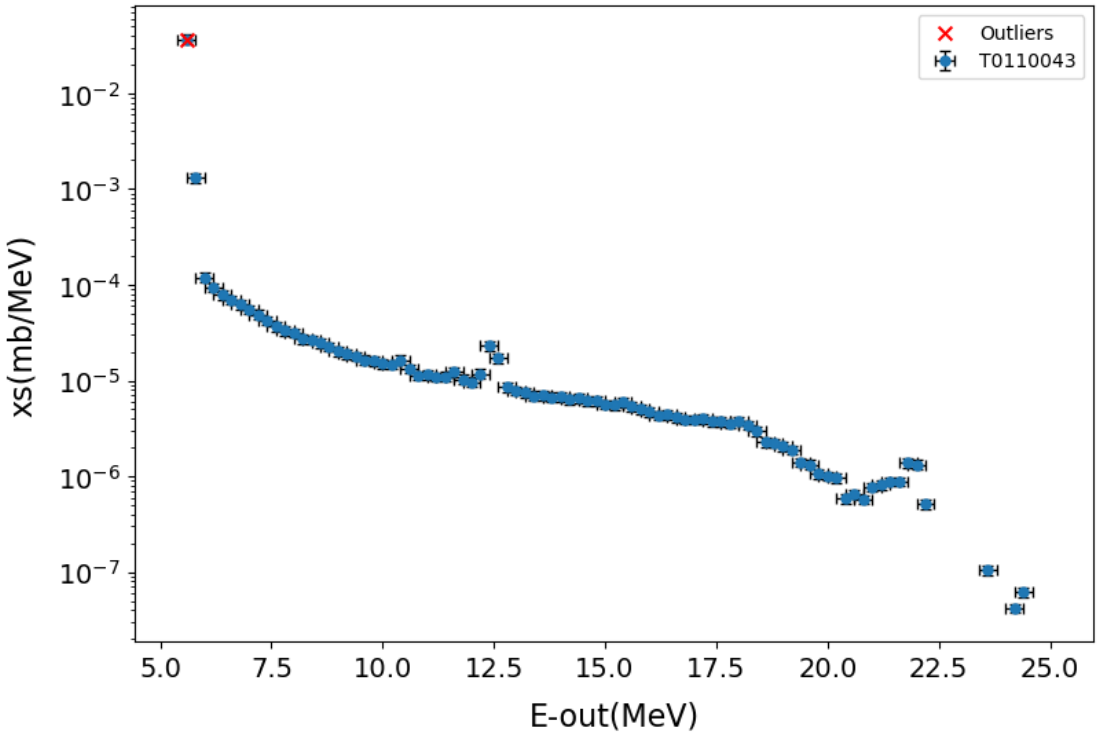

Figure 4.30: Outlier Detection in Group 5 using SVM Method.

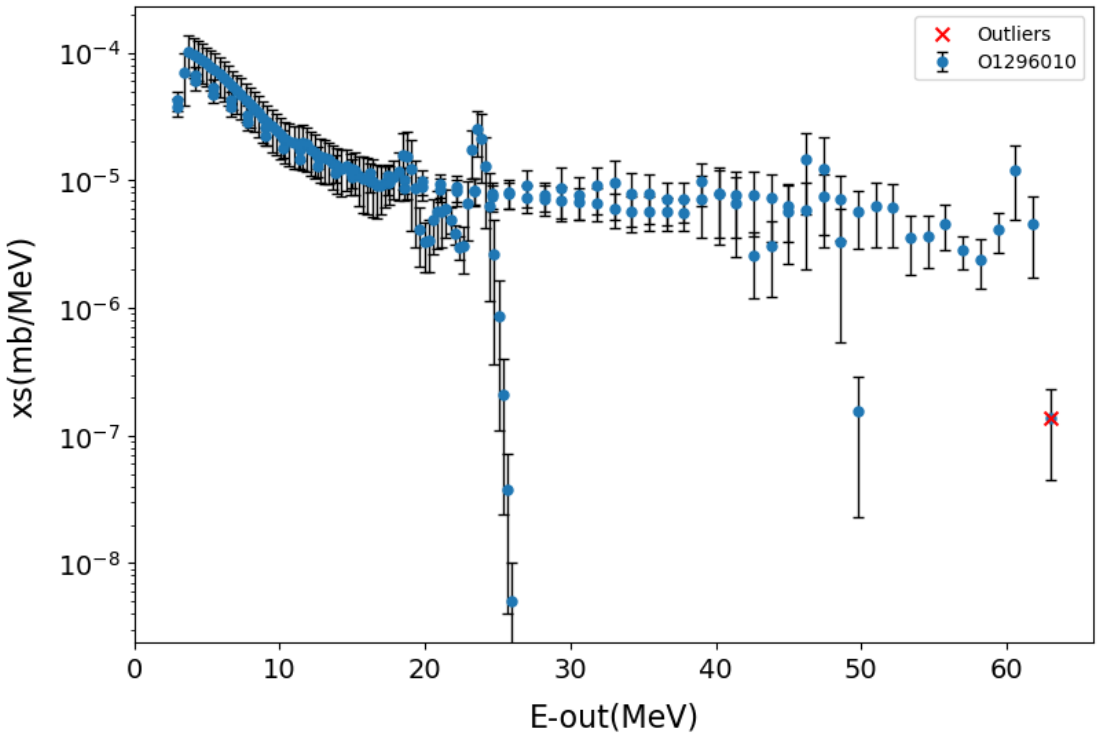

Figure 4.31: Outlier Detection in Group 5 using SVM Method.

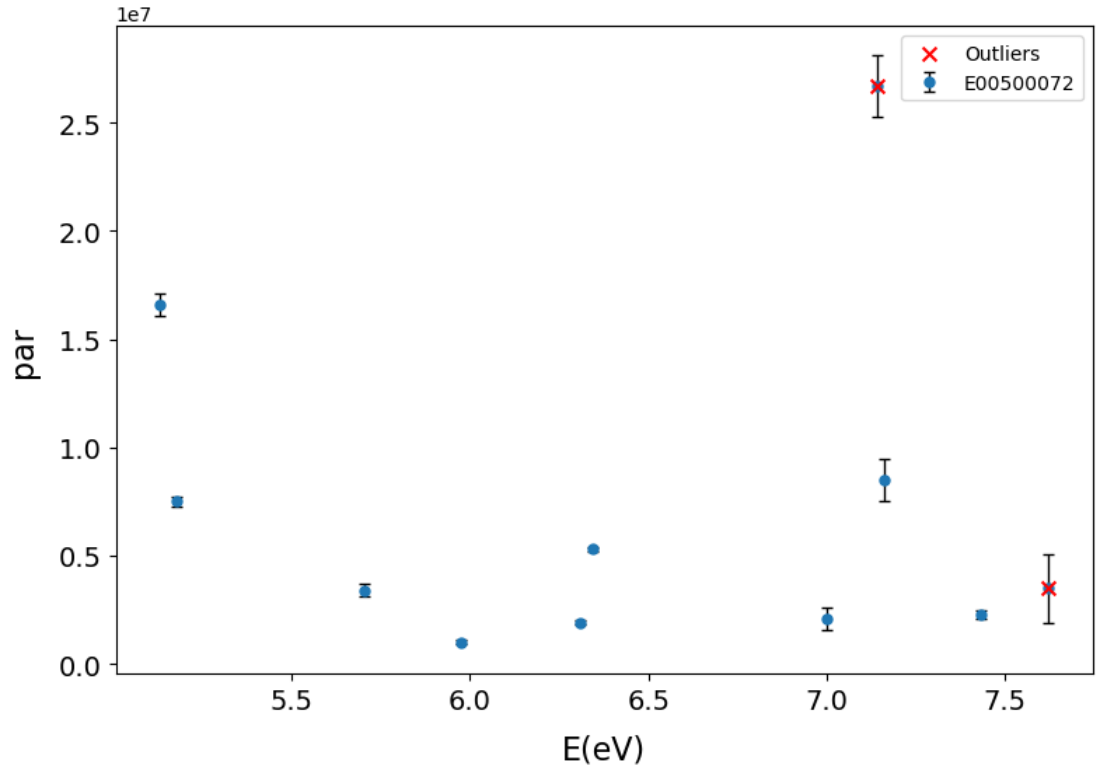

Figure 4.32: Outlier Detection in Group 6 using SVM Method.

Figures 4.26 to 4.32 show the results of using the SVM method for outlier detection. Overall, the method performs well across all data groups. While some false positives do appear, which

is common for these types of algorithms, SVM does a good job of identifying likely outliers in each dataset.

4.6.3 Conclusion on SVM Method

The SVM algorithm proved effective in detecting outliers across all our data groups. The use of default parameters serves as a reasonable starting point, demonstrating the algorithm's robustness across diverse scenarios. While these default settings are well-calibrated for a broad range of problems, they can be further fine-tuned either through domain-specific knowledge or hyperparameter optimization techniques.

Despite its overall effectiveness, the algorithm is computationally intensive, particularly for large datasets. This limitation was evident in our study where we had to divide the first data group into smaller subsets to manage the computational requirements. Thus, while SVM is powerful, its scalability can pose challenges for larger datasets.

4.7 Neural Network Models: Autoencoders

While previous subsections focused on machine learning algorithms, we now turn to a neural network-based methodology—specifically, autoencoders implemented using the Keras library. Due to the inherent complexities associated with neural networks, this section is subdivided to cover implementation details, evaluation strategies, and a specialized use-case for outlier detection. This structured approach aims to provide a comprehensive overview from architectural design to practical applications.

4.7.1 Implementation of the Autoencoder

Data Preprocessing:

```
# Data Splitting (Train/Test)
X_train, X_test = train_test_split(df, test_size=0.2, random_state=42)
# Data Scaling
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Firstly, the dataset is split into a training and test set using scikit-learn's `train_test_split` function, reserving 20% of the data for testing. The `random_state=42` is set for reproducibility. Then, the `MinMaxScaler` is applied to scale the features between 0 and 1, aiding in model convergence.

Model Architecture:

```
# Model Architecture
input_dim = X_train.shape[1]
encoding_dim = int(input_dim / 2)
input_layer = Input(shape=(input_dim,))
encoder = Dense(encoding_dim, activation="relu")(input_layer)
decoder = Dense(input_dim, activation="sigmoid")(encoder)
```

The architecture of the autoencoder is simple but effective, consisting of an input layer, an encoder layer, and a decoder layer. The encoder uses a `ReLU` activation function, and the decoder uses a `sigmoid` activation function. The encoding dimension is set to half of the input dimension, a parameter that can be adjusted depending on specific needs.

Model Compilation and Training:

```
# Model Compilation
autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss="
    mean_squared_error")
# Model Training
history = autoencoder.fit(X_train, X_train, epochs=50, batch_size=256,
    validation_data=(X_test, X_test))
```

The model is compiled using the `Adam` optimizer with a learning rate of 0.001 and Mean Squared Error as the loss function. Finally, the model is trained for 50 epochs with a batch size of 256, using the test data for validation during training.

4.7.2 Model Evaluation

For the evaluation of our autoencoder model, we rely on a combination of visual inspection of the training and validation loss curves, as well as quantitative metrics, including the Mean Squared Error (MSE).

Visual Inspection of Loss:

```
# Loss Plot
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.legend()
plt.yscale("log")
plt.show()
```

The loss curves for both the training and validation sets are illustrated in Figure 4.33. A logarithmic scale is employed to enhance visualization clarity. A downward trend across epochs typically signifies effective model learning. For data group 1, as shown in the figure, the loss curve stabilizes and flattens, indicating that the chosen number of epochs is sufficient for model convergence.

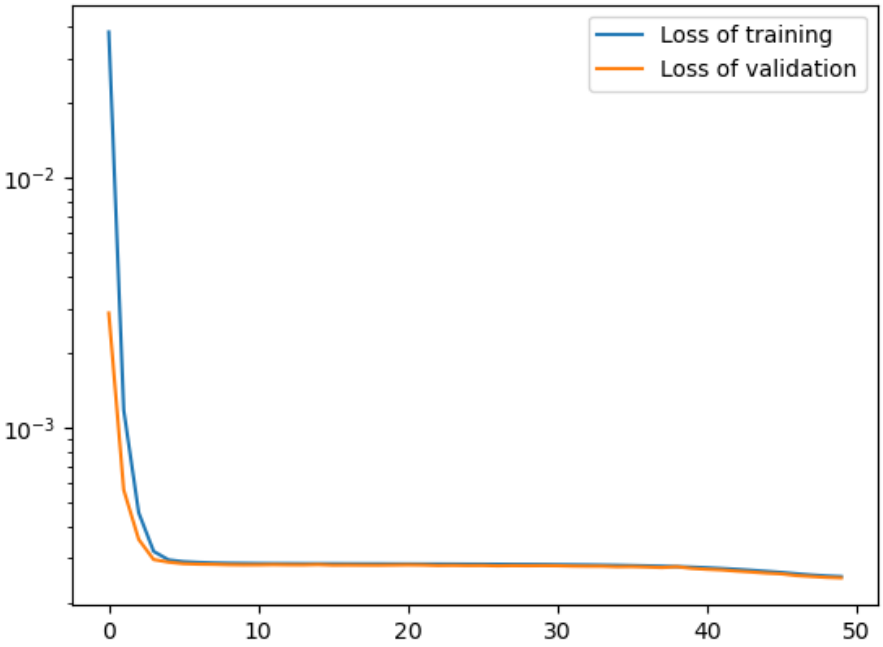

Figure 4.33: Training and Validation Loss Curves for Group 1 data.

Immediately following the visualization, we examine several performance metrics:

```
# Performance Metrics
predictions = autoencoder.predict(X_test)
mse = np.mean(np.power(X_test - predictions, 2), axis=1)
print(f"MSE: {np.mean(mse)}")

mean_value = np.mean(X_train, axis=0)
mse_naive = np.mean((X_test - mean_value) ** 2)
print(f"MSE of naive model: {mse_naive}")

std_value = np.std(X_train, axis=0)
print(f"Standard deviation: {std_value}")
```

Interpreting Metrics: To better comprehend the MSE value, consider these approaches:

- *Comparison to Naive Model:* A low MSE value could be a good sign, but it's informative to compare it with a naive model that always predicts the mean of the training set. A lower MSE for the autoencoder compared to the naive model would highlight its relative effectiveness.
- *Comparison to Data Variability:* Furthermore, comparing the MSE to the standard deviation of the features can offer insights. If the MSE is significantly lower than the standard deviation, the model is likely doing a good job in capturing the underlying patterns in the data.

By examining both the visual loss curves and these quantitative metrics, a comprehensive assessment of the model's performance can be attained.

4.7.3 Outlier Detection Using Autoencoder

After evaluating the model's performance, we utilize it for outlier detection. The primary objective is to identify data points that deviate significantly from the normal data distribution.

Data Preparation:

```
# Preparing the DataFrame for Outlier Detection
df = pd.read_csv(path)
df = clean_dataframe(df)
X4_ID_column = df["X4_ID"].copy()
df_without_X4_ID = df.drop("X4_ID", axis=1)
```

First, we prepare the dataframe by reading the data and applying any preprocessing steps such as cleaning. We preserve the `X4_ID` column for later use and remove it before scaling and making predictions.

Scaling and Predicting:

```
# Scaling and Predictions
df_scaled_complete = scaler.transform(df_without_X4_ID)
predictions = autoencoder.predict(df_scaled_complete)
```

The model is then applied to the scaled data to generate the reconstructed values.

Identifying Anomalies:

```
# Detecting Outliers
reconstruction_error = np.mean(np.power(df_scaled_complete - predictions
    , 2), axis=1)
threshold = np.percentile(reconstruction_error, 99.0)
anomalies_col2 = reconstruction_error > threshold
```

The reconstruction error, calculated as the mean squared error between the original and reconstructed data, serves as the metric for identifying outliers. We then select a threshold based on the 99th percentile of the reconstruction error to flag anomalies.

Post-Processing:

```
# Post-processing
df_original_values = pd.DataFrame(scaler.inverse_transform(
    df_scaled_complete), columns=df_without_X4_ID.columns, index=df.index
    )
df_original_values["X4_ID"] = X4_ID_column
df_original_values["Outliers"] = anomalies_col2

# Extracting the Outliers
outliers_df = df_original_values[df_original_values["Outliers"] == True
    ].drop("Outliers", axis=1)
```

Finally, the dataframe is post-processed to extract the outliers based on the flag set earlier. We also include the `X4_ID` column and calculate the percentage of detected outliers.

The autoencoder's architecture inherently learns to compress the most important features of the data. When it attempts to reconstruct an outlier, the reconstruction error tends to be high, making the model an effective tool for outlier detection.

4.7.4 Results and Graphical Analysis

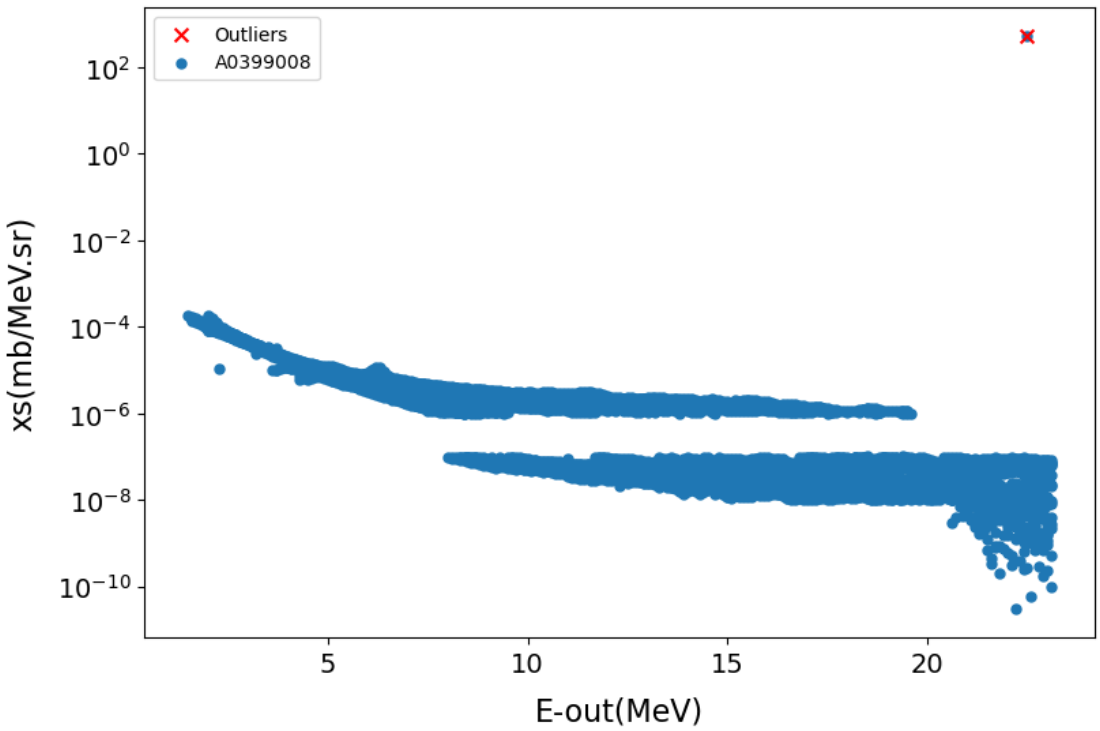

Figure 4.34: Outlier Detection in Group 2 using Autoencoder Method.

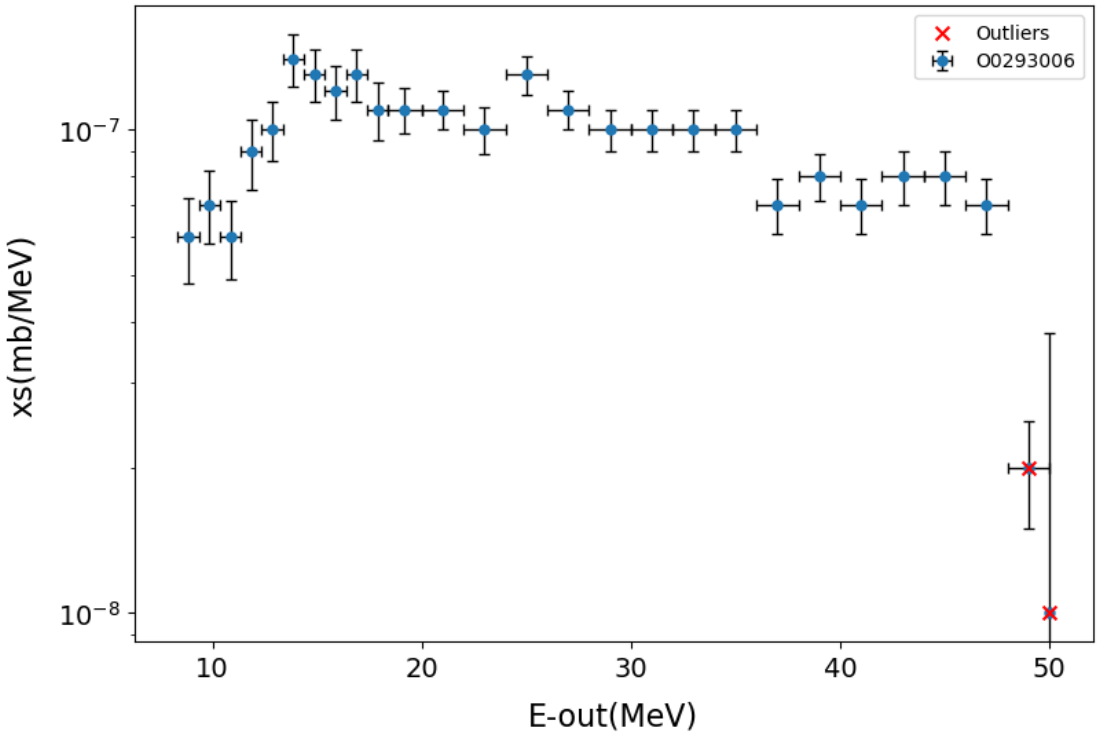

Figure 4.35: Outlier Detection in Group 5 using Autoencoder Method.

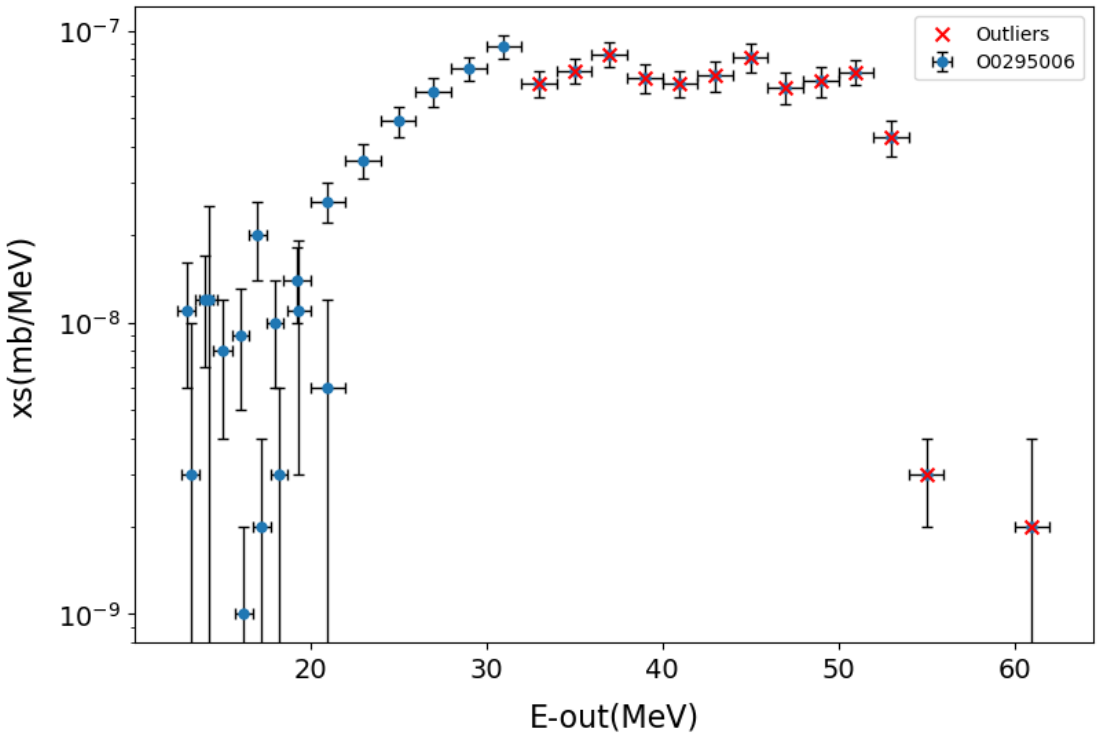

Figure 4.36: Outlier Detection in Group 5 using Autoencoder Method.

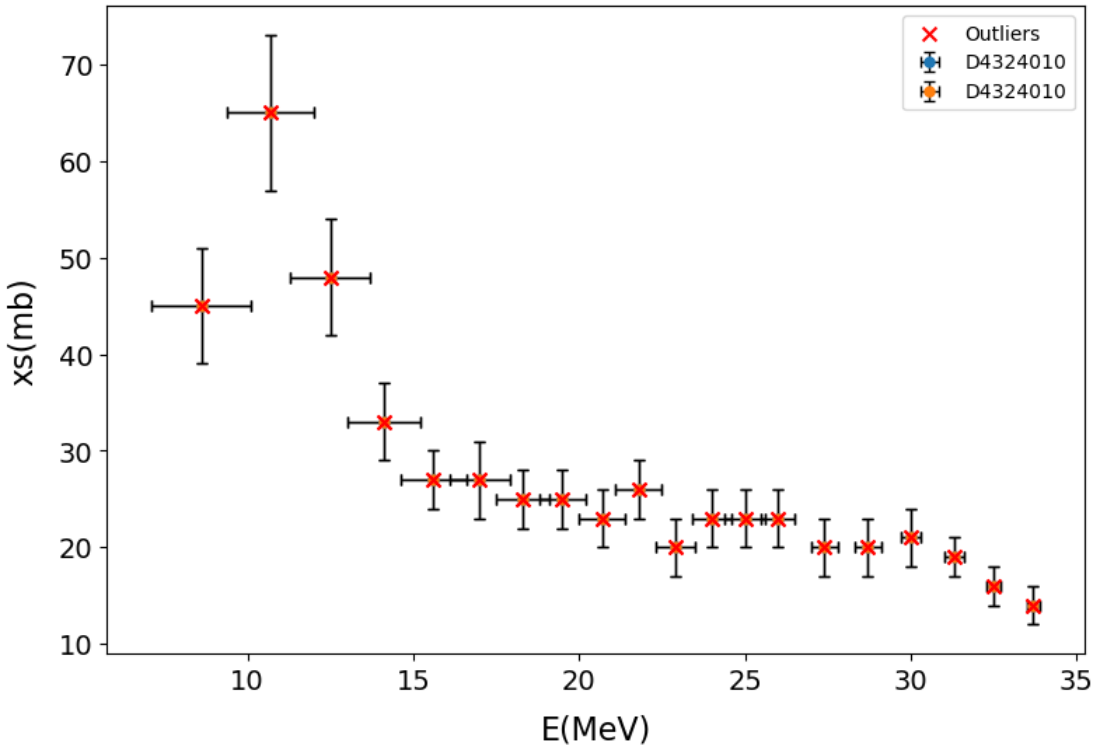

Figure 4.37: Outlier Detection in Group 4.2 using Autoencoder Method.

The results of the outlier detection present varied outcomes as evidenced by the figures. Some instances appear to be potential outliers, as shown in Figures 4.34 and 4.35. However, in other cases, the detection includes false positives, identifying inliers as outliers, such as in Figure 4.36.

Additionally, certain experiments are entirely flagged as outliers despite having a substantial number of data points, as illustrated in Figure 4.37.

4.7.5 Conclusions on Autoencoder Method

In contrast to traditional machine learning methodologies explored earlier in this thesis, the utilization of autoencoders, a neural network-based approach, necessitates a deeper level of understanding for effective application. Despite this complexity, the potential benefits of employing neural networks are significant. Future work could extend to optimizing various parameters such as the number of layers, activation functions, and learning rates.

Regarding the performance of the autoencoder in outlier detection, the results are promising but not without challenges. While the model successfully identified several potential outliers, it also produced a lower rate of false positives compared to other methods. However, it is important to note that the efficacy of the model is heavily reliant on the selected parameters.

One critical consideration is the volume of data available for training. The data groups used for training the autoencoder were identical to those used in traditional machine learning techniques, and in some cases, these groups had limited data points. This scarcity can adversely affect the model's performance. An alternative strategy could be to use the entire proton reaction database without specific categorization into groups. While this approach could potentially eliminate the need for data classification, thereby saving time, there is no guarantee of improved performance. Since classification is required for the machine learning methods, maintaining a consistent approach across methodologies serves as a more effective basis for comparison.

5 CONCLUSIONS

In this chapter, we consolidate the conclusions that arise from our thorough research, scripting, and data analysis efforts. The insights gained from the analyses in Chapter 4 are critical in shaping the conclusions we reach. This chapter is organized into three primary sections: Summary of Findings, Future Work, and Final Conclusions.

Each section delves into the specific conclusions reached from the utilization of various outlier detection techniques applied to the EXFOR nuclear data library. The methods employed in this thesis range from traditional statistical approaches such as the Interquartile Range (IQR) to more advanced machine learning algorithms, including Isolation Forest, DBSCAN, Local Outlier Factor (LOF), Support Vector Machine (SVM), and Autoencoders. Each methodology contributes valuable insights into the intricacies of outlier detection in complex scientific datasets.

Concluding this chapter is a section titled 'Final Conclusions.' Within this section, overarching conclusions about the entirety of the research conducted in this thesis are articulated. This final section provides a comprehensive summation of the work and signifies the completion of this thesis.

5.1 Summary of Findings

The preprocessing step is vital for the effective application of machine learning techniques. In this study, we opted to classify EXFOR proton reaction data into eight distinct groups based on the logical criteria of data result similarity. It's important to note that these groups vary in the number of data points they contain. Generally, machine learning algorithms perform better with larger datasets, which posed a limitation for groups like 3, 6, 7, 8, and to some extent, 5, as they contain fewer data points. Alternate classification schemes that allow for larger, more robust data sets could potentially yield improved results, especially when deploying neural networks, which are better suited for handling complex data.

Machine learning algorithms typically partition the dataset into a training set and a test set. This partitioning is often performed randomly, resulting in variable outcomes with each script execution. Although the impact of this randomization should ideally be minimal, it is strongly influenced by the type and quantity of data at hand. To reproduce the results presented in this thesis, the `random_state` hyperparameter in the machine learning models should be set to 42. For varied results, which could be better, worse, or comparable, one can either remove the `random_state` parameter or modify its value.

It is crucial to understand that machine learning algorithms are typically designed to identify a predetermined percentage of outliers in a dataset, which does not necessarily correspond to the actual number of outliers present. This leads to an inevitable occurrence of false positives. Hence, the expertise of a domain specialist is indispensable for the thorough interpretation of algorithmic results. While machine learning can flag potential outliers, a human expert ultimately needs to evaluate these flagged data points to distinguish between legitimate outliers, false positives, and genuine but anomalous data points. Given the sheer volume of data in the EXFOR library, manual analysis would be unfeasible, highlighting the invaluable role of machine learning techniques in this study. This work contributes significantly to the field by providing a freely accessible, user-friendly tool for data review, available to anyone with a computer equipped with Python.

To finish this section, we summarize the key findings and limitations of each outlier detection method applied in this study to provide a consolidated view of their performance and applicability.

- **Interquartile Range Method:** The IQR method stands out for its statistical foundation and its capability to incorporate uncertainties, potentially reducing false positives. However, it demonstrated limitations when applied to our complex data sets. Specifically, the IQR method is more suited for uniformly distributed data, which contrasts with the varied distributions present in the EXFOR library. The method struggled with identifying peaks in the data as part of the distribution rather than as outliers. Hence, while IQR has its merits, it is not the most compatible technique for outlier detection in the specialized context of this study.
- **Isolation Forest Method:** This method generally outperformed traditional statistical methods like the IQR, highlighting its effectiveness in detecting outliers. However, the algorithm's sensitivity to the distribution of the dataset revealed some drawbacks. Specifically, the method struggles with sparse regions in the data, often misidentifying isolated data points as outliers when they are simply under-sampled regions. This tendency towards false positive detection limits the method's applicability for our specific use-case.
- **DBSCAN Method:** The DBSCAN stands out for its density-based approach to outlier detection, offering a unique angle compared to traditional statistical methods. While the algorithm is known to be sensitive to hyperparameter settings, even its default configuration yielded insightful results in this study, making it a valuable addition to our outlier detection toolbox. However, DBSCAN does come with computational challenges. Specifically, the method proved to be resource-intensive for larger datasets, requiring us to subdivide our data to make the algorithm computationally feasible. This poses a scalability issue that needs to be addressed for broader applications.
- **Local Outlier Factor Method:** The LOF method shines in its adaptability, handling a variety of data distributions more robustly than some other methods. Particularly, it performs well in datasets with sharp peaks or clusters, avoiding the pitfall of misclassifying inliers as outliers. However, LOF is not without its challenges. The method is sensitive to dataset characteristics and demands careful hyperparameter tuning. In specific cases, different groupings of data may necessitate individualized hyperparameter settings, which adds to the time and effort required for effective implementation.
- **Support Vector Machine Method:** TThe SVM algorithm stood out for its effectiveness in outlier detection across various data groups, even when using default parameters. This speaks to its robustness and potential for fine-tuning through either domain-specific expertise or hyperparameter optimization techniques. However, the algorithm's computational intensity is a notable limitation, especially when dealing with large datasets. This was demonstrated in our study, as the first data group had to be split into smaller subsets to meet computational demands, highlighting challenges in scalability.
- **Autoencoder Method:** Utilizing autoencoders, a form of neural networks, presented a unique set of advantages and challenges. Despite the complexity associated with neural network-based approaches, the autoencoder's performance was promising, yielding a lower rate of false positives compared to traditional techniques. Nevertheless, the model's efficacy was sensitive to parameter selection. The study also revealed that data scarcity in certain groups negatively impacted the model's performance. Future work may consider a unified approach, training the model on the entire proton reaction database, although it remains uncertain whether this would guarantee better results while sacrificing data classification consistency.

In conclusion, each method for detecting outliers that was examined in this study has distinct advantages and limitations. Their effectiveness in identifying potential outliers varies depending on the specific characteristics of the datasets being analyzed; some methods excel in areas where others may fall short. In the realm of machine learning techniques, it is difficult to single out a definitive best performer. Although DBSCAN demonstrated marginally better outcomes, this efficacy is largely dependent on the nature of the data and the specific settings chosen for each analysis. Neural network methods, such as autoencoders, identified fewer outliers, which suggests they may be more accurate. However, the greater computational requirements and more complex parameter settings for neural networks raise questions about their practical suitability for the task at hand. Therefore, each technique offers its own perspective for identifying outliers, and their relative success is tied to the complexities of the datasets they are applied to and the specific objectives they aim to fulfill.

5.2 Future Work

The foundations established by this research pave the way for further study in the realm of outlier detection for specialized scientific databases like EXFOR. Our methodology has proven its utility in various respects but leaves room for advancements and refinements.

Future research could delve deeper into the fine-tuning of hyperparameters for different machine learning algorithms used in this study. While the default settings employed here provided a reasonable starting point, more meticulous hyperparameter optimization could potentially enhance the algorithms' performance. However, this fine-tuning would demand a substantial investment of time, and the benefits must be carefully weighed, especially as the results will still require expert evaluation and visual verification for final validation.

Another noteworthy consideration for future work centers around Group 7, which presents unique challenges due to its peculiar data structure. Modifying the preprocessing steps specifically for this group could make the existing methodologies more applicable and accurate, warranting further investigation.

One ambitious avenue that future research might explore is the comparison of EXFOR data with evaluated nuclear data libraries. These complex libraries consider various factors beyond just experimental results, offering an additional layer of validation for outlier detection. Yet, this approach is not without its complications, notably the need for rigorous preprocessing to make the data from both sources comparable. Ensuring that data are formatted equivalently will be crucial for meaningful comparative analyses.

By tackling these issues, subsequent research can build upon the groundwork of this study, potentially yielding more robust and refined techniques for outlier detection in scientific databases.

5.3 Final Conclusions

This master's thesis set out with a multifaceted objective aimed at integrating machine learning techniques into the realm of outlier detection within the proton induced reactions of EXFOR nuclear data library. The journey from ideation to realization was encapsulated through a series of structured steps, fulfilling each set aim in a systematic manner.

The study successfully developed a robust methodology tailored specifically for the complexity of the EXFOR data. Through comprehensive research and implementation, we not only devised

efficient strategies for data extraction, transformation, and preparation but also established a framework that serves as a blueprint for future research in applying machine learning to specialized scientific datasets. This foundation ensures that the machine learning algorithms implemented could operate effectively, fulfilling the thesis's objectives and setting the stage for continued exploration in this field.

Our exploration spanned various outlier detection techniques, from traditional statistical methods like the Interquartile Range (IQR) to more sophisticated machine learning algorithms such as Isolation Forest, DBSCAN, and the Local Outlier Factor (LOF). Each method was implemented, and its results were analyzed to assess its reliability, strengths, and weaknesses. Moreover, we took an ambitious step by integrating a neural network model, specifically an Autoencoder, into our repertoire of algorithms.

In the course of completing this work, we have successfully met all initial objectives and established a robust methodology for applying machine learning to outlier detection in the EXFOR library for proton-induced reactions. Perhaps even more significantly, the developed methodology is versatile enough to be effortlessly extended to other sections of the EXFOR library, offering the potential for comprehensive improvements. To foster further development and community engagement, all scripts and code generated through this research have been made publicly available on the GitHub EXFOR-ProtonReactions-Analysis Repository. This open-source approach not only fulfills the academic imperative of sharing knowledge but also sets the stage for ongoing collaborative efforts to refine and expand upon the work presented here.

6 SOCIAL & PROFESSIONAL RESPONSABILITY

This chapter examines the crucial aspects of social and professional responsibility within the context of enhancing the EXFOR Nuclear Data Library using machine learning techniques. The inherent ethical, professional, and social responsibilities that accompany the development and deployment of these technologies are explored. The chapter will also consider the alignment of the project with the United Nations' Sustainable Development Goals (SDGs). It is imperative that, as researchers and practitioners in the field of nuclear science and machine learning, we recognise the profound impact our work can have, not just scientifically, but also socially, ethically, and professionally.

6.1 Professional Responsability

When we talk about enhancing the EXFOR Nuclear Data Library using machine learning techniques, we are discussing a process that is far from simple. It requires a deep commitment to doing our work accurately and respectfully, understanding that we are building on the work of many others in our field.

Machine learning is an area that's always evolving. It's a bit like a race, where the finish line keeps moving further away just as you think you're getting close. So, it's our job as professionals in this field to keep learning, always staying up to speed with the latest techniques and ethical considerations. This is incredibly important because any inaccuracies or errors could potentially impact the evaluation of nuclear data libraries. In other words, we need to make sure our machine learning models are reliable and high-quality.

As professionals, we also have to share our work with our peers and contribute to the evaluation of others' work. This means giving constructive feedback and always seeking ways to improve our own work. We need to be mindful of others' confidentiality and intellectual property rights, and we need to follow professional guidelines or rules of conduct. Also, when we share our findings, we need to do so in a way that is clear and understandable, not only to other scientists but to the general public as well.

In essence, our task is not just to improve the EXFOR Nuclear Data Library. We have a responsibility to uphold the highest standards of scientific work. We must do so with honesty, transparency, and a deep respect for our field, always striving to push the boundaries of what we know and what we can achieve.

6.2 Social Responsibility and Sustainable Development Goals (SDGs)

Improving the EXFOR Nuclear Data Library through machine learning has extensive implications for sustainable development. Here, we specifically discuss the alignment of this project with several United Nations' Sustainable Development Goals (SDGs):

- **SDG 3: Good Health and Well-being:** Enhanced nuclear data libraries can improve the production of medical isotopes, which are critical for various diagnostic and therapeutic procedures in the healthcare sector. The accurate detection of outliers in the data could increase the efficiency and reliability of these isotopes, potentially enabling better disease diagnosis and treatment strategies.

- **SDG 7: Affordable and Clean Energy:** The improvements in the EXFOR Nuclear Data Library can lead to more efficient, safer nuclear energy production. Accurate and robust nuclear data are essential for the operation and safety assessment of nuclear power plants. Enhanced data accuracy could potentially reduce operational costs, thereby contributing to the affordability aspect of this goal.
- **SDG 9: Industry, Innovation and Infrastructure:** The application of machine learning techniques to the nuclear data library represents a significant innovation in the field. This could inspire further technological advancements and infrastructural improvements in the nuclear industry.
- **SDG 13: Climate Action:** Nuclear energy, being a low-carbon source of power, is pivotal in our attempts to mitigate climate change. Better nuclear data can result in more efficient and safer nuclear power plants, thereby helping to reduce greenhouse gas emissions.

By aligning the enhancement of the EXFOR Nuclear Data Library with these SDGs, we aim to ensure that our project contributes to sustainable development and the betterment of society as a whole.

In conclusion, the integration of machine learning techniques into the enhancement of the EXFOR Nuclear Data Library brings with it considerable professional and social responsibilities. As professionals, we have a duty to ensure the reliability, validity, and robustness of our research and development activities. As members of society, we are compelled to make certain that our work aligns with the global commitment to sustainable development and peaceful usage of nuclear technology. By fulfilling these responsibilities, we can ensure that our work not only advances our field but also contributes positively to society and our planet's future.

7 PROJECT MANAGEMENT

This chapter delves into the crucial aspects of project management involved in enhancing the EXFOR Nuclear Data Library using machine learning techniques. The facets covered in this chapter include both planning and budgeting processes integral to the successful execution of the project. The planning section offers a detailed account of the task distribution and project timeline, presenting a chronological breakdown of the project's structure and execution. We will also discuss some of the challenges faced during the project, including strategies and solutions employed to overcome them. Furthermore, the budgeting aspect of the project, an integral element that will be addressed in a subsequent section, plays a pivotal role in dictating resource allocation and influencing project timelines.

7.1 Planning

The planning phase was meticulously executed to ensure an optimal distribution of tasks, thereby facilitating the smooth progress of the project. Table 7.1 and Figure 7.1 both represent the timeline and distribution of tasks for this master's thesis project. The project was initiated on 1st June and concluded on 31st August, spanning over three months. The tasks were divided into two levels: Task Level 1, representing major phases of the project, and Task Level 2, indicating specific actions within these phases.

The project started with preparatory tasks, such as kick-off and data collection. Subsequent steps included data preprocessing, a crucial task that was allocated more time due to its complexity and importance to the overall project. Machine learning model training, outlier detection, and final model training followed this. Each phase was broken down into subtasks such as feature extraction, data cleaning, model optimization, and cross-validation to ensure comprehensive coverage of each aspect. The final stages of the project involved thesis writing and review, a process started soon enough to ensure adequate time for drafting, review, and revision.

Table 7.1: Project Timeline and Task Distribution

Task	**Duration (days)**	**Begin Date**	**End Date**
Project Kick-off and Preparation	9	1st June	9th June
- Data Collection	5	4th June	9th June
Data Preprocessing	31	9th June	10th July
- Feature Extraction	10	9th June	19th June
- Data Cleaning	21	19th June	10th July
Initial Machine Learning Model Training	16	10th July	26th July
- Model Optimization	9	10th July	19th July
- Cross-validation	7	19th July	26th July
Outlier Detection	15	27th July	10th Aug
- Outlier Analysis and Verification	15	27th July	10th Aug
Final Machine Learning Model Training	12	11th Aug	23th Aug
- Model Testing	12	11th Aug	23th Aug
Thesis Writing & Review	58	4th July	31st Aug
- Drafting	41	4th July	14th Aug
- Review and Revision	16	15th Aug	31st Aug

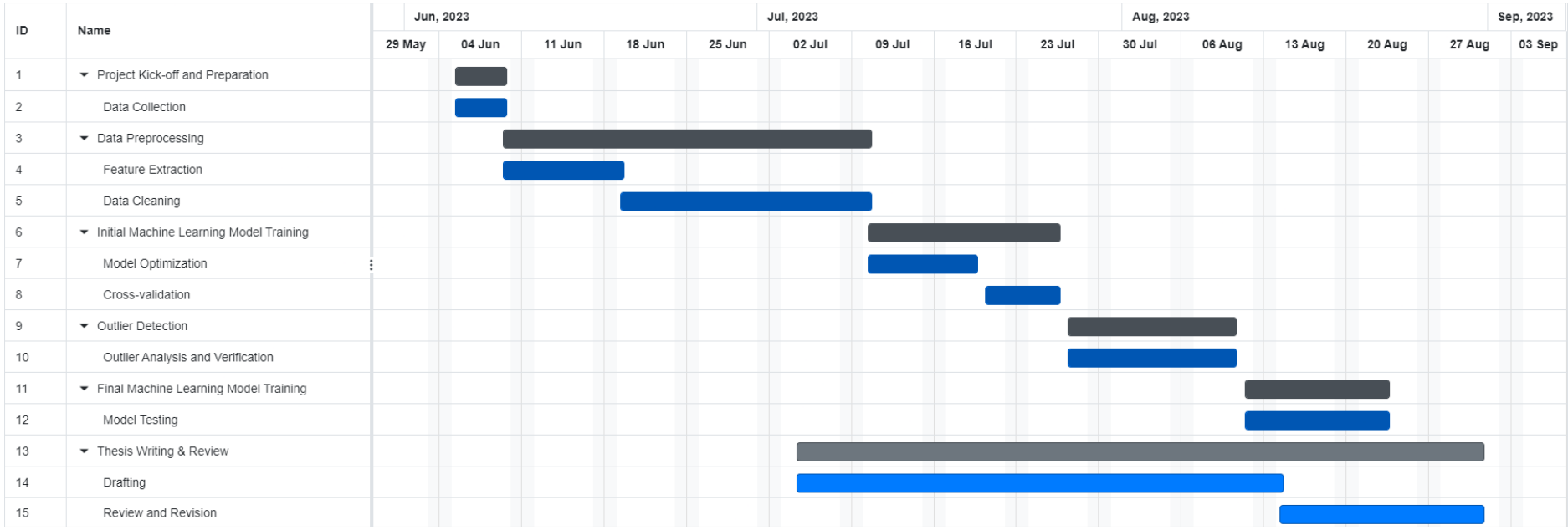

Figure 7.1: Gantt chart.

One of the most significant challenges encountered during the project was managing the tight timeline, particularly concerning thesis drafting. Due to the short duration of the project, drafting started concurrently with the later stages of the project. This parallel approach enabled progress on the thesis while still fine-tuning the project's technical aspects. Data preprocessing also posed a considerable challenge, given its critical role in the project. We approached this by allocating more time to this task, ensuring that the data was appropriately prepared for machine learning model training. This involved careful scheduling to avoid rushing the process, and iterative refinement as necessary.

In terms of the machine learning aspect, model optimization and outlier detection were challenging due to the complexity of the nuclear data and the potential impact of outliers on the results. We tackled this through a rigorous approach involving multiple iterations of model training, testing, optimization, and validation. This meticulous process ensured the reliable performance of the model and accurate outlier detection. By proactively addressing these challenges, we were able to effectively manage the project timeline and deliver the desired outcomes. This experience underscores the importance of robust project planning and adaptability in the face of unforeseen challenges.

7.2 Budget

In project management, budgeting is a critical component, serving as a financial map that aids the successful execution of a project. For this project, workforce expenses form the primary budgetary aspect, meticulously recorded in Table 7.2.

The student worked a total of 360 hours, corresponding to the 12-credit requirement for the thesis. The student was awarded a grant of 500€ per month for the project's execution, amounting to a total of 1500€ for the three-month duration of the project. This grant effectively means an hourly rate of 4.17€ for the student's work. On the other hand, the mentor dedicated approximately 30 hours to this project. With a rate of 35€ per hour, the cost associated with the mentor's contribution amounts to 1050€.

Table 7.2: Workforce Expenses

Concept	**Hours**	**Cost per hour (€)**	**Subtotal (€)**
Student	360	4.17	1500.00
Mentor	30	35.00	1050.00
Total			**2550.00**

In terms of resource expenses, a typical project would consider the cost of software licenses, peripherals, accommodation, utilities, and other operational costs. However, for this project, a decision was made to regard these resource expenses as being effectively amortized. This is largely because the project leveraged my personal resources. For instance, my personal computer was used for all tasks, negating the need for additional hardware investments.

Furthermore, the software tools employed for the project, including those for machine learning and document preparation like Overleaf, are open-source or freely available. This decision not only promotes the use of accessible resources but also significantly brings down the cost associated with software procurement. Regarding accommodation and utility expenses, these have been absorbed into my personal living costs as I carried out the project from my family

home. Thus, no separate accommodation expenses are factored into the project budget.

When considering the tax implications of the project budget, it's important to note that in Spain, grants are tax-exempt. Therefore, the grant amount of 1500€ for the student does not factor into the tax calculation. The taxable amount is solely the mentor's fee of 1050€. Applying the 21% tax rate to the mentor's fee, we obtain a tax amount of 220.50€. Therefore, the final budget for the project, including tax, becomes 1500 (student grant) + 1050 (mentor's fee) + 220.50 (tax) = 2770.50€.

8 BIBLIOGRAPHY

References

[1] S. Team and M. Santos, "Machine learning de a a la z: R y python para data science." urlhttps://www.udemy.com/course/machinelearning-es/, 2021. Udemy.

[2] N. I. of Standards and Technology, "1.3.5.17. detection of outliers." `https://www.itl.nist.gov/div898/handbook/eda/section3/eda35h.htm`.

[3] IAEA, "Experimental nuclear reaction data — iaea." `https://www.iaea.org/resources/databases/experimental-nuclear-reaction-data`.

[4] ASCO.org, "Proton therapy — cancer.net." `https://www.cancer.net/navigating-cancer-care/how-cancer-treated/radiation-therapy/proton-therapy`.

[5] J. Hopkins, "Proton therapy — johns hopkins medicine." `https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/proton-therapy`.

[6] M. Clinic, "Proton therapy - mayo clinic." `https://www.mayoclinic.org/tests-procedures/proton-therapy/about/pac-20384758`.

[7] M. Ofuya, L. McParland, L. Murray, S. Brown, D. Sebag-Montefiore, and E. Hall, "Systematic review of methodology used in clinical studies evaluating the benefits of proton beam therapy," *Clinical and Translational Radiation Oncology*, vol. 19, pp. 17–26, 11 2019.

[8] A. e. a. Author, *EXFOR Formats Manual*, 1979. In J. Nucl. Phys., Vol. 12, p. 345. EXFOR A0123.012.

[9] I. A. E. Agency, "Exfor formats description for users." `https://www-nds.iaea.org/exfor/exfor_formats.pdf`, 2008. [Accessed: 05/07/2023].

[10] O. Cabellos and N. Otsuka, "Handbook on nuclear data great-pioneer eu project," tech. rep., Great-Pioneer, 2022.

[11] A. Lewis, D. Neudecker, A. Koning, D. Barry, J. Brown, and G. Schnabel, "Wpec sg50: Developing an automatically readable, comprehensive and curated experimental nuclear reaction database," *EPJ Web of Conferences*, vol. 284, p. 18003, 2023. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0.

[12] I. NDS, "Github repository iaea nds." `https://github.com/IAEA-NDS`. Accessed: 15/07/23.

[13] "Oecd nuclear energy agency." `https://www.oecd-nea.org/`. Accessed: 13/07/2023.

[14] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.

[15] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, and D. I. Fotiadis, "Machine learning applications in cancer prognosis and prediction," *Computational and structural biotechnology journal*, vol. 13, pp. 8–17, 2015.

[16] R. J. Bolton and D. J. Hand, "Statistical fraud detection: A review," *Statistical Science*, pp. 235–249, 2002.

[17] A. K. Sikder and A. R. Ganguly, "An overview of computing approaches for predicting short term renewable energy," *Current Sustainable/Renewable Energy Reports*, vol. 5, no. 4, pp. 135–146, 2018.

[18] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[19] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia: Pearson Education Limited, 2016.

[20] F. E. Grubbs, "Procedures for detecting outlying observations in samples," *Technometrics*, vol. 11, no. 1, pp. 1–21, 1969.

[21] D. M. Hawkins, *Identification of outliers*, vol. 11. Springer, 1980.

[22] V. J. Hodge and J. Austin, "A survey of outlier detection methodologies," in *Artificial intelligence review*, vol. 22, pp. 85–126, Springer, 2004.

[23] V. Barnett and T. Lewis, *Outliers in statistical data*. Wiley, 1994.

[24] H. W. Ives, "Evaluation of tests for normality," *Statistical Distributions in Scientific Work*, vol. 4, pp. 37–52, 1986.

[25] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata, "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median," *Journal of Experimental Social Psychology*, vol. 49, no. 4, pp. 764–766, 2013.

[26] D. C. Hoaglin, F. Mosteller, and J. W. Tukey, *Exploring data tables, trends, and shapes*. Wiley, 1986.

[27] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[28] A. K. Jain, M. Murty, and P. Flynn, "Data clustering: a review," *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[29] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," in *Neural computation*, vol. 13, pp. 1443–1471, MIT Press, 2001.

[30] N. Chauhan and L. F. Vargas, "Anomaly detection using autoencoders," *Journal of Physics: Conference Series*, vol. 1222, no. 1, p. 012018, 2019.

[31] Wikipedia contributors, "Interquartile range — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Interquartile_range&oldid=1171498340`, 2023. [Accessed: 25/08/2023].

[32] Chris Kuo and Dr. Dataman, "Handbook of anomaly detection with python outlier detection." `https://towardsdatascience.com/use-the-isolated-forest-with-pyod-3818eea68f08`, 2019. [Accessed: 25/08/2023].

[33] O. Ali, M. K. Ishak, and M. K. Bhatti, "A machine learning approach for early covid-19 symptoms identification," *Computers, Materials and Continua*, vol. 70, pp. 3803–3820, 09 2021.

[34] A. Elmogy, H. Rizk, and A. Sarhan, "Ofcod: On the fly clustering based outlier detection framework," *Data*, vol. 6, p. 1, 12 2020.

[35] B. Koo and B. Shin, "Using geometry based anomaly detection to check the integrity of ifc classifications in bim models," *Journal of KIBIM*, vol. 7, pp. 18–27, 03 2017.

[36] Arden Dertat, "Applied deep learning - part 3: Autoencoders." `https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798`, 2017. [Accessed: 25/08/2023].

[37] A. Koning, "EXFORTABLES-1.0 User Manual," 2019.

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

AI Artificial Intelligence

DBSCAN Density-based clustering non-parametric algorithm

DL Deep Learning

ENDL Evaluated Nuclear Data Libraries

EXFOR EXchange FORmat

IQR Interquartile Rrange

LOF Local Outlier Factor

MEDUSAL Machine-readable Experimental Data User Application & Library

ML Machine Learning

NEA Nuclear Energy Agency

OECD Organisation for Economic Co-operation and Development's

SDG Sustainable Development Goals

SVM Support Vector Machines

WPEC Working Party on International Nuclear Data Evaluation Co-operation

ANNEXES

A 'Experiment" Class Source Code and Documentation

```
class Experiment:
    """
    This class models an experimental setup for scientific data collection.
    Attributes include various experimental parameters and data in the form of a Pandas DataFrame.
    """
    def __init__(self):
        """
        Initializes an Experiment object with default values set to None and an empty Pandas DataFrame for data.
        """
        self.title = None
        self.target_Z = None
        self.target_A = None
        self.target_state = None
        self.projectile = None
        self.reaction = None
        self.E_inc = None
        self.final_Z = None
        self.final_A = None
        self.final_state = None
        self.MTrat = None
        self.Ratio_isomer = None
        self.quantity = None
        self.frame = None
        self.MF = None
        self.MT = None
        self.X4_ID = None
        self.X4_code = None
        self.author = None
        self.year = None
        self.data_points = None
        self.data = pd.DataFrame()
        self.reference = None


    def __str__(self):
        """
        Overloaded string method to print non-empty attributes of an Experiment object.
        """
        print("Title: ", self.title)
        if self.target_Z != None: print("Target Z: ", self.target_Z)
        if self.target_A != None: print("Target A: ", self.target_A)
        if self.target_state != None: print("Target state: ", self.target_state)
        if self.projectile != None: print("Projectile: ", self.projectile)
        if self.reaction != None: print("Reaction: ", self.reaction)
        if self.E_inc != None: print("Incident energy: ", self.E_inc)
        if self.final_Z != None: print("Final Z: ", self.final_Z)
        if self.final_A != None: print("Final A: ", self.final_A)
```

```
48          if self.final_state != None: print("Final state: ", self.
               final_state)
49          if self.MTrat != None: print("MT ratio: ", self.MTrat)
50          if self.Ratio_isomer != None: print("Ratio isomer: ", self.
               Ratio_isomer)
51          if self.quantity != None: print("Quantity: ", self.quantity)
52          if self.frame != None: print("Frame: ", self.frame)
53          if self.MF != None: print("MF: ", self.MF)
54          if self.MT != None: print("MT: ", self.MT)
55          if self.X4_ID != None: print("X4 ID: ", self.X4_ID)
56          if self.X4_code != None: print("X4 code: ", self.X4_code)
57          if self.author != None: print("Author: ", self.author)
58          if self.year != None: print("Year: ", self.year)
59          if self.data_points != None: print("Data points: ", self.
               data_points)
60          if not self.data.empty: print("Data: ", self.data)
61          if self.reference != None: print("Reference:\n", self.reference)
62          return ""
63
64
65      def to_dataframe(self):
66          """
67          Converts the attributes of the object into a single DataFrame
               where each attribute is a column.
68          Returns a DataFrame containing the enriched data.
69          """
70          # Starting with the data in self.data, we are going to add
               information from the other attributes
71          # The information will be added as a new column in the dataframe
                for all the rows in the dataframe
72
73          # List with all the attributes to add to the dataframe
74          attributes = ["Experiment", "Target Z", "Target A", "Target
               state", "Reaction", "Incident energy",
75                        "Final Z", "Final A", "Final state", "MT ratio", "
                           Ratio isomer", "Quantity", "Frame",
76                        "MF", "MT", "Author", "Year"]
77
78          # Create a copy of the data
79          data = self.data.copy()
80          # Add the attributes to the dataframe
81          for attribute in attributes:
82              # Create a list with the value of the attribute for all the
                   rows in the dataframe
83              if attribute == "Experiment": value = [self.title]*len(data)
84              elif attribute == "Target Z": value = [self.target_Z]*len(
                   data)
85              elif attribute == "Target A": value = [self.target_A]*len(
                   data)
86              elif attribute == "Target state": value = [self.target_state
                   ]*len(data)
87              elif attribute == "Reaction": value = [self.reaction]*len(
                   data)
88              elif attribute == "Incident energy": value = [self.E_inc]*
                   len(data)
89              elif attribute == "Final Z": value = [self.final_Z]*len(data
                   )
```

```
            elif attribute == "Final A": value = [self.final_A]*len(data
                )
            elif attribute == "Final state": value = [self.final_state]*
                len(data)
            elif attribute == "MT ratio": value = [self.MTrat]*len(data)
            elif attribute == "Ratio isomer": value = [self.Ratio_isomer
                ]*len(data)
            elif attribute == "Quantity": value = [self.quantity]*len(
                data)
            elif attribute == "Frame": value = [self.frame]*len(data)
            elif attribute == "MF": value = [self.MF]*len(data)
            elif attribute == "MT": value = [self.MT]*len(data)
            elif attribute == "Author": value = [self.author]*len(data)
            elif attribute == "Year": value = [self.year]*len(data)
            # Add the column to the dataframe
            data[attribute] = value
        # Return the dataframe
        return data


    def add_numeric_attributes(self):
        """
        Adds numeric attributes to the DataFrame by converting suitable
            string values to their respective numeric forms.
        """
        attributes = ["E_inc", "MF", "MT", "MTrat", "Ratio_isomer", "
            final_A", "final_Z", "target_A", "target_Z"]

        for attr in attributes:
            if attr == "E_inc":
                value = getattr(self, attr)
                if value is not None:
                    first_element = value.split()[0]
                    self.data[attr] = float(first_element)
                else:
                    self.data[attr] = None
            elif attr in ["MTrat", "final_A", "final_Z"]:
                value = getattr(self, attr)
                if value is not None:
                    self.data[attr] = int(value)
                else:
                    self.data[attr] = None
            else:
                self.data[attr] = getattr(self, attr)


    def encode_categorical_attributes(self):
        """
        Transforms categorical attributes to a format suitable for model
            training by using one-hot encoding.
        """
        categorical_attributes = {
            "projectile": ["projectile_p"],
            "final_state": ["final_state_+", "final_state_1", "
                final_state_2", "final_state_G", "final_state_M"],
            "frame": ["frame_C", "frame_L"],
            "quantity": ["qty_Angular distribution", "qty_Cross section"
                , "qty_Cross section ratio", "qty_Delayed nubar", "
```

```
            qty_Differential cross section", "qty_Fission yields", "
            qty_Prompt nubar", "qty_Resonance Parameters", "qty_Total
             nubar"],
        "reaction": ["reaction_(n, )", "reaction_(p, el)", "
            reaction_(p, f)", "reaction_(p, x)", "reaction_(p, xa)",
            "reaction_(p, xd)", "reaction_(p, xg)", "reaction_(p, xh)
            ", "reaction_(p, xn)", "reaction_(p, xp)", "reaction_(p,
            xt)", "reaction_(p,2a)", "reaction_(p,2n)", "reaction_(p
            ,2n)g", "reaction_(p,2n)m", "reaction_(p,2na)", "
            reaction_(p,2np)", "reaction_(p,2p)", "reaction_(p,2p)g",
             "reaction_(p,2p)m", "reaction_(p,3a)", "reaction_(p,3n)"
            , "reaction_(p,3n)g", "reaction_(p,3n)m", "reaction_(p,3n
            )n", "reaction_(p,3na)", "reaction_(p,3np)", "reaction_(p
            ,3np)g", "reaction_(p,3np)m", "reaction_(p,4n)", "
            reaction_(p,4n)g", "reaction_(p,4n)m", "reaction_(p,a)",
            "reaction_(p,a)g", "reaction_(p,a)m", "reaction_(p,d)", "
            reaction_(p,d2a)", "reaction_(p,da)", "reaction_(p,f)", "
            reaction_(p,f)g", "reaction_(p,f)m", "reaction_(p,f)n", "
            reaction_(p,g)", "reaction_(p,g)g", "reaction_(p,g)m", "
            reaction_(p,h)", "reaction_(p,h)g", "reaction_(p,n)", "
            reaction_(p,n")g", "reaction_(p,n")m", "reaction_(p,n")n"
            , "reaction_(p,n"_01)", "reaction_(p,n"_40)", "reaction_(
            p,n2a)", "reaction_(p,n2p)", "reaction_(p,n3a)", "
            reaction_(p,na)", "reaction_(p,na)g", "reaction_(p,na)m",
             "reaction_(p,non)", "reaction_(p,np)", "reaction_(p,np)g
            ", "reaction_(p,np)m", "reaction_(p,npa)", "reaction_(p,p
            )", "reaction_(p,p)m", "reaction_(p,pa)", "reaction_(p,pd
            )", "reaction_(p,pt)", "reaction_(p,t)", "reaction_(p,xa)
            ", "reaction_(p,xd)", "reaction_(p,xg)", "reaction_(p,xh)
            ", "reaction_(p,xn)", "reaction_(p,xp)", "reaction_(p,xt)
            ", "reaction_Exchange_scattering", "
            reaction_Inelastic_scattering", "reaction_ratio"],
        "target_state": ["target_state_m"]
    }
    new_columns_data = {}

    for attribute, possible_values in categorical_attributes.items()
        :
        # Get the current attribute value of this instance
        current_attribute_value = getattr(self, attribute)

        for value in possible_values:
            # Get the value after the last underscore
            comparison_value = value.split("_")[-1]
            # Check if the attribute value of the current instance
                matches the comparison value
            new_columns_data[value] = int(current_attribute_value ==
                comparison_value)

    # Convert the dictionary into a DataFrame and concatenate it
        with the original DataFrame
    new_columns_df = pd.DataFrame(new_columns_data, index=self.data.
        index)
    self.data = pd.concat([self.data, new_columns_df], axis=1)



def prepare_data(self):
```

```
        """
        Combines the functionalities of add_numeric_attributes and
           encode_categorical_attributes to prepare the data for
           analysis.
        Returns the prepared DataFrame.
        """
        self.add_numeric_attributes()
        self.encode_categorical_attributes()
        self.data["X4_ID"] = str(getattr(self, "X4_ID"))
        return self.data


    def plot(self, xlog=False, ylog=False, fig_size=(9,6)):
        """
        Creates a plot of the stored data with optional logarithmic
           scaling for x and/or y axes.
        Parameters:
        - xlog (bool): Whether to use log scale on the x-axis.
        - ylog (bool): Whether to use log scale on the y-axis.
        - fig_size (tuple): Tuple specifying the dimensions of the plot.
        """
        if self.data.empty:
            print("No data to plot")
        else:
            # check if the data in the third column are all NaN or 0
            y_err = self.data.iloc[:,2].isnull().values.all() or self.
               data.iloc[:,2].eq(0).all()
            # check if the data in the fourth column are all NaN or 0
            x_err = self.data.iloc[:,3].isnull().values.all() or self.
               data.iloc[:,3].eq(0).all()

            # Get the headers of the data
            headers = list(self.data.columns.values)

            # Set the size of the plot
            plt.figure(figsize=fig_size)
            # Plot the data with a scatter plot
            # If y_err and x_err are False, plot the data with error
               bars
            if y_err == False and x_err == False:
                plt.errorbar(x=self.data[headers[0]], y=self.data[
                   headers[1]],
                             xerr=self.data[headers[3]], yerr=self.data[
                                headers[2]],
                             fmt="o", ecolor="black", capsize=3,
                                elinewidth=1,  markersize=5)
            # If y_err is False, plot the data with error bars on the y-
               axis
            elif y_err == False:
                plt.errorbar(x=self.data[headers[0]], y=self.data[
                   headers[1]], yerr=self.data[headers[2]],
                             fmt="o", ecolor="black", capsize=3,
                                elinewidth=1,  markersize=5,)
            # If x_err is False, plot the data with error bars on the x-
               axis
            elif x_err == False:
                plt.errorbar(x=self.data[headers[0]], y=self.data[
                   headers[1]], xerr=self.data[headers[3]],
```

```
205                                 fmt="o", ecolor="black", capsize=3,
                                        elinewidth=1,  markersize=5,)
206             # If y_err and x_err are False, plot the data without error
                    bars
207             else:
208                 sns.scatterplot(x=headers[0], y=headers[1], data=self.
                        data)
209
210
211             # Set the scale of the plot
212             if xlog == True: plt.xscale("log")
213             if ylog == True: plt.yscale("log")
214
215             # Set the title of the plot
216             plt.title(self.title, fontsize=18, pad=15)
217             # Set the labels of the plot
218             plt.xlabel(headers[0], fontsize=16, labelpad=10)
219             plt.ylabel(headers[1], fontsize=16, labelpad=10)
220             # Set the ticks of the plot
221             plt.xticks(fontsize=14)
222             plt.yticks(fontsize=14)
223             # Show the plot
224             plt.show()
```

B Utility Functions for Proton Experiment Data Analysis

B.1 read_experiment()

```
def read_experiment(filename):
    """
    This function reads a file specified by "filename", processes its content line by line, and populates the properties
    of an Experiment object according to specific keywords present in the file. It finally returns this populated
    Experiment object.

    File Format:
    The file should contain lines starting with specific keywords, followed by a colon and the value (EXFORTABLES format).
    (e.g., "# Target Z: 12")

    Properties of Returned Experiment Object:
    - title (str): Extracted from the filename.
    - target_Z (int or None): Atomic number of the target.
    - target_A (int or None): Mass number of the target.
    - target_state (int/str): the state of the target.
    - projectile (str): the projectile used in the experiment.
    - reaction (str): the reaction that occurred during the experiment.
    - E_inc (str): the incident energy of the projectile.
    - final_Z (int/str): the atomic number of the final state.
    - final_A (int/str): the mass number of the final state.
    - final_state (int/str): the state of the final state.
    - MTrat (float/str): the ratio of the number of gamma rays to the number of neutrons.
    - Ratio_isomer (float/str): the ratio of the number of isomers to the number of ground state.
    - Quantity (str): the quantity of the experiment.
    - Frame (str): the frame of the experiment.
    - MF (int/str): the MF of the experiment.
    - MT (int/str): the MT of the experiment.
    - X4_ID (str): the EXFOR ID of the experiment.
    - X4_code (str): the EXFOR code of the experiment.
    - author (str): the author of the experiment.
    - year (int/str): the year of the experiment.
    - data_points (int/str): the number of data points in the experiment.
    - data (pd.DataFrame): the data of the experiment.
    - Reference (str): the reference of the experiment.

    If a property value is not found in the file or cannot be converted to its respective type, the value is set to None.

    Dependencies:
    - Pandas: Required for storing experimental data in a DataFrame.
    - NumPy: Required for handling "nan" values.

    Parameters:
        filename (str): Name or path of the file to read.

    Returns:
        experiment (Experiment): Populated Experiment object.
```

```
47
48      Raises:
49          FileNotFoundError: If the file specified by "filename" does not
                exist.
50      """
51
52      # Create the experiment
53      experiment = Experiment()
54      experiment.title = filename.split("\\")[-1]
55      print("Reading experiment: ", experiment.title)
56      read_header = False          # Flag to read the header
57      read_ref = False             # Flag to read the reference
58      temp_line = ""               # Temporary line to store the reference
59
60      # Open the file
61      with open(filename, "r") as f:
62          # Read the first line
63          lines = f.readlines()
64          for line in lines:
65
66              if line.startswith("#") and not read_ref and not read_header
                    :
67                  if line.startswith("# Target Z"):
68                      try:
69                          experiment.target_Z = int(line.split(":")[1].
                                strip())
70                      except ValueError:
71                          experiment.target_Z = line.split(":")[1].strip()
                                if line.split(":")[1].strip() != "" else
                                None
72                  elif line.startswith("# Target A"):
73                      try:
74                          experiment.target_A = int(line.split(":")[1].
                                strip())
75                      except ValueError:
76                          experiment.target_A = int(line.split(":")[1].
                                strip()) if line.split(":")[1].strip() != ""
                                else None
77                  elif line.startswith("# Target state"):
78                      try:
79                          experiment.target_state = int(line.split(":")
                                [1].strip())
80                      except ValueError:
81                          experiment.target_state = line.split(":")[1].
                                strip() if line.split(":")[1].strip() != ""
                                else None
82                  elif line.startswith("# Projectile"):
83                      experiment.projectile = line.split(":")[1].strip()
                            if line.split(":")[1].strip() != "" else None
84                  elif line.startswith("# Reaction    :"):           #
                        This includes the ":" because there is another line
                        with the same beginning in some files
85                      experiment.reaction = line.split(":")[1].strip() if
                            line.split(":")[1].strip() != "" else None
86                  elif line.startswith("# E-inc"):
87                      experiment.E_inc = line.split(":")[1].strip() if
                            line.split(":")[1].strip() != "" else None
88                  elif line.startswith("# Final Z"):
```

```
                    try:
                        experiment.final_Z = int(line.split(":")[1].
                            strip())
                    except ValueError:
                        experiment.final_Z = line.split(":")[1].strip()
                            if line.split(":")[1].strip() != "" else None
                elif line.startswith("# Final A"):
                    try:
                        experiment.final_A = int(line.split(":")[1].
                            strip())
                    except ValueError:
                        experiment.final_A = line.split(":")[1].strip()
                            if line.split(":")[1].strip() != "" else None
                elif line.startswith("# Final state"):
                    try:
                        experiment.final_state = int(line.split(":")[1].
                            strip())
                    except ValueError:
                        experiment.final_state = line.split(":")[1].
                            strip() if line.split(":")[1].strip() != ""
                            else None
                elif line.startswith("# MTrat"):
                    try:
                        experiment.MTrat = float(line.split(":")[1].
                            strip())
                    except ValueError:
                        experiment.MTrat = float(line.split(":")[1].
                            strip()) if line.split(":")[1].strip() != ""
                            else None
                elif line.startswith("# Ratio isomer"):
                    try:
                        experiment.Ratio_isomer = float(line.split(":")
                            [1].strip())
                    except ValueError:
                        experiment.Ratio_isomer = float(line.split(":")
                            [1].strip()) if line.split(":")[1].strip() !=
                             "" else None
                elif line.startswith("# Quantity"):
                    experiment.quantity = line.split(":")[1].strip() if
                        line.split(":")[1].strip() != "" else None
                elif line.startswith("# Frame"):
                    experiment.frame = line.split(":")[1].strip() if
                        line.split(":")[1].strip() != "" else None
                elif line.startswith("# MF"):
                    try:
                        experiment.MF = int(line.split(":")[1].strip())
                    except ValueError:
                        experiment.MF = int(line.split(":")[1].strip())
                            if line.split(":")[1].strip() != "" else None
                elif line.startswith("# MT"):
                    try:
                        experiment.MT = int(line.split(":")[1].strip())
                    except ValueError:
                        experiment.MT = int(line.split(":")[1].strip())
                            if line.split(":")[1].strip() != "" else None
                elif line.startswith("# X4 ID"):
                    experiment.X4_ID = line.split(":")[1].strip() if
                        line.split(":")[1].strip() != "" else None
```

```
                elif line.startswith("# X4 code"):
                    experiment.X4_code = line.split(":")[1].strip() if line.split(":")[1].strip() != "" else None
                elif line.startswith("# Author"):
                    experiment.author = line.split(":")[1].strip() if line.split(":")[1].strip() != "" else None
                elif line.startswith("# Year"):
                    try:
                        experiment.year = int(line.split(":")[1].strip())
                    except ValueError:
                        experiment.year = int(line.split(":")[1].strip()) if line.split(":")[1].strip() != "" else None
                elif line.startswith("# Data points"):
                    try:
                        experiment.data_points = int(line.split(":")[1].strip())
                    except ValueError:
                        experiment.data_points = int(line.split(":")[1].strip()) if line.split(":")[1].strip() != "" else None
                    read_header = True
                elif line.startswith("# Reference"):
                    read_ref = True

            elif read_header:
                header = line.split()[1:]
                data_list = [[] for i in range(len(header))]
                read_header = False

            elif read_ref:
                # If line only contains "#\n", then it is the end of the reference
                if line == "#\n":
                    # add the reference to the experiment except for the last two characters (which are "# ")
                    experiment.reference = temp_line[:-2]
                    read_ref = False        # Reset the flag
                    temp_line = ""          # Reset the temporary line
                    continue
                temp_line += line[1:]

            else:
                data = line.split()
                # Get number of splits
                n = len(data)
                data_list[0].append(float(data[0]))
                data_list[1].append(float(data[1]))
                # If there are 2 splits, then there are no values for dE and dxs
                if n == 2:
                    data_list[2].append(np.nan)
                    data_list[3].append(np.nan)
                # If there are 3 splits, then there are no values for dE
                elif n == 3:
                    data_list[2].append(float(data[2]))
                    data_list[3].append(np.nan)
```

```
176                 # If there are 4 splits, then there are values for dE
                        and dxs
177                 elif n == 4:
178                     data_list[2].append(float(data[2]))
179                     data_list[3].append(float(data[3]))
180 
181         if read_ref:        # This being True means that the file ended
                directly after the reference
182             experiment.reference = temp_line[:-2]
183 
184         # Close the file
185         f.close()
186 
187     # Create a data frame using "header" and "E", "xs", "dxs" and "dE"
        lists
188     experiment.data = pd.DataFrame(data_list, index=header).T
189 
190     # Return the experiment
191     return experiment
```

B.2 write_experiments_to_binary()

```
def write_experiments_to_binary(experiments, filename):
    """
    This function writes a list of Experiment objects to a binary file.
    It uses Python"s pickle library for object serialization.

    Parameters:
        experiments (list): A list of Experiment objects to be written
           to the file.
        filename (str): The name of the file where the experiments will
           be written.

    Returns:
        None

    Example:
        experiments = [experiment1, experiment2, experiment3]
        filename = "experiments.bin"
        write_experiments_to_binary(experiments, filename)

    Note:
        The function uses Python"s pickle library, so it"s essential to
           ensure that the Experiment objects are picklable.
        Be aware of the potential security risks if you"re unpickling
           data from an untrusted source.
    """

    with open(filename, "wb") as f:
        for experiment in experiments:
            pickle.dump(experiment, f)
```

B.3 read_experiments_from_binary()

```
def read_experiments_from_binary(filename):
    """
    This function reads a list of Experiment objects from a binary file
       and returns them as a list.
    It uses Python"s pickle library for object deserialization.

    Parameters:
        filename (str): The name of the binary file to read the
           experiments from.

    Returns:
        loaded_experiments (list): A list of experiments that were read
           from the binary file.

    Note:
        The function uses Python"s pickle library, so be cautious of
           potential security risks if you"re unpickling data from an
           untrusted source.
    """

    loaded_experiments = []

    # Use a with statement to ensure the file is properly closed after
       reading
    with open(filename, "rb") as f:
        while True:
            try:
                loaded_experiments.append(pickle.load(f))
            except EOFError:
                break
            except pickle.PickleError:
                print("Error in deserializing object. Skipping...")
                break

    return loaded_experiments
```

B.4 write_experiments_to_txt()

```
def write_experiments_to_txt(experiments, filename):
    """
    write_experiments_to_txt(experiments, filename)

    Serializes a list of experiment objects into a text file, capturing various attributes of each experiment.
    The function handles empty string attributes by writing them as such in the output file. Once all experiment details are written, the file is closed.

    Parameters:
        experiments (list): A list of experiment objects.

        filename (str): The name of the output text file where the experiment details will be serialized.

    Returns:
        None

    Example:
        experiments = [experiment1, experiment2]
        filename = "experiments.txt"
        write_experiments_to_txt(experiments, filename)

    Note:
        Ensure that each experiment object in the list has all the mentioned attributes. Missing attributes will lead to an error.
    """

    with open(filename, "w") as f:
        for experiment in experiments:
            # Write the header
            f.write("# Title       : " + experiment.title + "\n")
            # Write the information. If the information is "", then write an empty string
            f.write("# Reaction    : " + experiment.reaction  + "\n" if experiment.reaction is not None else "# Reaction    : \n")
            f.write("# Ratio isomer: " + str(experiment.Ratio_isomer) + "\n" if experiment.Ratio_isomer is not None else "# Ratio isomer: \n")
            f.write("# Quantity    : " + experiment.quantity + "\n" if experiment.quantity is not None else "# Quantity    : \n")
            f.write("# Frame       : " + experiment.frame + "\n" if experiment.frame is not None else "# Frame       : \n")
            f.write("# MF          : " + str(experiment.MF) + "\n" if experiment.MF is not None else "# MF          : \n")
            f.write("# MT          : " + str(experiment.MT) + "\n" if experiment.MT is not None else "# MT          : \n")
            f.write("# X4 ID       : " + experiment.X4_ID + "\n" if experiment.X4_ID is not None else "# X4 ID       : \n")
            f.write("# X4 code     : " + experiment.X4_code + "\n" if experiment.X4_code is not None else "# X4 code     : \n")
```

```
38              f.write("# Author       : " + experiment.author + "\n" if
                    experiment.author is not None else "# Author       : \n")
39              f.write("# Year         : " + str(experiment.year) + "\n" if
                    experiment.year is not None else "# Year         : \n")
40              f.write("# Data points : " + str(experiment.data_points) + "
                    \n" if experiment.data_points is not None else "# Data
                    points : \n")
41              # Write the data
42              f.write(experiment.data.to_string(header=True, index=False))
43              # Write the reference
44              # If the reference is "", then write an empty string
45              f.write("\n# Reference   : \n" + experiment.reference + "\n"
                     if experiment.reference is not None else "# Reference
                    : \n")
46              # Write a line to separate the reference from the next
                    experiment
47              f.write("# END\n")
48
49          f.write("# END OF FILE")
```

B.5 read_experiments_from_txt()

```
def read_experiments_from_txt(filename):
    """
    Reads a text file containing a structured list of experiments and returns a list of Experiment objects.

    The function expects the file to be structured as follows:
    - Each experiment block starts with a line for the title ("# Title:") and ends with "# END".
    - Between these lines, each line should follow the format: "# [field_name] : [field_value]".
    - After "# END", the data of the experiment is present with the first line as the header.
    - References start with "# Reference :" and end with "# END".
    - The file should end with a line containing "# END OF FILE".

    If a field is missing, the corresponding attribute in the Experiment object is set to an empty string ("").

    Parameters:
    ------------
    filename : str
        The name of the text file containing the experiments" details.

    Returns:
    ---------
    experiments : list
        A list of Experiment objects populated with the details read from the text file.

    Raises:
    -------
    ValueError:
        If the data in the file is not structured as expected.

    Example:
    --------
    experiments = read_experiments_from_txt("experiment_data.txt")

    Note:
    -----
    The function expects that each Experiment object has attributes corresponding to the fields in the text file.
    """

    # Create a list of experiments
    experiments = []
    # Open the file
    f = open(filename, "r")
    # Read the first line
    line = f.readline()
    # While you dont read "# END OF FILE" read lines
    while True:
        # If the line starts with "# END OF FILE", then you have reached the end of the file
        if line.startswith("# END OF FILE"):
            # Close the file
```

```
            f.close()
            # Return the list of experiments
            return experiments
        # While you dont read "# END" read lines
        while not line.startswith("# END"):
            # Create a new experiment
            experiment = Experiment()
            # Read the title exluding any posible "\n" at the end
            experiment.title = line.split(":")[1][:-1]
            # Read the rest of the information. If the information is
               empty, then set it to an empty string. Exclude any
               posible "\n" at the end
            line = f.readline()
            experiment.reaction = line.split(":")[1][:-1] if line.split(
               ":")[1] != "" else ""
            line = f.readline()
            try:
                experiment.Ratio_isomer = float(line.split(":")[1])
            except ValueError:
                experiment.Ratio_isomer = ""
            line = f.readline()
            experiment.quantity = line.split(":")[1][:-1] if line.split(
               ":")[1] != "" else ""
            line = f.readline()
            experiment.frame = line.split(":")[1][:-1] if line.split(":"
               )[1] != "" else ""
            line = f.readline()
            try:
                experiment.MF = float(line.split(":")[1])
            except ValueError:
                experiment.MF = ""
            line = f.readline()
            try:
                experiment.MT = float(line.split(":")[1])
            except ValueError:
                experiment.MT = ""
            line = f.readline()
            experiment.X4_ID = line.split(":")[1][:-1] if line.split(":"
               )[1] != "" else ""
            line = f.readline()
            experiment.X4_code = line.split(":")[1][:-1] if line.split("
               :")[1] != "" else ""
            line = f.readline()
            experiment.author = line.split(":")[1][:-1] if line.split(":
               ")[1] != "" else ""
            line = f.readline()
            try:
                experiment.year = int(line.split(":")[1])
            except ValueError:
                experiment.year = ""
            line = f.readline()
            try:
                experiment.data_points = int(line.split(":")[1])
            except ValueError:
                experiment.data_points = ""
            # Read the data. All data starts without "#". The first line
                is the header.
            # Read the header
```

```
98              line = f.readline()
99              header = line.split()
100             # Read the data
101             data = []
102             line = f.readline()
103             while not line.startswith("#"):
104                 data.append(line.split())
105                 line = f.readline()             # In the last iteration,
                        the line is "# Reference   :\n"
106             # Create a dataframe with the data
107             experiment.data = pd.DataFrame(data, columns=header)
108             # Read the references. References start with "# Reference
                   :\n" and end with "# END"
109             line = f.readline()
110             experiment.reference = ""
111             while not line.startswith("# END"):
112                 experiment.reference += line
113                 line = f.readline()             # In the last iteration,
                        the line is "# END"
114             # If the reference is empty, then set it to "". If not, then
                    remove the last "\n"
115             if experiment.reference != "": experiment.reference[:-1]
116             # Add the experiment to the list of experiments
117             experiments.append(experiment)
118             # Read the line "# END OF FILE" or next experiment
119             line = f.readline()
```

B.6 read_proton_experiments_from_exfortables()

```
def read_proton_experiments_from_exfortables(path):
    """
    Reads all proton experiment files located in a given directory and its subdirectories.

    The function recursively traverses through all directories and subdirectories under the specified path.
    It reads files that do not have names ending with "list" or "ruth" and returns a list of Experiment objects.

    Parameters:
    ------------
    path : str
        The absolute or relative path to the root directory containing proton experiment files.

    Returns:
    ---------
    experiments : list
        A list of Experiment objects containing data read from the files under the specified path.

    Raises:
    -------
    FileNotFoundError:
        If the specified path does not exist.
    ValueError:
        If the data in the files is not structured as expected.

    Example:
    --------
    experiments = read_proton_experiments_from_exfortables("./proton_experiment_data/")

    Notes:
    ------
    - The function assumes that each Experiment object has attributes corresponding to the fields in the text files.
    - The function relies on `read_experiment()` for reading individual experiment files.
    """

    # Get name of all directories in the path
    dirs = [d for d in os.listdir(path) if os.path.isdir(os.path.join(path, d))]

    experiments = []

    # Go through all directories
    for d in dirs:
        # Get name of all subdirectories in the path that dont end with "list"
        subdirs = [sd for sd in os.listdir(os.path.join(path, d)) if os.path.isdir(os.path.join(path, d, sd)) and not sd.endswith("list")]
        # Go through all subdirectories
```

```
45          for sd in subdirs:
46              # Check if there is any subsubdirectory inside the
                    subdirectory
47              contains_subsubdir = False
48              for item in os.scandir(os.path.join(path, d, sd)):
49                  if item.is_dir():
50                      contains_subsubdir = True
51                      break
52              # If there is no subsubdirectory, then the files are in the
                    subdirectory
53              if contains_subsubdir:
54                  # Get name of all subdirectories in the path
55                  subsubdirs = [ssd for ssd in os.listdir(os.path.join(
                        path, d, sd)) if os.path.isdir(os.path.join(path, d,
                        sd, ssd))]
56                  # Go through all subdirectories
57                  for ssd in subsubdirs:
58                      # Get the name of all the files which dont end with
                            "list" or "ruth"
59                      files = [f for f in os.listdir(os.path.join(path, d,
                            sd, ssd)) if os.path.isfile(os.path.join(path, d
                            , sd, ssd, f)) and not f.endswith("list") and not
                            f.endswith("ruth")]
60                      # Go through all files
61                      for f in files:
62                          # Read the experiment
63                          experiment = read_experiment(os.path.join(path,
                                d, sd, ssd, f))
64                          # Add the experiment to the list of experiments
65                          experiments.append(experiment)
66              else:
67                  # Get the name of all the files which dont end with "
                        list" or "ruth"
68                  files = [f for f in os.listdir(os.path.join(path, d, sd)
                        ) if os.path.isfile(os.path.join(path, d, sd, f)) and
                         not f.endswith("list") and not f.endswith("ruth")]
69                  # Go through all files
70                  for f in files:
71                      # Read the experiment
72                      experiment = read_experiment(os.path.join(path, d,
                            sd, f))
73                      # Add the experiment to the list of experiments
74                      experiments.append(experiment)
75      return experiments
```

B.7 filter_experiments()

```
def filter_experiments(experiments, attribute, value):
    """
    Filters a list of Experiment objects based on the specified attribute and its value.

    The function evaluates the attribute for each Experiment object in the list, keeping only
    those objects where the attribute matches the given value. If the attribute does not exist,
    it will return None and print a list of available attributes.

    Parameters:
    ------------
    experiments : list
        A list of Experiment objects to filter.
    attribute : str
        The attribute name based on which the filtering is to be done.
    value : str | int | float
        The value of the attribute for filtering the list of Experiment objects.

    Returns:
    ---------
    filtered_list : list | None
        A list of Experiment objects that have the specified attribute and value.
        Returns None if the attribute does not exist or no experiments meet the condition.

    Raises:
    -------
    AttributeError:
        If the attribute does not exist in the Experiment objects.

    Example:
    --------
    filtered_exp = filter_experiments(experiment_list, "year", 2021)

    Notes:
    ------
    - Make sure the attribute exists in the Experiment object.
    - This function is case-sensitive for string attributes.
    """

    # Check if the attribute exists
    if not hasattr(experiments[0], attribute):
        print("Attribute {} does not exist\n".format(attribute))
        # Print the list of available attributes
        print("Available attributes:")
        print([attr for attr in dir(experiments[0]) if not attr.startswith("__")])
        return None
    # Filter the list of experiments to get only the ones with the given attribute and value
    filtered_list = [obj for obj in experiments if getattr(obj, attribute) == value]
```

```
48
49      if len(filtered_list) == 0:
50          print("No experiments with {} = {}\n".format(attribute, value))
51          # Print the list of available values for the given attribute
52          print("Available values for {}:".format(attribute))
53          print(list(np.unique([getattr(obj, attribute) for obj in
                experiments])))
54          return None
55
56      else:
57          print("{} experiments with {} = {}".format(len(filtered_list),
                attribute, value))
58
59      # Return the filtered list
60      return filtered_list
```

B.8 get_unique_values()

```
def get_unique_values(experiments, attribute):
    """
    Retrieves the distinct values for a specified attribute from a list
       of Experiment objects.

    The function examines each Experiment object in the list to identify
        all the unique values
    of the specified attribute. If the attribute is not present, it
       returns None and displays
    a list of available attributes for the Experiment objects.

    Parameters:
    ------------
    experiments : list
        A list of Experiment objects from which to extract unique values
            of the attribute.
    attribute : str
        The name of the attribute whose unique values are to be
           determined.

    Returns:
    ---------
    unique_values : list | None
        A list of unique values for the given attribute.
        Returns None if the attribute does not exist within the
           Experiment objects.

    Raises:
    -------
    AttributeError:
        If the attribute does not exist in the Experiment objects.

    Example:
    --------
    unique_years = get_unique_values(experiment_list, "year")

    Notes:
    ------
    - Ensure the attribute exists in the Experiment objects for accurate
        results.
    - The function employs the numpy "unique" function for identifying
       unique values.
    """

    # Check if the attribute exists
    if not hasattr(experiments[0], attribute):
        print("Attribute {} does not exist\n".format(attribute))
        # Print the list of available attributes
        print("Available attributes:")
        print([attr for attr in dir(experiments[0]) if not attr.
           startswith("__")])
        return None
    # Get the unique values of the attribute
    unique_values = list(np.unique([getattr(obj, attribute) for obj in
       experiments]))
    # Return the unique values
```

```
47      return unique_values
```

B.9 clean_dataframe()

```
def clean_dataframe(df, uncertainties=False):
    """
    Cleans a DataFrame by removing redundant columns for analysis.

    This function eliminates all columns with only one unique value and,
    if the ‘uncertainties‘ flag is False, all columns starting with "d".

    Parameters:
    -----------
    df : pd.DataFrame
        The DataFrame to be cleaned.
    uncertainties : bool, optional
        Whether to retain columns that start with "d". Default is False.

    Returns:
    --------
    pd.DataFrame
        A DataFrame with the redundant columns removed.

    Example:
    --------
    cleaned_df = clean_dataframe(original_df)

    Notes:
    ------
    - The function relies on pandas for DataFrame operations.
    """
    # Drop the columns which heading starts with "d"
    if not uncertainties: df = df.drop([col for col in df.columns if col
        .startswith("d")], axis=1)

    # Get the number of unique values per column
    unique_values = df.apply(lambda x: len(x.unique()))
    # Get the columns with only one unique value
    columns_to_drop = unique_values[unique_values == 1].index
    # Drop the columns
    df = df.drop(columns_to_drop, axis=1)
    # Return the cleaned dataframe
    return df
```

B.10 classify_experiments()

```
def classify_experiments(experiments, attribute):
    """
    Classifies a list of Experiment objects based on a specified attribute.

    The function generates individual DataFrames for each unique value of the given attribute
    and writes these as CSV files. Finally, it returns a dictionary containing all these
    DataFrames.

    Parameters:
    -----------
    experiments : list
        A list of Experiment objects to be classified.
    attribute : str
        The attribute based on which the classification is performed.

    Returns:
    --------
    dfs : dict of pd.DataFrame
        A dictionary with keys as the unique values of the attribute and the corresponding
        DataFrames as values.

    Example:
    --------
    classified_dfs = classify_experiments(experiment_list, "year")

    Notes:
    ------
    - The function employs the `get_unique_values` function to identify unique attribute values.
    - The output DataFrames are written as CSV files in a directory corresponding to the attribute name.
    """
    # Get the unique values of the attribute
    unique_values = get_unique_values(experiments, attribute)

    # Create an empty dataframe per unique value of the attribute
    dfs = {}
    for value in unique_values:
        dfs[value] = pd.DataFrame()

    # Go through all experiments
    for experiment in experiments:
        # Get the value of the attribute
        value = getattr(experiment, attribute)
        # Add the experiment to the dataframe with the corresponding value of the attribute
        dfs[value] = pd.concat([dfs[value], experiment.to_dataframe()]).reset_index(drop=True)

    # Create a new csv file for each dataframe
    for value in unique_values:
```

```
48          dfs[value].to_csv("EXFOR_ProtonReactions_Classsified_by_{}/{}_
                {}.csv".format(attribute, attribute, value))
49
50      # Return the dataframe
51      return dfs
```

B.11 classify_experiments_by_data()

```
def classify_experiments_by_data(experiments):
    """
    Classifies a list of Experiment objects based on their column
       headers after data preparation.

    The function processes each Experiment object, extracts its
       dataframe representation using
    the 'prepare_data' method, and then groups these dataframes by their
        column headers.
    Each grouped dataframe is then saved as a CSV file. Finally, it
       prints the summary of the
    classification process.

    Parameters:
    -----------
    experiments : list
        A list of Experiment objects to be classified.

    Returns:
    --------
    None

    Example:
    --------
    classify_experiments_by_data(experiment_list)

    Notes:
    ------
    - Assumes each Experiment object has a 'prepare_data' method that
       returns a DataFrame.
    - The output grouped DataFrames are written as CSV files with a
       specific naming convention.
    """
    # Dictionary to store dataframes grouped by their column headers
    grouped_dataframes = {}

    total_experiments = len(experiments)
    print(f"Processing {total_experiments} experiments...")

    for idx, experiment in enumerate(experiments, start=1):
        print(f"Processing experiment {idx} out of {total_experiments
           }...")

        # Prepare the data
        df = experiment.prepare_data()

        # Create a key based on the column headers
        key = tuple(df.columns)

        # If the key already exists in the dictionary, concatenate the
           dataframe. Otherwise, create a new entry.
        if key in grouped_dataframes:
            grouped_dataframes[key] = pd.concat([grouped_dataframes[key
               ], df], axis=0)
        else:
            grouped_dataframes[key] = df
```

```
48
49      # Display the detected groups of headers on the screen
50      group_number = 1
51      for key in grouped_dataframes.keys():
52          print(f"\nGroup {group_number}:")
53          print(", ".join(key))
54          print("="*50)
55          group_number += 1
56
57      # Save each grouped dataframe to a CSV file
58      total_groups = len(grouped_dataframes)
59      print(f"\nSaving {total_groups} grouped dataframes to CSV files...")
60      for idx, (key, df) in enumerate(grouped_dataframes.items(), start=1)
            :
61          filename = f"EXFOR_ProtonReactions_Classified_Group_{idx}.csv"
62          df.to_csv(filename, index=False)
63          print(f"Group {idx}"s dataframe saved as {filename}")
64
65      print(f"\nFinished! {len(grouped_dataframes)} groups of experiments
            found based on column headers.")
```

B.12 plot_experiments()

```
def plot_experiments(experiments, xlog=False, ylog=False, fig_size=(9,6)
    ):
    """
    Plots a list of experiments using matplotlib and seaborn libraries.

    The function accepts a list of experiment objects and plots them on
       a 2D graph based on
    their data attributes. It can plot with or without error bars along
       the x and y axes.
    Additionally, it supports logarithmic scaling for both axes and
       allows figure size customization.

    Parameters:
    -----------
    experiments : list
        A list of experiment objects to be plotted. Each object should
           have a "data" attribute
        which is a pandas DataFrame containing the relevant data.
    xlog : bool, optional
        A flag to specify if the x-axis should be logarithmic. Default
           is False.
    ylog : bool, optional
        A flag to specify if the y-axis should be logarithmic. Default
           is False.
    fig_size : tuple, optional
        The dimensions of the figure to be plotted. Default is (9, 6).

    Returns:
    --------
    None :
        The function does not return anything; it generates and displays
            the plot.

    Example:
    --------
    plot_experiments([experiment1, experiment2], xlog=True, ylog=False,
       fig_size=(10,8))

    Notes:
    ------
    - The function first validates that all experiments have the same
       DataFrame columns.
    - The plotting accounts for possible error attributes in the x and y
        axes.
    - The resulting plot can be customized through optional parameters
       like `xlog`, `ylog`, and `fig_size`.
    """

    # First, check if there are experiments to plot
    if not experiments:
        print("No experiments to plot")
        return

    # Then, verify that all dataframes have the same headers
    headers = list(experiments[0].data.columns.values)
    for experiment in experiments[1:]:
```

```
        if list(experiment.data.columns.values) != headers:
            print("Mismatch in headers between experiments.")
            return

    # Now proceed to set up and plot each experiment
    plt.figure(figsize=fig_size)
    for experiment in experiments:
        data = experiment.data
        y_err = data.iloc[:,2].isnull().values.all() or data.iloc[:,2].
            eq(0).all()
        x_err = data.iloc[:,3].isnull().values.all() or data.iloc[:,3].
            eq(0).all()

        # Depending on y_err and x_err, we decide how to plot
        if y_err == False and x_err == False:
            plt.errorbar(x=data[headers[0]], y=data[headers[1]],
                         xerr=data[headers[3]], yerr=data[headers[2]],
                         fmt="o", ecolor="black", capsize=3, elinewidth
                            =1,  markersize=5, label=experiment.X4_ID)
        elif y_err == False:
            plt.errorbar(x=data[headers[0]], y=data[headers[1]], yerr=
                data[headers[2]],
                         fmt="o", ecolor="black", capsize=3, elinewidth
                            =1,  markersize=5, label=experiment.X4_ID)
        elif x_err == False:
            plt.errorbar(x=data[headers[0]], y=data[headers[1]], xerr=
                data[headers[3]],
                         fmt="o", ecolor="black", capsize=3, elinewidth
                            =1,  markersize=5, label=experiment.X4_ID)
        else:
            sns.scatterplot(x=headers[0], y=headers[1], data=data, label
                =experiment.X4_ID)

    # Set scale, labels, title, and display the plot
    if xlog: plt.xscale("log")
    if ylog: plt.yscale("log")

    plt.xlabel(headers[0], fontsize=16, labelpad=10)
    plt.ylabel(headers[1], fontsize=16, labelpad=10)
    plt.xticks(fontsize=14)
    plt.yticks(fontsize=14)
    plt.legend()
    plt.show()
```

B.13 plot_outliers()

```
def plot_outliers(outliers_df, experiments, xlog=False, ylog=False,
    fig_size=(9,6)):
    """
    Plots outliers along with their corresponding experiments using
        matplotlib and seaborn libraries.

    The function accepts a DataFrame containing outlier data and a list
        of experiment objects.
    It plots these outliers and the normal data on a 2D graph. Both
        outlier and normal data points
    can be visualized with or without error bars along the x and y axes.
         Logarithmic scaling for both axes
    and figure size customization are also supported.

    Parameters:
    -----------
    outliers_df : pd.DataFrame
        A DataFrame containing the outlier data points. Should have an "
            X4_ID" column which matches the experiment IDs.
    experiments : list
        A list of experiment objects to be plotted. Each object should
            have a "data" attribute which is a pandas DataFrame.
    xlog : bool, optional
        A flag to specify if the x-axis should be logarithmic. Default
            is False.
    ylog : bool, optional
        A flag to specify if the y-axis should be logarithmic. Default
            is False.
    fig_size : tuple, optional
        The dimensions of the figure to be plotted. Default is (9, 6).

    Returns:
    --------
    None :
        The function does not return anything; it generates and displays
             the plot.

    Example:
    --------
    plot_outliers(outliers_dataframe, [experiment1, experiment2], xlog=
        True, ylog=False, fig_size=(10,8))

    Notes:
    ------
    - The function first identifies the DataFrame columns that are
        specific to outliers, not present in normal data.
    - It then groups outliers based on these specific columns before
        plotting.
    - Normal data and outliers from the same experiment are plotted
        together for better visualization.
    """

    # Get the column names from the "data" attribute of a corresponding
        experiment
    example_experiment = next(experiment for experiment in experiments
        if experiment.X4_ID in outliers_df["X4_ID"].tolist())
```

```
    data_columns = example_experiment.data.columns.values.tolist()

    # Get columns from outliers_df that are not in "data" and are also not "X4_ID"
    groupby_columns = [col for col in outliers_df.columns if col not in data_columns and col != "X4_ID"]

    grouped_outliers = outliers_df.groupby(groupby_columns)

    print("Grouped outliers according to columns: {}".format(groupby_columns))

    for _, group in grouped_outliers:
        plt.figure(figsize=fig_size)
        x4_ids = group["X4_ID"].tolist()

        # Add a legend entry for the outliers without actually plotting them
        plt.scatter([], [], color="red", s=50, label="Outliers", marker="x")

        for experiment in experiments:
            if experiment.X4_ID in x4_ids:
                data = experiment.data
                headers = list(data.columns.values)

                y_err = not (data.iloc[:,2].isnull().values.all() or data.iloc[:,2].eq(0).all())
                x_err = not (data.iloc[:,3].isnull().values.all() or data.iloc[:,3].eq(0).all())

                # Plot normal data
                if y_err and x_err:
                    plt.errorbar(x=data[headers[0]], y=data[headers[1]],
                                 xerr=data[headers[3]], yerr=data[headers[2]],
                                 fmt="o", ecolor="black", capsize=3, elinewidth=1,  markersize=5, label=experiment.X4_ID)
                elif y_err:
                    plt.errorbar(x=data[headers[0]], y=data[headers[1]],
                                 yerr=data[headers[2]],
                                 fmt="o", ecolor="black", capsize=3, elinewidth=1,  markersize=5, label=experiment.X4_ID)
                elif x_err:
                    plt.errorbar(x=data[headers[0]], y=data[headers[1]],
                                 xerr=data[headers[3]],
                                 fmt="o", ecolor="black", capsize=3, elinewidth=1,  markersize=5, label=experiment.X4_ID)
                else:
                    plt.scatter(x=data[headers[0]], y=data[headers[1]],
                                s=25, label=experiment.X4_ID)

        # Actually plot the outliers
        for experiment in experiments:
            if experiment.X4_ID in x4_ids:
```

```
                specific_outliers = outliers_df[outliers_df["X4_ID"] ==
                    experiment.X4_ID]
                plt.scatter(specific_outliers[headers[0]],
                    specific_outliers[headers[1]], color="red", s=50,
                    zorder=3, marker="x")

        if xlog: plt.xscale("log")
        if ylog: plt.yscale("log")
        plt.xlabel(headers[0], fontsize=16, labelpad=10)
        plt.ylabel(headers[1], fontsize=16, labelpad=10)
        plt.xticks(fontsize=14)
        plt.yticks(fontsize=14)
        plt.legend()
        plt.show()
```

C Interquartile Range Functions

C.1 detect_outliers_IQR()

```
def detect_outliers_IQR(grouped_df, experiment, limit=1.5, uncertainties=False):
    """
    Identifies outliers in the data of an Experiment object using the Interquartile Range (IQR) method.

    The function takes a grouped dataframe and an Experiment object, then calculates the first (Q1) and third (Q3) quartiles
    along with the interquartile range (IQR) for specified columns. Outliers are determined based on these statistics.

    Parameters:
    -----------
    grouped_df : DataFrame
        The grouped dataframe containing the data.
    experiment : object
        An Experiment object containing the data and metadata.
    limit : float, optional
        The limit factor to multiply with the IQR to determine the range for outliers (default is 1.5).
    uncertainties : bool, optional
        Whether to consider uncertainties in the data (default is False).

    Returns:
    --------
    DataFrame
        A dataframe containing only the outliers.

    Example:
    --------
    detect_outliers_IQR(grouped_df, experiment_object)

    Notes:
    ------
    - Assumes the Experiment object contains a dataframe representation in its 'data' attribute.
    - If uncertainties are considered, it assumes the Experiment object"s dataframe includes columns for uncertainties.
    """

    if not uncertainties:
        # Get the names of the first two columns
        if experiment.data.columns[0] != "Z":
            first_column = experiment.data.columns[0]
            second_column = experiment.data.columns[1]
        else:
            first_column = experiment.data.columns[2]
            second_column = experiment.data.columns[3]

        Q1 = grouped_df[[first_column, second_column]].quantile(0.25)
        Q3 = grouped_df[[first_column, second_column]].quantile(0.75)
        IQR = Q3 - Q1
```

```
46
47          # Here is where we define the condition for outliers
48          outlier_condition = ((grouped_df[[first_column, second_column]]
                < (Q1 - limit * IQR)) |
49                              (grouped_df[[first_column, second_column]] >
                                    (Q3 + limit * IQR)))
50
51          return grouped_df[outlier_condition.any(axis=1)]
52
53      else:
54
55          if experiment.data.columns[0] != "Z":
56              first_column = experiment.data.columns[0]
57              second_column = experiment.data.columns[1]
58              if experiment.data[experiment.data.columns[2]].isnull().
                    values.any():
59                  Q1 = grouped_df[second_column].quantile(0.25)
60                  Q3 = grouped_df[second_column].quantile(0.75)
61                  IQR = Q3 - Q1
62
63                  # Here is where we define the condition for outliers
64                  outlier_condition = ((grouped_df[second_column] < (Q1 -
                        limit * IQR)) |
65                                       (grouped_df[second_column] > (Q3 +
                                           limit * IQR)))
66
67                  return grouped_df[outlier_condition]
68              else:
69                  third_column = experiment.data.columns[2]
70          else:
71              first_column = experiment.data.columns[2]
72              second_column = experiment.data.columns[3]
73              third_column = experiment.data.columns[4]
74
75      # Calculate the bounds using IQR
76      Q1 = grouped_df[second_column].quantile(0.25)
77      Q3 = grouped_df[second_column].quantile(0.75)
78      IQR = Q3 - Q1
79
80      # Convert lower_bound and upper_bound to Series
81      lower_bound = (Q1 - limit * IQR)
82      upper_bound = (Q3 + limit * IQR)
83
84      # Check if both ends of the uncertainty range are outside the bounds
85      lower_values = grouped_df[second_column] - grouped_df[third_column]
86      upper_values = grouped_df[second_column] + grouped_df[third_column]
87
88      # Check if either end of the uncertainty range crosses the bounds
89      outliers_condition = ((lower_values < lower_bound) & (upper_values <
            lower_bound)) | ((upper_values > upper_bound) & (lower_values >
           upper_bound))
90
91      # Return the points that meet the condition of being outliers
92      return grouped_df[outliers_condition]
```

C.2 IQR_method()

```
def IQR_method(df, experiments, min_observations=20, uncertainties=False):
    """
    Applies the IQR method to identify outliers from a DataFrame using data from a list of Experiment objects.

    The function first identifies columns by which to group the data. It then filters the groups based on the
    minimum number of observations specified. For each valid group, it applies the ‘detect_outliers_IQR‘ function
    to identify outliers.

    Parameters:
    -----------
    df : DataFrame
        The input DataFrame containing the data to be checked for outliers.
    experiments : list
        A list of Experiment objects containing relevant data and metadata.
    min_observations : int, optional
        The minimum number of observations required to consider a group for outlier detection (default is 20).
    uncertainties : bool, optional
        Whether to consider uncertainties in the data (default is False).

    Returns:
    --------
    DataFrame
        A DataFrame containing only the outliers.

    Example:
    --------
    IQR_method(df, experiment_list)

    Notes:
    ------
    - Assumes the Experiment object contains a dataframe representation in its ‘data‘ attribute.
    - If uncertainties are considered, it assumes the Experiment object"s dataframe includes columns for uncertainties.
    """

    # Get the name of the "data" columns from a corresponding experiment
    example_exp = next(experiment for experiment in experiments if experiment.X4_ID in df["X4_ID"].tolist())
    data_columns = example_exp.data.columns.values.tolist()

    # Get columns from outliers_df that are not in data and also not X4_ID
    groupby_columns = [col for col in df.columns if col not in data_columns and col != "X4_ID"]

    # Group by columns and perform initial size count
```

```
    grouped = df.groupby(groupby_columns).size().reset_index(name="count")

    # Filter groups by minimum number of observations
    valid_groups = grouped[grouped["count"] >= min_observations][groupby_columns].to_dict("records")
    filtered_df = df[df.set_index(groupby_columns).index.isin([tuple(d.values()) for d in valid_groups])]

    print("Number of groups with at least {} observations: {}\n".format(min_observations, len(valid_groups)))

    print("Calculating outliers {} uncertainties...\n".format("with" if uncertainties else "without"))

    # Detect outliers
    outliers_df = (filtered_df.groupby(groupby_columns, group_keys=False)
                  .apply(lambda x: detect_outliers_IQR(x, example_exp, uncertainties=uncertainties))
                  .reset_index(drop=True))

    print("Percentage of outliers: {:.2f}%".format(len(outliers_df) / len(filtered_df) * 100))

    return outliers_df
```

D Classification Data Groups

In the development of a machine learning application for analyzing proton-induced reactions from the EXFOR library, data is classified into eight distinct groups. Each group is characterized by specific measured parameters that serve as features for the machine learning model. The classification is as follows:

- **Group 1: Differential Cross-Section by Angle**
  - **Angle(deg)**: The angle in degrees at which the reaction occurs.
  - **xs(mb/sr)**: The differential cross-section in millibarns per steradian.
  - **dxs(mb/sr)**: The uncertainty in the differential cross-section.
  - **dAngle(deg)**: The uncertainty in the angle.
- **Group 2: Energy Outgoing and Cross-Section**
  - **E-out(MeV)**: The outgoing energy in mega electronvolts.
  - **xs(mb/MeV.sr)**: The differential cross-section in millibarns per MeV per steradian.
  - **dxs(mb/MeV.sr)**: The uncertainty in the differential cross-section.
  - **dE(MeV)**: The uncertainty in the outgoing energy.
- **Group 3: Energy and Ratio**
  - **E(MeV)**: The energy in mega electronvolts.
  - **ratio**: The ratio of the observed value to the expected value.
  - **dratio**: The uncertainty in the ratio.
  - **dE(MeV)**: The uncertainty in the energy.
- **Group 4: Energy and Total Cross-Section**
  - **E(MeV)**: The energy in mega electronvolts.
  - **xs(mb)**: The total cross-section in millibarns.
  - **dxs(mb)**: The uncertainty in the total cross-section.
  - **dE(MeV)**: The uncertainty in the energy.
- **Group 5: Energy Outgoing and Cross-Section per MeV**
  - **E-out(MeV)**: The outgoing energy in mega electronvolts.
  - **xs(mb/MeV)**: The differential cross-section in millibarns per MeV.
  - **dxs(mb/MeV)**: The uncertainty in the differential cross-section.
  - **dE(MeV)**: The uncertainty in the outgoing energy.
- **Group 6: Energy in eV and Parity**
  - **E(eV)**: The energy in electronvolts.
  - **par**: Refers to a parameter, such as "resonance width" (WID).
  - **dpar**: The uncertainty in the parameter.
  - **dE(eV)**: The uncertainty in the energy.
- **Group 7: Atomic and Mass Numbers with Yield**

  - **Z**: The atomic number.
  - **A**: The mass number.
  - **Yield**: The yield of the reaction.
  - **dYield**: The uncertainty in the yield.

- **Group 8: Energy and Average Number of Neutrons Emitted**
  - **E(MeV)**: The energy in mega electronvolts.
  - **nubar**: The average number of neutrons emitted.
  - **dnubar**: The uncertainty in the average number of neutrons emitted.
  - **dE(MeV)**: The uncertainty in the energy.

Each of these groups serves a specific purpose in the analysis and provides a unique set of features for the machine learning model to learn from. By classifying the data into these groups, the model can be more effectively trained to make accurate predictions for various types of proton-induced reactions.

POLITÉCNICA

Escuela Técnica Superior de Ingenieros Industriales
Universidad Politécnica de Madrid

José Gutiérrez Abascal, 2. 28006 Madrid
Tel.: 91 336 3060
info.industriales@upm.es

www.industriales.upm.es