**TECHNICAL COMPUTING**

**DISSERTATION RESEARCH PROJECT PROPOSAL 2024**

**Developing an explainable deep learning model for 3D drug-drug interaction prediction**

**Marcus Abraham**

**S4203822**

**Zainab Loukil**

**24-04-2024**

**Table of** **Contents**

[**Acknowledgments** 6](#_Toc164694484)

[**Ethical Issues** 7](#_Toc164694485)

[**Table of Abbreviations** 8](#_Toc164694486)

[**Abstract** 9](#_Toc164694487)

[**Background** 10](#_Toc164694488)

[**Chapter 1 - Introduction** 11](#_Toc164694489)

[**1.1 Overview** 11](#_Toc164694490)

[**1.2 Aim** 11](#_Toc164694491)

[**1.3 Problem Statement** 11](#_Toc164694492)

[**1.4 Research Objectives** 11](#_Toc164694493)

[**1.5 Research Questions** 12](#_Toc164694494)

[**1.6 Scope** 12](#_Toc164694495)

[**1.7 Conclusion** 12](#_Toc164694496)

[**Chapter 2 - Literature Review** 13](#_Toc164694497)

[**2.1 Introduction** 13](#_Toc164694498)

[**2.2 Brief History** 13](#_Toc164694499)

[**2.3 Early Machine Learning Research** 14](#_Toc164694500)

[**2.4 Deep Learning Research** 15](#_Toc164694501)

[**2.5 Limits of Current Models** 16](#_Toc164694502)

[**2.6 Explainable Methods** 16](#_Toc164694503)

[**2.7 Identified Gaps** 18](#_Toc164694504)

[**2.8 Conclusion** 19](#_Toc164694505)

[**Chapter 3 – Research Methodology** 20](#_Toc164694506)

[**3.1 Introduction** 20](#_Toc164694507)

[**3.2 Overview** 20](#_Toc164694508)

[**3.3 Dataset Collection** 20](#_Toc164694509)

[**3.4 Dataset Pre-processing** 22](#_Toc164694510)

[**3.5 Model/Explainable Techniques** 23](#_Toc164694511)

[**3.6 Benchmarking** 24](#_Toc164694512)

[**3.7 Conclusion** 25](#_Toc164694513)

[**Chapter 4 - Implementation** 26](#_Toc164694514)

[**4.1 Introduction** 26](#_Toc164694515)

[**4.2 Data Gathering & Analysis** 26](#_Toc164694516)

[**4.3 Data Pre-Processing** 30](#_Toc164694517)

[**4.4 3D-DDI-GCN Model** 34](#_Toc164694518)

[**4.5 User Presentation** 38](#_Toc164694519)

[**4.6 Conclusion** 41](#_Toc164694520)

[**Chapter 5 – Benchmarking** 42](#_Toc164694521)

[**5.1 Introduction** 42](#_Toc164694522)

[**5.2 LSTM Model** 42](#_Toc164694523)

[**5.3 2D-GCN Model** 43](#_Toc164694524)

[**5.1 Conclusion** 47](#_Toc164694525)

[**Chapter 6 – Analysis** 48](#_Toc164694526)

[**6.1 Introduction** 48](#_Toc164694527)

[**6.2 LSTM Results** 48](#_Toc164694528)

[**6.2 2D-GCN Results** 49](#_Toc164694529)

[**6.3 3D-DDI-GCN** 51](#_Toc164694530)

[**6.4 Conclusion** 54](#_Toc164694531)

[**Chapter 7 – Discussion and Conclusion** 55](#_Toc164694532)

[**References** 56](#_Toc164694533)

[**Appendix** 58](#_Toc164694534)

[**A – Literature Review Table** 58](#_Toc164694535)

[**B – Data Gathering and Cleaning Code** 63](#_Toc164694536)

[**data\_gathering.py** 63](#_Toc164694537)

[**descriptive\_analytics.py** 68](#_Toc164694538)

[**C – Model Code** 70](#_Toc164694539)

[**graph\_creation.py** 70](#_Toc164694540)

[**gnn\_3d.py** 72](#_Toc164694541)

[**visualise\_3d.py** 81](#_Toc164694542)

[**lstm.py** 82](#_Toc164694543)

[**gnn\_smiles\_dummy.py** 83](#_Toc164694544)

[**gnn\_smiles.py** 85](#_Toc164694545)

**List of Figures**

[Figure 1: XSMILES Visual Representation 17](#_Toc164790018)

[Figure 2: DDI-GCN Model View 19](#_Toc164790019)

[Figure 3: Methodology Overview 20](#_Toc164790020)

[Figure 4: Dataset Collection Methodology 21](#_Toc164790021)

[Figure 5: Dataset Pre-processing Methodology 22](#_Toc164790022)

[Figure 6: Model Architecture Diagram 23](#_Toc164790023)

[Figure 7: Benchmarking Methodology 24](#_Toc164790024)

[Figure 8: API Request Reduction Code 26](#_Toc164790025)

[Figure 9: Data Gathering Test Code 27](#_Toc164790026)

[Figure 10: Data Gathering Test Result 27](#_Toc164790027)

[Figure 11: Data Gathering Code 27](#_Toc164790028)

[Figure 12: Euclidian Distance Code 28](#_Toc164790029)

[Figure 13: Descriptive Analytics Code 29](#_Toc164790030)

[Figure 14: Descriptive Analytics Result 29](#_Toc164790031)

[Figure 15: Graph Creation Code 30](#_Toc164790032)

[Figure 16: Atom Charge Code 30](#_Toc164790033)

[Figure 17: Atom Bond Code 31](#_Toc164790034)

[Figure 18: Graph Data Concatenation Code 32](#_Toc164790035)

[Figure 20: 3D-DDI-GCN Model Code 34](#_Toc164790036)

[Figure 21: Model Initialisation Code 35](#_Toc164790037)

[Figure 22: Model Training Code 36](#_Toc164790038)

[Figure 23: Model Evaluation Code 36](#_Toc164790039)

[Figure 24: Model GUI 37](#_Toc164790040)

[Figure 25: GUI Code 37](#_Toc164790041)

[Figure 26: Prediction Code 38](#_Toc164790042)

[Figure 27: GUI Error Checking 38](#_Toc164790043)

[Figure 28: GUI Prediction 39](#_Toc164790044)

[Figure 30: LSTM SMILES Concatenation Code 41](#_Toc164790045)

[Figure 31: LSTM Model Code 42](#_Toc164790046)

[Figure 32: 2D-GCN Model Code 43](#_Toc164790047)

[Figure 33: 2D-GCN Dummy Data Code 43](#_Toc164790048)

[Figure 34: 2D-GCN SMILES To Graph Code 44](#_Toc164790049)

[Figure 35: 2D-GCN Data Creation 44](#_Toc164790050)

[Figure 36: 2D-GCN Training Code 45](#_Toc164790051)

[Figure 37: 2D-GCN Evaluation Code 45](#_Toc164790052)

[Figure 38: LSTM Loss/Accuracy 47](#_Toc164790053)

[Figure 40: LSTM Confusion Matrix 47](#_Toc164790054)

[Figure 41: 2D-GCN Accuracy/Loss 49](#_Toc164790055)

[Figure 42: 2D-GCN Dummy Data Comparison 49](#_Toc164790056)

[Figure 43: 2D-GCN Precision/Recall/F1 50](#_Toc164790057)

[Figure 44: 3D-DDI-GCN Accuracy/Loss 50](#_Toc164790058)

[Figure 45: 3D-DDI-GCN Precision/Recall/F1 51](#_Toc164790059)

[Figure 46: 3D-DDI-GCN Visualisation Before Training 52](#_Toc164790060)

[Figure 47: 3D-DDI-GCN Visualisation After Training 52](#_Toc164790061)

**List of Tables**

[Table 1: Table of Abbreviations 8](#_Toc164686180)

[Table 2: Literature Review Table 61](#_Toc164686181)

# **Acknowledgments**

I want to thank my dissertation supervisor, Dr. Zainab Loukil, for her guidance and support throughout the dissertation research process.

I would also like to thank my friends and partner for supporting me for the duration of this academic year.

Credit is also due to the writers of the DDI-GCN model (Zhong *et al.*, 2023), whose suggestions for future work inspired this dissertation.

# **Ethical Issues**

I confirm that I have submitted my ethical form based on the timeline provided in the module and received approval from my supervisor / university.

I declare that this dissertation has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree or anywhere else. Except where states otherwise by citation and reference or acknowledgment, the work presented is entirely my own.

I confirm that all the Tables and Figures in this dissertation are my own works or a regeneration from other people’s work with citation.

I confirm that all the tools, software and datasets have been used in this dissertation followed the terms and conditions in their license agreement and university REC code of conduct.

I confirm that all the citations in this dissertation have been listed in the bibliography and they are all accessible. In case that university wants to cross examine the citations, I will provide the content for the references which are not accessible.

I confirm in this dissertation no human/animal study has been conducted.

# **Table of Abbreviations**

|  |  |  |
| --- | --- | --- |
| Abbreviation | Term | Definition |
| ADR | Adverse Drug Reaction | An unexpected result or side effect that results from a drug or interaction involving drugs. |
| DDI | Drug-Drug Interaction | Interaction between two or more drugs, which can lead to ADRs. |
| GUI | Graphical User Interface | An interface with graphical components. |
| SMILES | Simplified Molecular-Input Line-Entry System | A form of notation for representing chemical compounds. |
| SVM | Support Vector Machines | A supervised machine learning algorithm. |
| RF | Random Forest | An ensemble machine learning algorithm. |
| GCN | Graph Convolutional Network | A type of neural network, used for graph data. |
| RNN | Recurrent Neural Network | A type of neural network, used for sequential and time series data. |
| LSTM | Long Short-Term Memory (LSTM) | A type of recurrent neural network that was designed to address problems with traditional RNNs. |
| DoX | Degree of Explainability | A proposed metric for measuring explainability in deep learning models, referenced in the literature review (Sovrano and Vitali, 2023). |
| GCNN-BiLSTM | Graph Convolutional Neural Network - Bi-Directional Long Short-Term Memory | A hybrid deep learning model that combines a GCN and LSTM, created by Lee and Chen (2021a). |
| DDI-GCN | Drug-Drug Interaction Graph Convolutional Network | An explainable deep learning model created by Zhong *et al.*, (2023). |
| 3D-DDI-GCN | 3-Dimensional Drug-Drug Interaction Graph Convolutional Network | The model that is proposed in the methodology section. |

Table 1: Table of Abbreviations

# **Abstract**

Prediction of adverse reactions in drugs continues to be a challenge in both research and practical settings. The application of deep learning models allows for a higher degree of accuracy when predicting these reactions, however, they lack the ability to explain their predictions, reducing willingness to adopt these models in situations where human lives rely on the results. This dissertation focused on improving upon an existing explainable deep learning model, proposing a method of representing chemical compounds with their 3D properties, and visualising the results after in an explainable way. 3D data from the Decagon and PubChem datasets were used, with cleaning and pre-processing performed to turn the data into graph representations. The graphs were given to a GCN model, 3D-DDI-GCN, which had the goal of predicting side effects for drug pairs and visualising the most important sub-structures within each drug that led to that prediction. Two other models were used for benchmarking, an LSTM and a regular GCN. The 3D-DDI-GCN model showed potential, however ultimately lacked the large dataset required, and suffered from underfitting problems. Future work should focus on improving the data given to the model and expanding on scope to incorporate interactions between more than just single pairs of drugs.

# **Background**

The following section details a brief overview of the terms, models and measurements used in this dissertation. All abbreviations are defined here, and in subsequent chapters when they are used for the first time. In addition to this, any abbreviations used can be found with their full name and definition in *Table 1: Table of Abbreviations*.

The literature reviews main focus is on the prediction of Adverse Drug Reactions (ADRs). This term can sometimes be referred to as side effects. However it is important to note that, while all ADRs are side effects, not all side effects are ADRs. ADRs can be defined as any harmful or unwanted effect of a drug, while a side effect could be unintended but beneficial.

While discussing the approaches towards ADR prediction, the topic of polypharmacy and Drug-Drug Interaction (DDI) is then focused on. Polypharmacy is the taking of two or more drugs at the same time, and DDI is the interaction of two or more drugs that results in a change in a drug’s effectiveness or adverse reactions.

During the data gathering process, the euclidian distance formula was used, mathematically defined as *d = √(x1-x0)2 + (y1-y0) 2 +(z1-z0) 2*. Euclidian distance can be defined as the straight line distance between two points in space (Suwanda, Syahputra and Zamzami, 2020, p. 3).

Throughout the methodology and implementation, both Long Short Term Memory (LSTM) models and Graph Convolutional Network (GCN) models were utilised. These are both deep learning models, and in the context of this report are used for the task of DDI classification. To measure their performance, several metrics including accuracy, precision, recall, f1score and confusion matrices were used. These are all methods of assessing the performance of the previously mentioned models, and can provide insights into their behaviour.

# **Chapter 1 - Introduction**

## **1.1 Overview**

This research endeavours to address the challenge of enhancing the explainability of deep learning algorithms in the field of drug-drug interaction (DDI) prediction, to increase trust in models, and enable wider adoption by healthcare practitioners. Explainability allows for deep learning models to explain their predictions, which aids in debugging and understanding them. This will reduce risk to human life, and costs for pharmaceutical companies performing clinical trials.

## **1.2 Aim**

The aim is to develop an accurate deep-learning model for drug-drug interaction prediction while emphasizing explainability in the results. This entails leveraging both the predictive power of deep learning algorithms and explainable AI methods to enhance the model’s explainability.

## **1.3 Problem Statement**

How to increase the explainability of deep learning algorithms for drug-drug interaction prediction, to improve reliability and reduce patient risk?

## **1.4 Research Objectives**

1) To investigate current deep learning approaches for drug-drug interaction prediction.

2) To develop an explainable machine learning model for drug-drug interaction prediction.

3) To evaluate the developed model against other models.

4) To validate the developed model on unseen data.

## **1.5 Research Questions**

1) What are the current deep learning models used for drug-drug interaction prediction?

2) How can deep learning models be made more explainable?

3) What are the best methods of measuring accuracy and explainability in deep learning models?

4) What existing models would be most relevant for benchmarking?

5) How can a reduced accuracy from explainable methods be accounted for?

6) How is the proposed model’s performance affected by unseen data?

## **1.6 Scope**

This research will primarily focus on developing a deep-learning model that emphasizes explainability in its results. This will be achieved by combining deep learning and explainable methods, allowing the model to explain its predictions. Multiple data types will be used to train the model, including chemical structure, and chemical properties (charge, molecular weight, no. bonds, atom count, etc).

For the purpose of creating a working proof of concept, predictions will be limited to drug-drug interactions (interactions between drug pairs), and will not include prediction of side effects from interaction between more than two drugs. A user-friendly interface and visualization tools will present the model's predictions, aiming to make it understandable to both researchers and healthcare practitioners. The model’s performance will be evaluated using appropriate metrics, as well as comparison to traditional methods, and other case studies found in the literature review.

## **1.7 Conclusion**

Through these components, this research aims to contribute to the advancement of drug-drug side effect prediction by creating a model that combines deep learning predictive capabilities with a high degree of explainability. Before this model could be created, a thorough review of existing literature was undertaken to understand the current landscape of drug side effect prediction, which is detailed in the literature review section.

# **Chapter 2 - Literature Review**

## **2.1 Introduction**

The proceeding chapter examines the literature surrounding Adverse Drug Reactions (ADRs). Context is given as to the history of ADRs, as well as some of the science surrounding them. Early machine learning approaches towards addressing the issue of ADR prediction are explored, highlighting their limitations. Deep learning approaches are then explored, showing more recent work in the field. The limits of these deep learning approaches are identified and summarised, to be addressed in the methodology section.

## **2.2 Brief History**

To understand the current landscape of the pharmaceutical industry, it is useful to first look at the past. When discussing side effects and adverse reactions from drugs, the most notable example in history is the Thalidomide incident that occurred in the late 1950s to early 1960s. Initially approved and marketed as anti-nausea medication, pregnant women who had taken the drug gave birth to children with severe birth defects (Vargesson and Stephens, 2021, p. 1455). Less severe side effects were also noted, consisting of rash, fatigue, and constipation. This was a turning point in the pharmaceutical industry, causing clinical trials to become stricter, and testing for ADRs more thoroughly.

Regardless of stricter regulations, traditional methods of detecting ADRs fail to provide a fully comprehensive solution alone. Methods such as post-marketing surveillance and adverse event reporting help to identify ADRs, however, these are reactive solutions which only provide this information after potentially causing significant damage to large groups of patients (Yang and Kar, 2023, p. 1).

The collection of adverse drug reaction data into organised databases opened up the possibility for deeper data analysis, and eventually machine learning, to be applied. The more information that is gathered about drugs properties and interactions, the more we understand them. Databases such as SIDER provide information on drug-ADR pairs, gathered from drug labels, academic research, and post-marketing surveillance (Kuhn *et al.*, 2016, p. 1). This data ranges from specific information about the drugs to the frequency and severity of their ADRs.

When trying to predict ADRs, multiple factors may be considered. These factors can be categorised into one of four categories, patient, social, drug, or disease (Alomar, 2013, p. 85). Patient factors include age, gender, weight, genetics, and allergies. These can be some of the hardest to gather large datasets for, often leading to them being left out of generalised predictive models. The same can be said for social factors, such as drinking or smoking. The most common factors used to predict ADRs are drug-related factors, such as dose or frequency of use. Polypharmacy, taking multiple drugs together, is a significant drug-related factor that contributes to adverse reactions. This is also referred to as drug-drug interaction (DDI). Disease factors relate to when drugs are used to treat one disease and worsen another that was already present.

## **2.3 Early Machine Learning Research**

Early usage of machine learning for ADR prediction was focused on the interaction of single drugs with existing protein structures within the body. This allowed researchers to create drugs that avoided interaction with those proteins. One such study found that Support Vector Machines (SVM’s) could identify and associate ADR-related protein structures with a high degree of accuracy, correctly classifying 93.9% of the proteins related to known ADRs (Ji *et al.*, 2012, p. 319). It should be noted however that the data used was limited in its diversity, and the model therefore would not generalise well to new datasets or less common ADR-related protein structures. In addition to this, as it has been established, while protein interaction can cause ADRs they are not the only factor involved. Research into protein structures paved the way to exploring other factors, primarily drug-related, as they had the largest quantity of data available.

As the field of ADR prediction grew, the next logical step was to begin incorporating multiple types of data and ADR factors into machine learning models. Researchers creating models that utilised chemical structure and biological information such as protein targets noted that they were able to achieve a significantly higher accuracy when incorporating multiple types of data in comparison to only using chemical structures (Liu *et al.*, 2012, p. 32). A comparison of five models (logistic regression, naïve Bayes, K-nearest neighbor, Random Forest, and SVM) with fivefold cross-validation revealed better performance by SVM and Random Forest (RF) across all metrics. The results achieved in this study prove that using a hybrid fusion of multiple data types helps increase accuracy when predicting ADRs.

Some studies chose to focus on patient-related factors, such as Valeanu *et al* (2020, p. 1), who created a patient-tailored model. This allowed patients to enter information about themselves, and the drugs they are taking, and a report would be generated to highlight their likelihood of experiencing ADRs. The report included a severity profile that assessed the risk to the patient’s health, as well as a list of probabilities for each ADR they may experience. This model is especially significant, because in addition to patient factors it considers polypharmacy, a major factor that causes ADRs, and one that has limited data and studies that focus on it. Perhaps the most notable part of the model, it was presented in a web application, with a user-friendly graphical user interface (GUI). This is less commonly seen and focused on in studies that aim to create predictive models for ADRs, even in the present day. However for models to be adopted by pharmaceutical and medical practitioners this is extremely important, as it promotes trust in the model.

## **2.4 Deep Learning Research**

When deep learning began to be applied to ADR prediction, it significantly improved accuracy and opened up new possibilities for research into ADRs. Deep learning models can handle larger datasets much better than traditional machine learning methods (Dubey and Pandit, 2022, p. 211), and the complexity of the model’s understanding helped improve our understanding of how side effects are created. They also provide the benefit of generalising to unseen data and handling non-linear relationships.

Studies have found that, in comparison to non-machine learning statistical methods, models that out-perform the statistical methods tend to use a wider combination of datasets and sources (Kim *et al.*, 2022, p. 12). This is to be expected, given the potential causes of ADRs are numerous. It is therefore reasonable to ascertain that, as the complexity of data grows, the complexity of the models required to learn their underlying patterns does as well.

One method of improving a model’s accuracy when analysing multiple sources of data is to implement hybrid models. This allows for multiple types of models to work together, compensating for each other’s weaknesses. It has been proven that this leads to increased accuracy, as well as being more robust to noise in the data. GCNN-BiLSTM (Lee and Chen, 2021a), the first hybrid deep learning model for ADR prediction, was able to achieve high precision despite a small dataset. It analysed chemical structure in the form of Simplified Molecular Input Line Entry System (SMILES) strings, which represent the atoms and bonds of the chemical compound. Additionally, GCNN-BiLSTM showed the ability to output descriptive language about the side effects it was predicting, which is uncommon among other models. One approach which utilised attention mechanisms with an LSTM concluded that its accuracy could be further improved in combination with other models which analyse drug structure, further supporting the validity of hybrid models which use multiple types of data (Qian *et al.*, 2023, p. 4886).

The argument has however been made that while multiple sources of data provide a more holistic view of the causes of an ADR, new drugs may not yet have all of the recorded information required for complex models to make a prediction (Saifuddin *et al.*, 2023, p. 2). It is therefore worth considering the application of these models when creating them, whether for further research into novel compounds or ADR profiling of known compounds.

The Decagon method defined a Graph Convolutional Network (GCN) based approach to drug-drug interaction (DDI) prediction for ADRs, which resulted in the gathering of a polypharmacy dataset, which is valuable for future research (Zitnik, Agrawal and Leskovec, 2018). This also introduces the idea of modelling chemical compounds with graph data structures, combined with deep learning models which take graph input. Further validation of the Decagon polypharmacy datasets usefulness for DDI prediction, and the use of GCNs for DDI problems, has been shown in more recent research (Hong, Jeon and Kim, 2023, pp. 277–283) that looked to identify the techniques with the most potential for polypharmacy prediction.

Given the complexity of models required for DDI prediction, and the limited datasets available, there is a significant need for more research into the field (Lee and Chen, 2021b, p. 1892). Studies into ADR prediction focus mostly on single drug-ADR connections, limiting the amount of useful data and past work to build upon (Nguyen, Nguyen and Mamitsuka, 2021, p. 174). This need for DDI research is applicable both for actively used compounds and developing ones, as DDI-related side effects are commonly known to be one of the main reasons for failure in drug development (Askr *et al.*, 2023, p. 5999).

## **2.5 Limits of Current Models**

One of the prevailing limitations highlighted in deep learning research for ADR prediction is the lack of interpretability and explainability in models. While they provide high accuracy, deep learning models introduce a level of uncertainty as to why predictions have been made. This causes a lack of trust, which impedes the adoption of machine learning for ADR prediction. Interpretable models are therefore especially important in medical and pharmaceutical fields, where lives are at risk if wrong predictions are made.

Deep learning models can be considered a “black box”, meaning data is given as input to the model and it outputs a prediction, while the internal processes that govern the model’s decision-making process are obscured (Lee and Chen, 2019, p. 1340). The case has been made that both doctors and patients would be less likely to trust the predictions of an algorithm that doesn’t provide reasoning for its predictions and, arguably more importantly, models become much harder to debug when they fail or make incorrect predictions (Vamathevan *et al.*, 2019, p. 474).

## **2.6 Explainable Methods**

To account for the interpretability and explainability issue with deep learning models, several methods can be used to help explain predictions. It is important to first highlight the distinction between explainability and interpretability, two terms that can sometimes be used interchangeably. Interpretability is how transparent the model is, showing its entire decision-making process of how it arrives at a prediction from an input. Explainability is how well the model explains that decision-making process to a non-technical user. Techniques for both are similar and are often used together, so when being discussed in research the two terms can overlap.

One of the biggest impediments to interpretable AI is that it can lead to a decrease in accuracy due to a need for simplifying the model architecture, while explainable AI methods have shown less of this, often having no effect on accuracy (Vo *et al.*, 2022, p. 2120). It is, therefore, reasonable to focus on explainable methods rather than interpretable ones, when only a clear explanation of a model’s prediction is required rather than the whole decision process.

Explainability can often be introduced by visualising the patterns that a deep learning model has learned. In the context of ADR prediction, this is easiest to do when the model being used is predicting using chemical structure. By highlighting sub-structures within the whole structure, that contribute most to the prediction, the model’s predictions can be explained. One approach towards this showed that neural fingerprints associated with specific ADRs could be identified with a high degree of accuracy (Dey *et al.*, 2018, p. 8).

The XSMILES technique, created for the visualisation of attribution scores with SMILES strings, demonstrates similar concepts (Heberle *et al.*, 2023). The visualization it outputs is shown in *Figure 1*. Given it is a general purpose open source technique, designed to be used by other applications, XSMILES has a wide array of applications for ADR prediction. The identification of important substrings within SMILES strings can help existing deep learning models increase explainability.

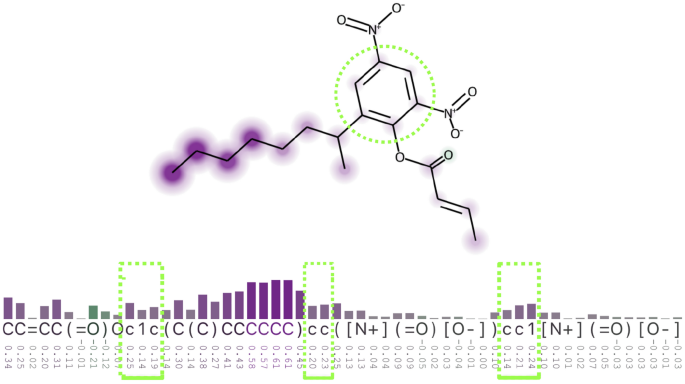


Figure 1: XSMILES Visual Representation

*Heberle et al., (2023) – Fig. 1 Available at: https://jcheminf.biomedcentral.com/articles/10.1186/s13321-022-00673-w/figures/1*

One issue that can arise when discussing explainable methods is a lack of widely accepted quantifiable methods by which a deep learning model’s explainability can be measured. One method, Degree of Explainability (DoX), proposes a measure of explainability for XAI techniques, which assumes a model’s explainability can be measured by the relevant questions its output can provide answers to (Sovrano and Vitali, 2023). However, this only provides a measure of explainability for text-based explainable techniques, such as deep learning models that output descriptive text alongside their predictions. While this is a step in the right direction, it is clear that there is a long way to go for explainable visualisation techniques to be objectively measured.

## **2.7 Identified Gaps**

Given the gaps in the current research that have been highlighted, it is clear that while deep learning models achieve high levels of accuracy when predicting ADRs deep learning cannot be fully adopted in ADR prediction or the drug discovery pipeline as a whole until the issue of explainability is addressed. Additionally, the models must be presented in an easy-to-use interface, to promote trust among the researchers and medical staff who would be utilising them. Few existing models seem to target DDI predictions, due to a lack of varied datasets. There is therefore a clear need for a high accuracy, high explainability deep learning model, that can predict ADRs that result from drug-drug interactions, and provides a clean user interface for researchers to interact with.

Recent work by Zhong *et al* (2023, p. 1) utilises Graph Convolutional Networks (GCNs) for this purpose, improving their explainability while maintaining a high level of accuracy. It defines DDI-GCN, a GCN that predicts DDI’s using chemical structure. Chemical structures are gathered in the form of SMILES strings, and converted to molecular graphs using RDKit. These are then given as input for a GCN, which uses attention mechanisms to learn important graph sub-structures for the prediction, similar to XSMILES. These subgraphs are then presented to the user, as seen in *Figure 2*.

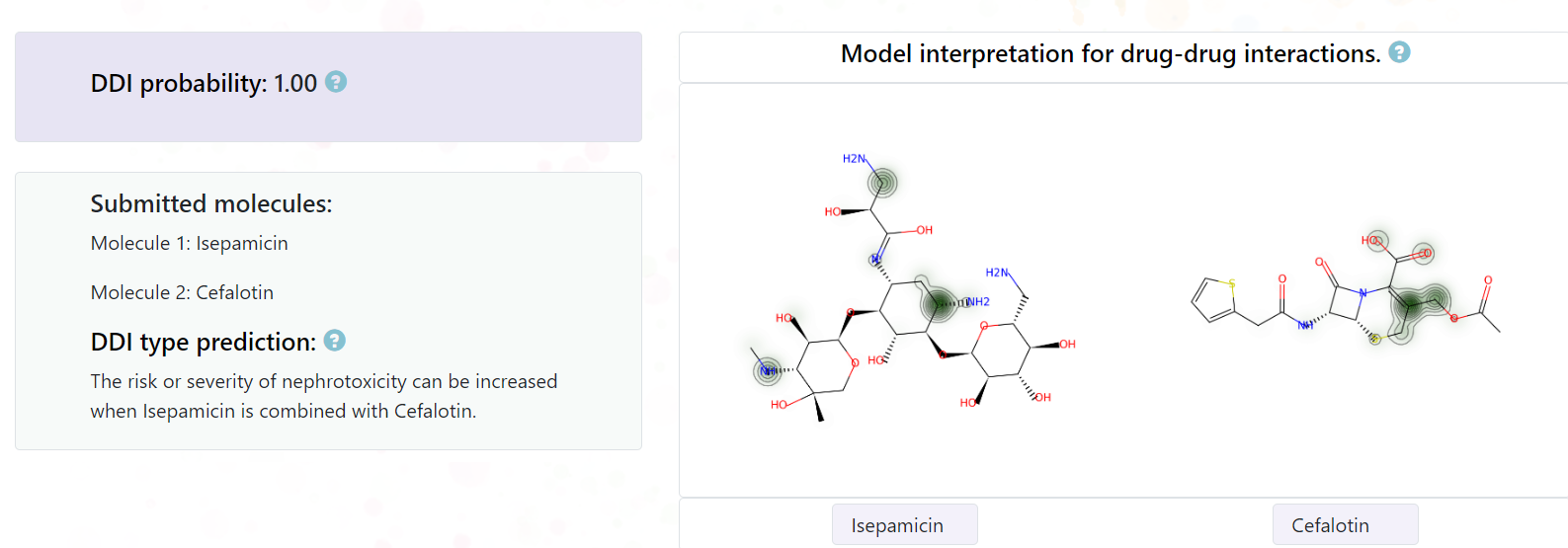


Figure 2: DDI-GCN Model View

Zhong *et al* (2023) *– Model interpretation for drug-drug interactions. Available at: http://wengzq-lab.cn/ddi/new\_result.php?row=0&taskid=EFC1D082-06C9-0289-8221-1D6EB487D111*

In discussion about the model’s future work, the suggestion was made that DDI-GCN could benefit from considering complex forms such as isomers and intramolecular hydrogens. This is because SMILES strings only represent a drug’s chemical structures in a 2D way, meaning that two drugs could have the same atom and bond information, but be arranged completely differently in 3D space. This would alter the way that they interact with other drugs. It is therefore worth investigating the extent to which a drug’s 3D structural properties affect its interactions, and how they can be used to improve model accuracy.

## **2.8 Conclusion**

Throughout the literature review, a timeline of research from the first ADR to the latest model for DDI prediction has been explored. Based off of the identified gaps in the existing literature, a solution can now be proposed to address them. The next section aims to detail the methodology involved with this solution.

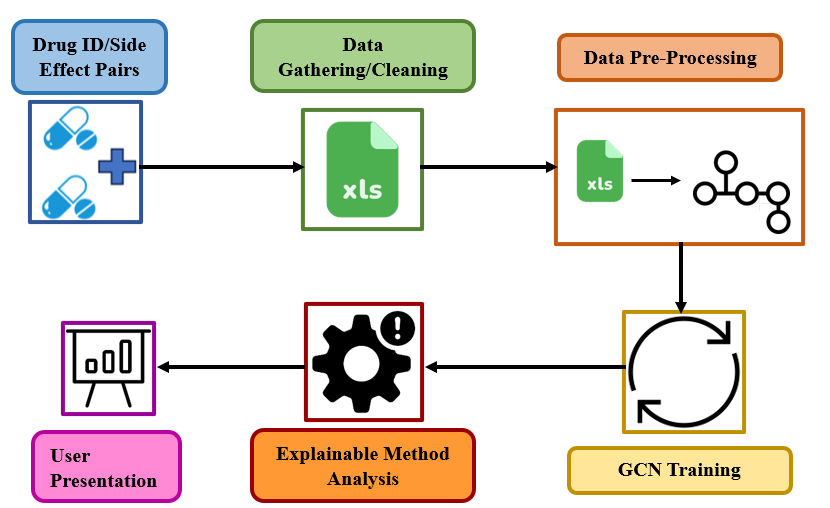
# **Chapter 3 – Research Methodology**

## **3.1 Introduction**

Based on the discussion in the literature review, a new model is proposed, called 3D-DDI-GCN. This is a Graph Convolutional Network, designed to predict side effects from drug-drug interactions, which represents chemical compounds in a 3-dimensional way. The proceeding sections in this chapter discuss the specifics of each step involved with the data gathering, cleaning, and processing, as well as 3D-DDI-GCN’s architecture, and the process of benchmarking.

## **3.2 Overview**

The methodology executed in the implementation section is shown in *Figure 3*, giving an overview of each major step of the process.

Figure 3: Methodology Overview

## **3.3 Dataset Collection**

The process of data collection is summarised in *Figure 4*. It was designed to reduce the unnecessary data so that the model could learn only relevant patterns.

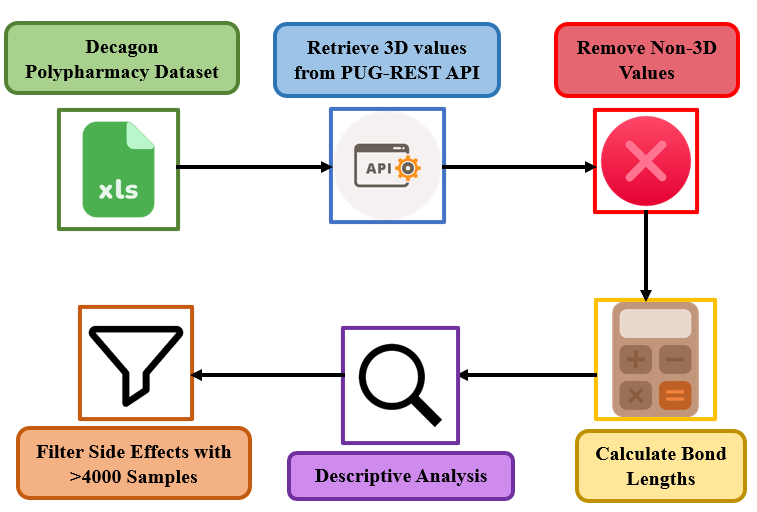
****

Figure 4: Dataset Collection Methodology

The dataset used was the Decagon Polypharmacy dataset, which was discussed previously in the literature review chapter. This consists of drug compound identifier (CID) pairs, with their associated side effect identifier and name labels. The CID’s reference compounds from the PubChem database, which contains the 3D information necessary for the 3D-DDI-GCN model. The side effect identifiers correspond to side effects from the SIDER database, which contains information regarding definitions of side effects, and their synonyms.

Pubchem provides the PUG REST API for data gathering (Kim *et al.*, 2023, p. 1377), allowing requests to be made for compound’s information using their CID. The API provides the option of specifically requesting a compound’s 3D structural information in an SDF file. The data was iterated over, requesting the 3D information for each drug pair and matching them to their associated side effect, saving the result in a new dataset.

The 3D information chosen was the atoms, including their types, coordinates, charges, as well as bonds and their types. This provides a total picture of the atoms of a compound, their spatial arrangement, and the bonds between them. The lengths of all atom’s bonds were then calculated to provide further spatial information for the model. It should be noted that, due to the smaller size of the dataset, complex features were avoided, and the amount of features was kept small to avoid noise in the data.

## **3.4 Dataset Pre-processing**

The process of data pre-processing is summarised in *Figure 5*. This transforms the compound’s 3D data into graphs and then creates representations of those graphs for the model to learn from.

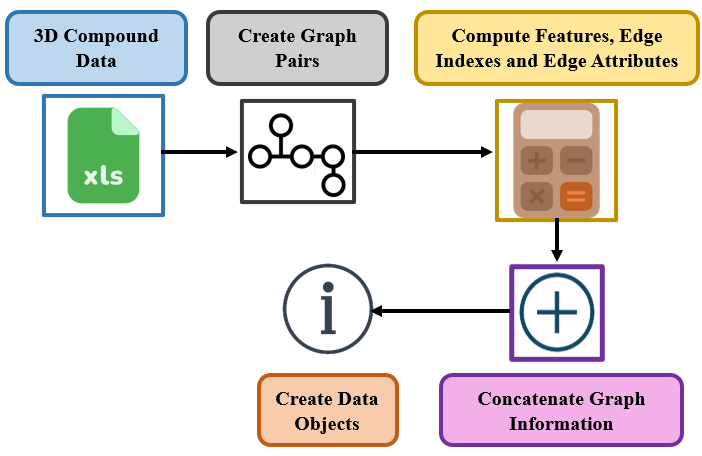


Figure 5: Dataset Pre-processing Methodology

The 3D data translates naturally to graph format, with the atoms of the compounds represented by nodes, and the bonds represented by edges. Nodes contain the type of atom, coordinates, and charge, while edge weights represent distances between nodes. To represent these graphs, their feature matrices, edge indexes, and edge attributes were then calculated. This is necessary so that they can be understood by the 3D-DDI-GCN model. The features, edge indexes, and attributes for both compounds are then concatenated, with padding where necessary to allow for concatenation. The resulting data is put into data objects, which can then be input into the model.

## **3.5 Model/Explainable Techniques**

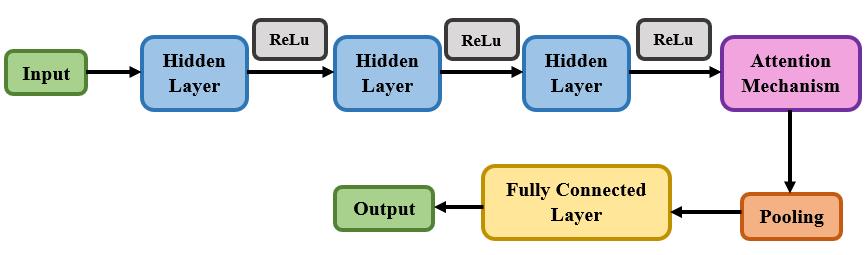
**The 3D-DDI-GCN model architecture is summarised at a high level in *Figure 6*. The hidden layers are Pytorch Geometric graph convolutional layers where graphs are turned into hidden representations which use neighbouring node information to group nodes by patterns that they learn. The Rectified Linear Unit (ReLU) activation functions between hidden layers introduce non-linearity to the model, helping to learn more complex patterns in the graph pairs. The attention mechanism calculates attention scores for the nodes of the graphs, finding important nodes for the patterns it has identified. Global mean pooling is used to combine the node-level embeddings learned from the hidden layers into a single graph-level embedding. The fully connected layer maps the graph-level embedding to the output dimension, which is the number of unique side effect labels. This output represents the model's prediction scores or logits for each class. Given the model is performing single-label classification, the highest score is then used to provide a prediction for the graph.

Figure 6: Model Architecture Diagram

Due to the format of the input data, it was decided that GCNs provide the most potential out of all deep learning models which could be used for explainable graph classification of chemical compound pairs. By using the previously mentioned attention mechanism, predictions can be traced backwards through the model to determine why they have been made, by highlighting the most important nodes in the graph for that prediction.

With the most important nodes in the graph being output from the model, they can then be visualised on the original separated graph pairs. This is presented to the user with a 3D visualisation of both graphs, which appears when they input IDs for the drug pair they would like to predict side effects for. Important nodes and highlighted in yellow, allowing for further investigation of their features. This provides explainability for what patterns the model is finding when it makes predictions. Providing this in a user-friendly GUI was a vital step in ensuring that non-technical medical practitioners/researchers could trust the 3D-DDI-GCN model.

## **3.6 Benchmarking**

The 3D-DDI-GCN model was benchmarked against two other models, a Bi-directional Long Short-Term Memory (LSTM) network model, and a 2D-GCN model. This allows for the 3D-DDI-GCN model to be compared to both a traditional machine learning method, and a model equivalent to DDI-GCN. The training process for these models is summarised in *Figure 7*. The full code for these models can be found in Appendix C.

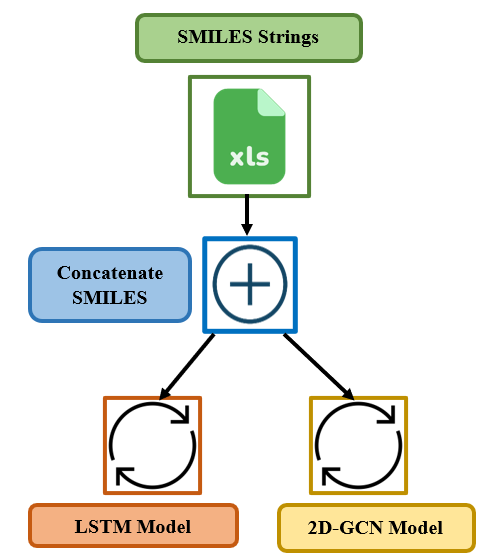


Figure 7: Benchmarking Methodology

For the benchmark models, 2D representations of the compound pairings used in the 3D-DDI-GCN model were given as input to the benchmark models. The 2D compounds were represented as SMILES strings, containing only atom and bond information, with no spatial data.

A BiLSTM model was used chosen due to its ability to capture long-term dependencies far better than traditional RNNs, helping to counter the vanishing gradient problem. SMILES string pairs were concatenated before being input to the model. Being bi-directional, each part of the input sequence uses information from both the past and present, allowing it to better capture relationships between the compound pairs. This is specifically important for concatenated pairs of SMILES strings, where relationships need to be captured between characters across the whole pair. Overfitting in LSTMs is common however when working with small datasets, and given the small amount of samples for each side effect this was taken into consideration when discussing its results.

A 2D-GCN model (standard GCN with no added 3D elements) was used as a copy of DDI-GCN for benchmarking, taking the SMILES string pairs and turning them into graphs. These graph’s nodes represented the compound’s atoms, while the edges represented the bonds. Given that chemical compounds fit a format that is very similar to a graph, and GCN’s are designed to handle graph data, it allows for the data to be kept in a much more recognisable format. This aids any later explainable methods being used. Like RNN’s, GCN’s can handle graphs of varying sizes, which was important for the different sized compounds being input to the model.

## **3.7 Conclusion**

In conclusion, this methodology chapter proposes a new deep learning method, outlining the methods which were used in the implementation chapter, with reasoning provided for each decision made. First, an overview diagram which highlights each step of the process is given, and each proceeding section focuses on a single step of that process. Details for data collection, cleaning and processing show how the dataset was prepared for the proposed model. The 3D-DDI-GCN models architecture is defined, as well as the models which it was benchmarked against. The next chapter shows the implementation of this methodology.

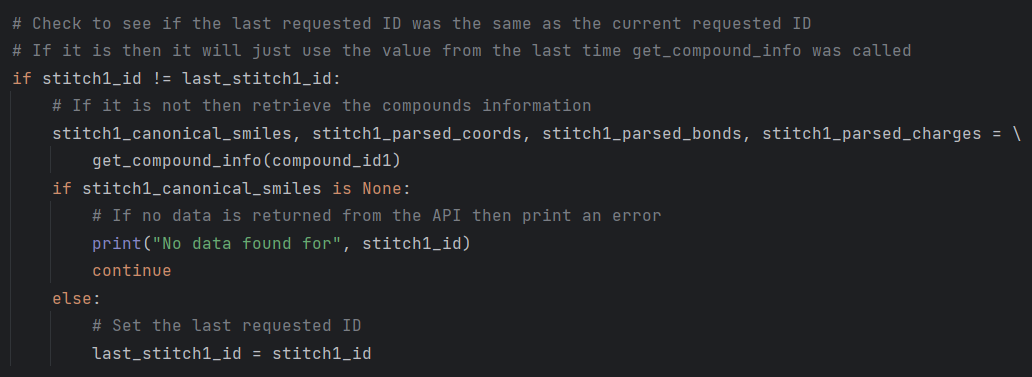
# **Chapter 4 - Implementation**

## **4.1 Introduction**

This chapter details the implementation of the plans laid out in the methodology chapter. Code snippets are provided to give a clear outline of the entire process, including screenshots for the models GUI.

## **4.2 Data Gathering & Analysis**

The full code for the entire data gathering and analysis process can be found in Appendix B, split between *“data\_gathering.py”* and *“descriptive\_analytics.py”*.Using the PUG REST API, data was retrieved about the chemical compounds using their CID’s. This allowed for the compounds SMILES strings, atom types, atom coordinates, atom charges, and bonds/bond types to be retrieved using the chemical ID’s in the Decagon dataset. Special consideration had to be made for the 5 requests per second limit placed on the API, considering that each row in the dataset contained two compounds, and there were over 1,000,000 rows of data. To reduce the requests made to the API, a check was implemented to re-use retrieved data if the same chemical ID appeared in multiple consecutive rows. This can be seen in *Figure 8*.

***Figure 8: API Request Reduction Code*

Retrieval from the API was first tested using a well-known chemical compound, Aspirin, as seen in*Figure 9* and *Figure 10*. The SMILES, atom, bond and charge data are retrieved as separate lists which reference the indexes of individual atoms.

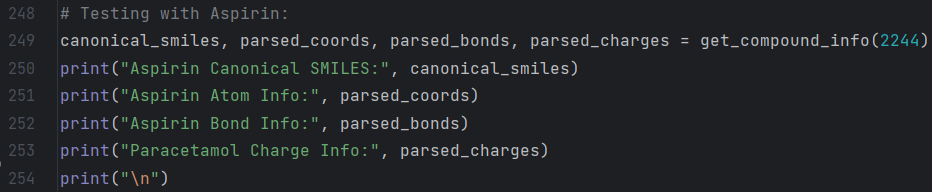
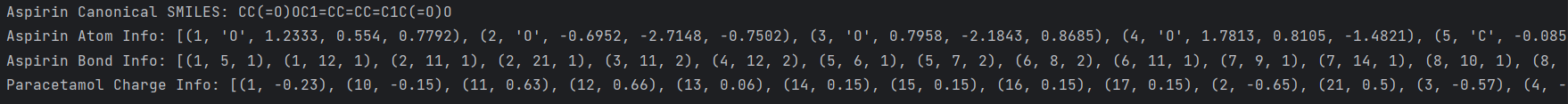
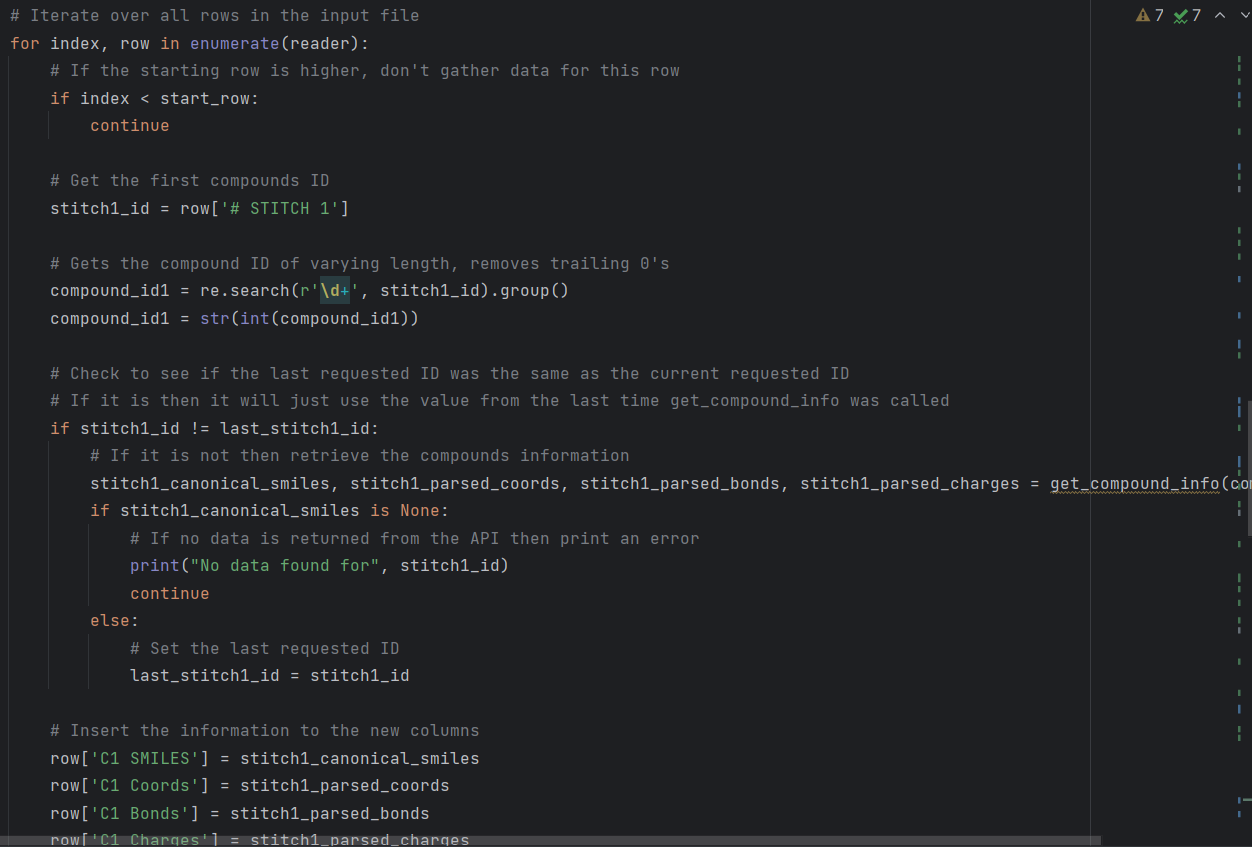


Figure 9: Data Gathering Test Code

Figure 10: Data Gathering Test Result

*Figure 11* shows the core loop of the data gathering process. The process for both compounds in a pair is to retrieve their ID, and give it as input to *“get\_compound\_info”*, which returns the compounds 3D information. This is then written to new columns which hold that information, and saved to *“gatheredData.csv”*.

Figure 11: Data Gathering Code

To provide further spatial information for the model, the length of the bonds between atoms was calculated using their coordinates, with the Euclidian distance formula, defined in the background chapter. The code, seen in *Figure 12*, shows this formula being used. The resulting data is then written to a new file labelled *“computedData.csv”*.

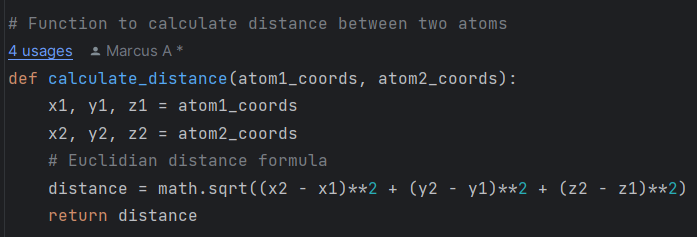
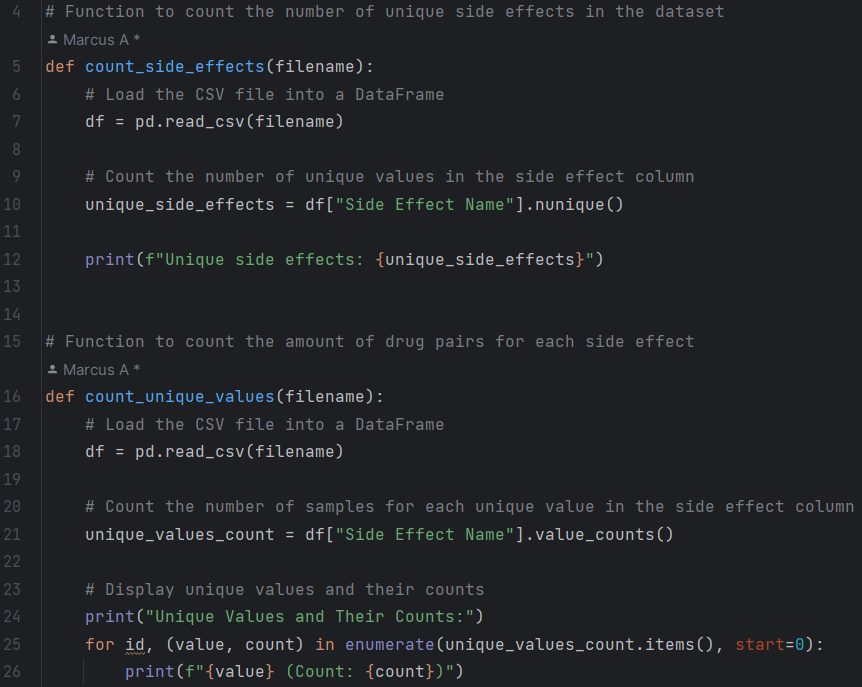


Figure 12: Euclidian Distance Code

The full dataset was then investigated, analysing the number of unique side effects, and the amount of samples for each side effect. The code for this can be seen in *Figure 13*, and the result in *Figure 14*. This was done to determine if there would be enough samples for each side effect to ensure that the 3D-DDI-GCN model learns sufficient patterns in the data for its predictions.



-

Figure 13: Descriptive Analytics Code

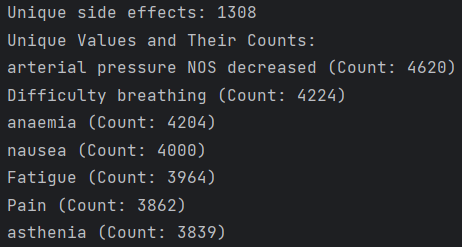


Figure 14: Descriptive Analytics Result

The analytics revealed that there were 1308 unique side effects, and while a select few contained a few thousand samples, the majority of side effects had less than 100. This shows that while the Decagon dataset contains over 1,000,000 rows of data, the amount of samples per side effect are relatively small. It was decided that only the 3 side effects with the most amount of samples would be used in the training data for the model, to reduce noise in the data. The end result was *“filteredData.csv”*, which has 13049 rows of drug pairs, associated with one of three side effects. This limits the scope of the proceeding models to a classification task for predicting one of three side effects.

## **4.3 Data Pre-Processing**

The process of data pre-processing is necessary to turn the 3D graph information into a proper representation which can be used as input to the 3D-DDI-GCN model. Before the compounds could be represented as data objects, their information first had to be put into a graph format. As python does not have inherent support for graph data structures, it was decided that the compounds would be turned into NetworkX graphs, rather than build a list or dictionary representation of a graph. NetworkX is a package which provides pre-made classes for graph data structures (Platt, 2019, p. 12). The benefit of NetworkX graphs for this case was that they provided a simple way of creating graphs, and easily converting them to a format which fits with Pytorch Geometric’s GCN layers.

The full code for the entire data pre-processing process can be found in Appendix C, *“graph\_creation.py”*. *Figure 15* shows the initial process of creating the graphs from their coordinate lists. These are lists of sequential atom indexes, atom types, and x/y/z coordinates. A list of graphs is then returned. In *Figure 16*, each graph in the list is updated with a list of their atoms charges.

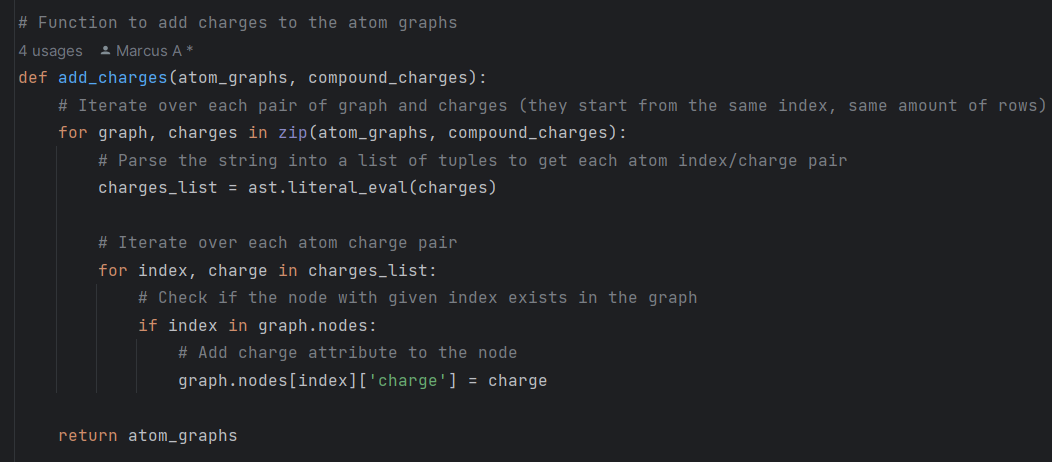
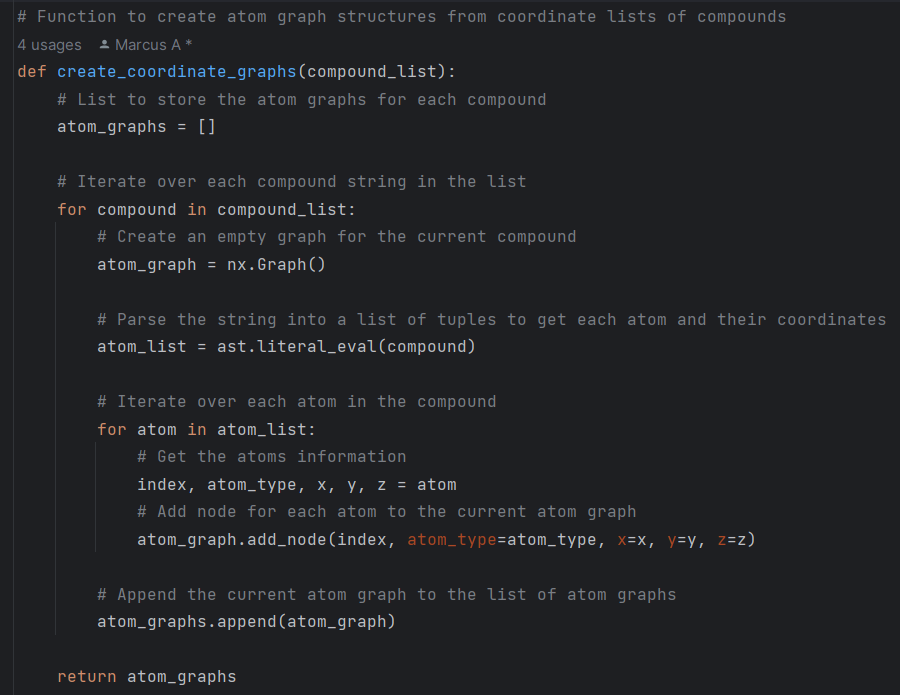
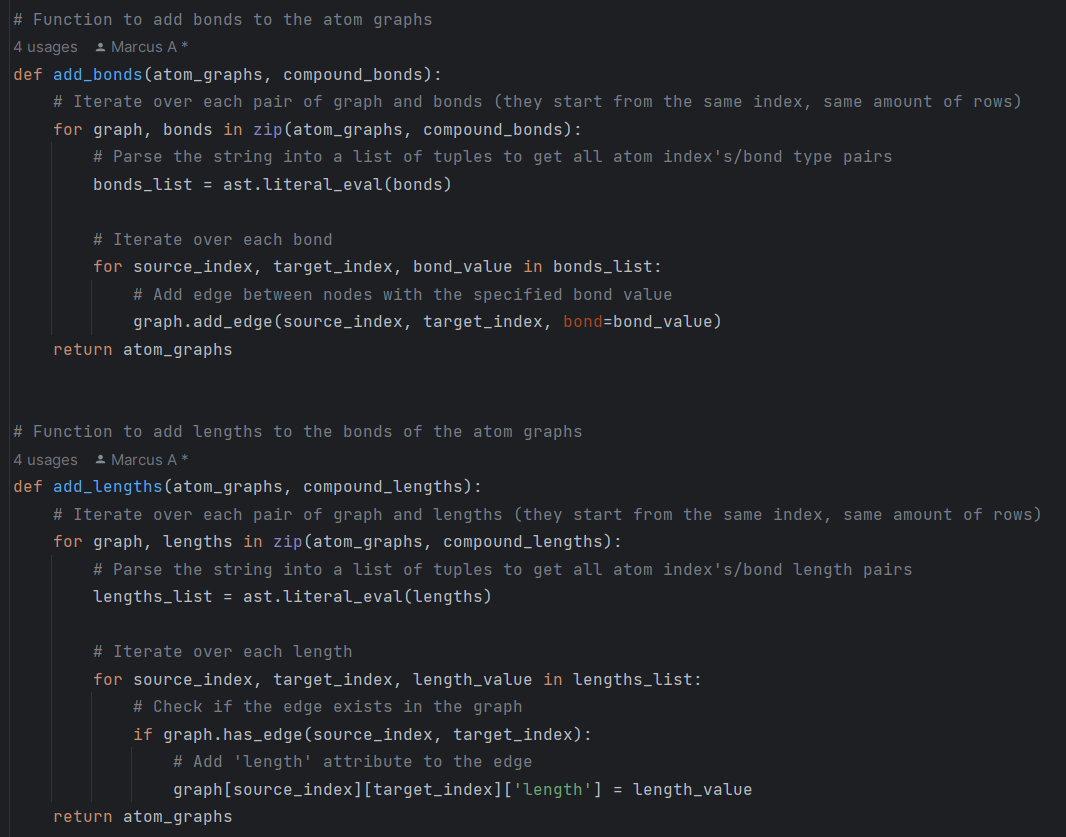
Figure 15: Graph Creation Code

Figure 16: Atom Charge Code

Like with the charges, each graph in the list is updated with a list of their bonds and the length of those bonds, as seen in *Figure 17*. This results in a now complete 3D representation of the chemical compounds, with atoms represented by nodes with coordinate and charge properties, and bonds represented by edges.

Figure 17: Atom Bond Code

The process seen in *Figure 18* and *Figure 19* turns the NetworkX graphs into data representations, with the node features, edge indexes, and edge attributes. This is done so that all three for each pair can be concatenated, and put into Pytorch Geometric data objects, which are compatible with the GCN layers of the 3D-DDI-GCN model.

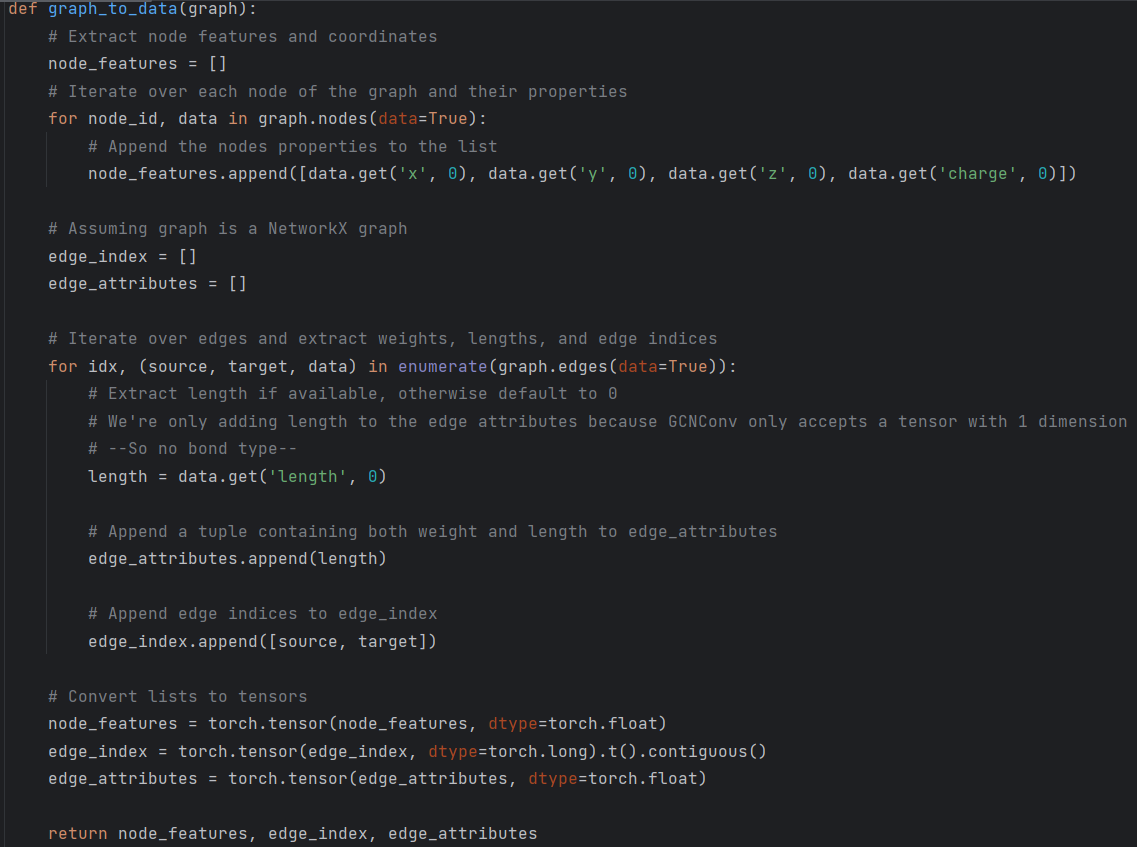
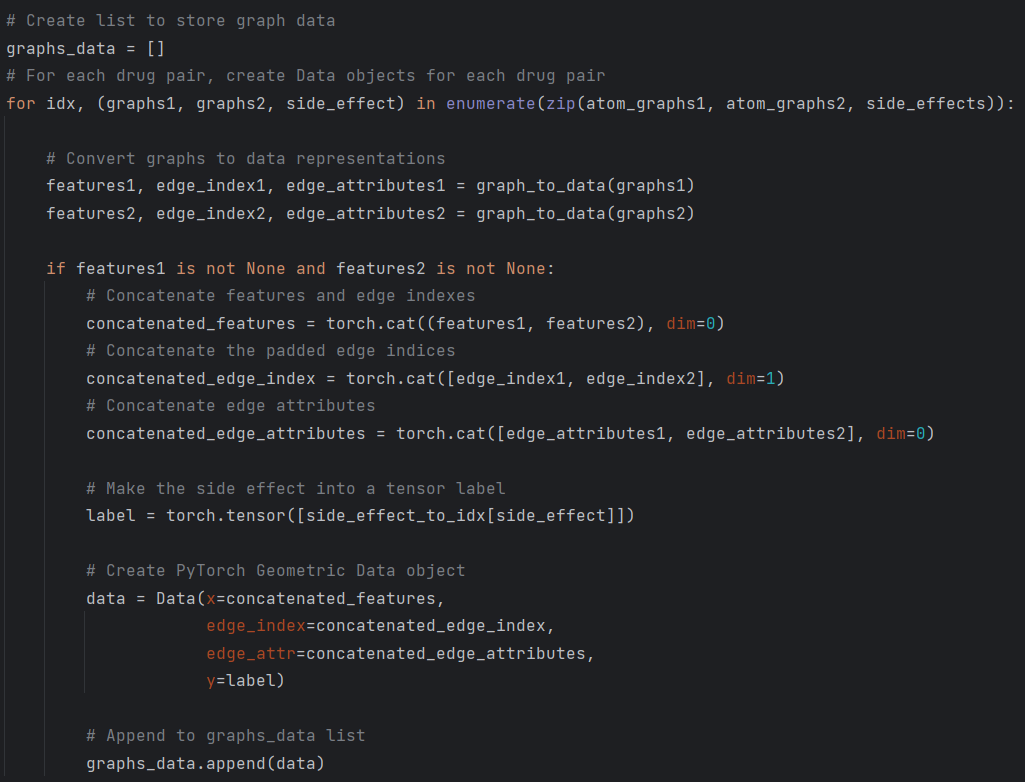
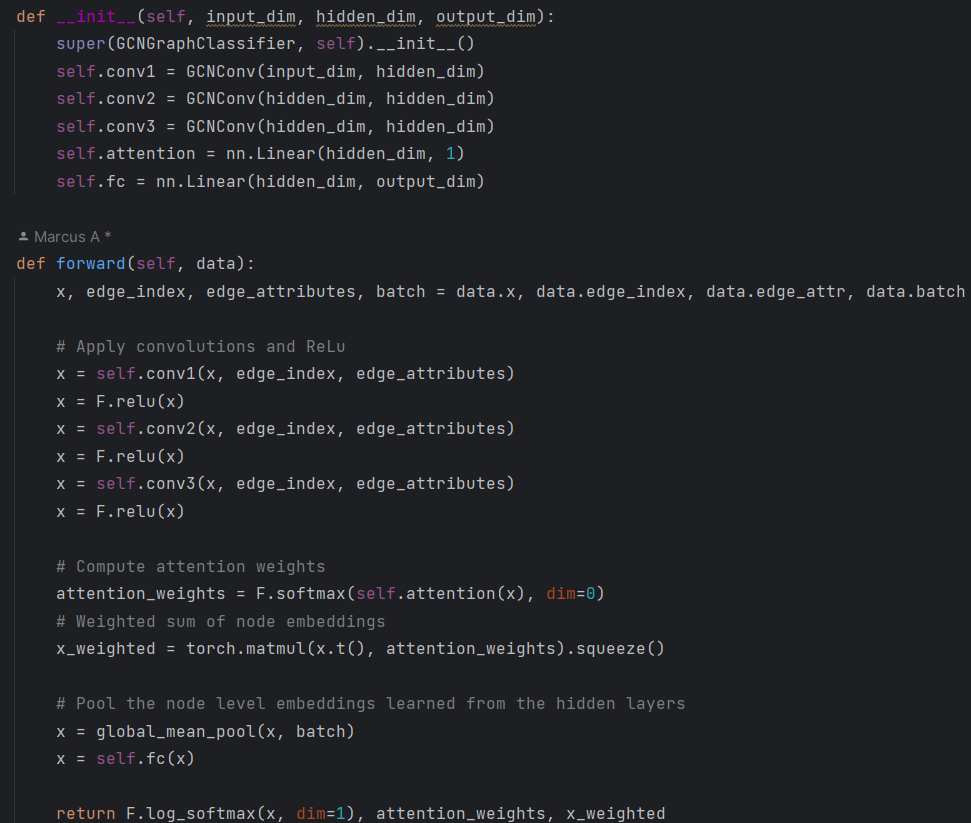


Figure 18: Graph Data Concatenation Code

Figure 19: Graph Data Creation Code

## **4.4 3D-DDI-GCN Model**

The full code for the 3D-DDI-GCN can be found in Appendix C, in *“gnn\_3d.py”*. *Figure 20* shows the implementation of the model architecture laid out in the methodology. GCNConv layers from Pytorch Geometric were used due to their efficient data handling for graph data. The model returns probabilities for each side effect label, and the models attention weights, to later be used for explainability.

Figure 20: 3D-DDI-GCN Model Code

With the data prepared from the pre-processing part of the implementation, the list of data objects is then shuffled and split into training and test data. This is shown in *Figure 21*. Training and testing is executed on separate data to ensure that the model is tested on unseen data. It is shuffled to randomly distribute the different classes across the data, as in its there are large groups of the same compound ID paired with others, which will be more likely to have the same side effects.

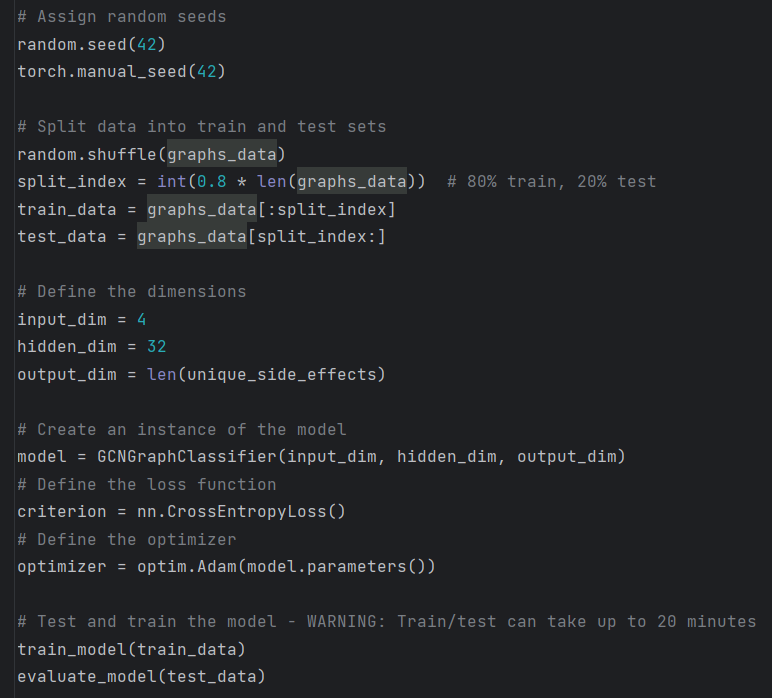
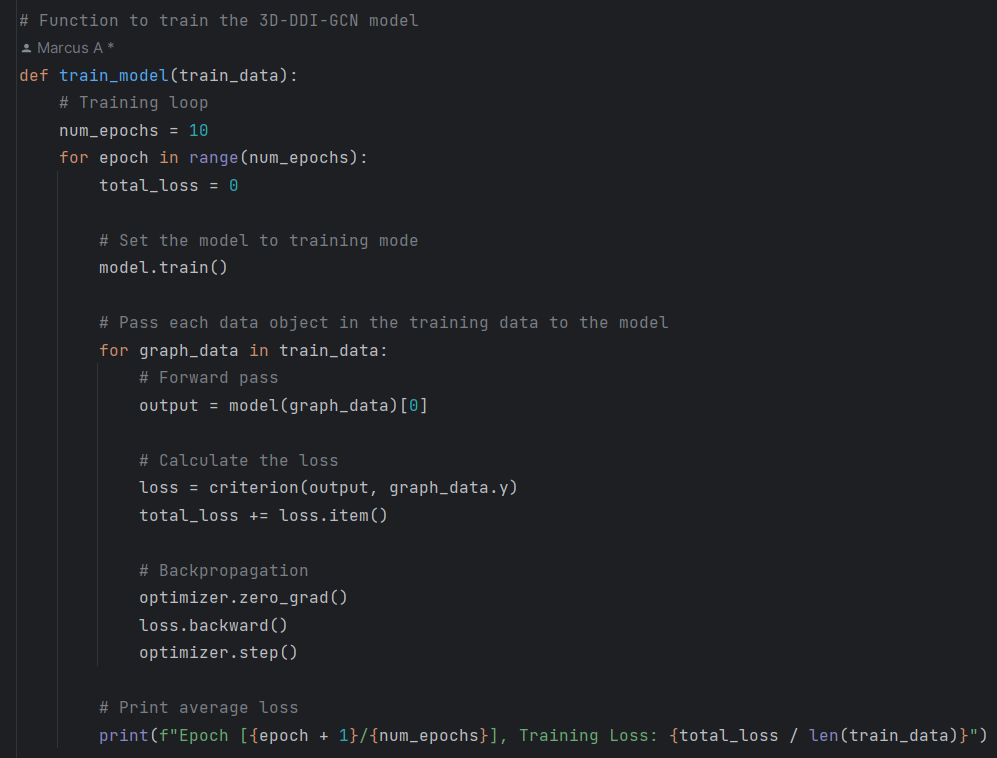


Figure 21: Model Initialisation Code

As seen in *Figure 22* and *Figure 23*, the model goes through a training loop and an evaluation loop. The training loop prints the loss for each iteration, to show how well the model is optimising. The evaluation loop finds the accuracy of the model based off of the amount of correctly predicted labels, and also returns precision, recall and F1-score.

Figure 22: Model Training Code

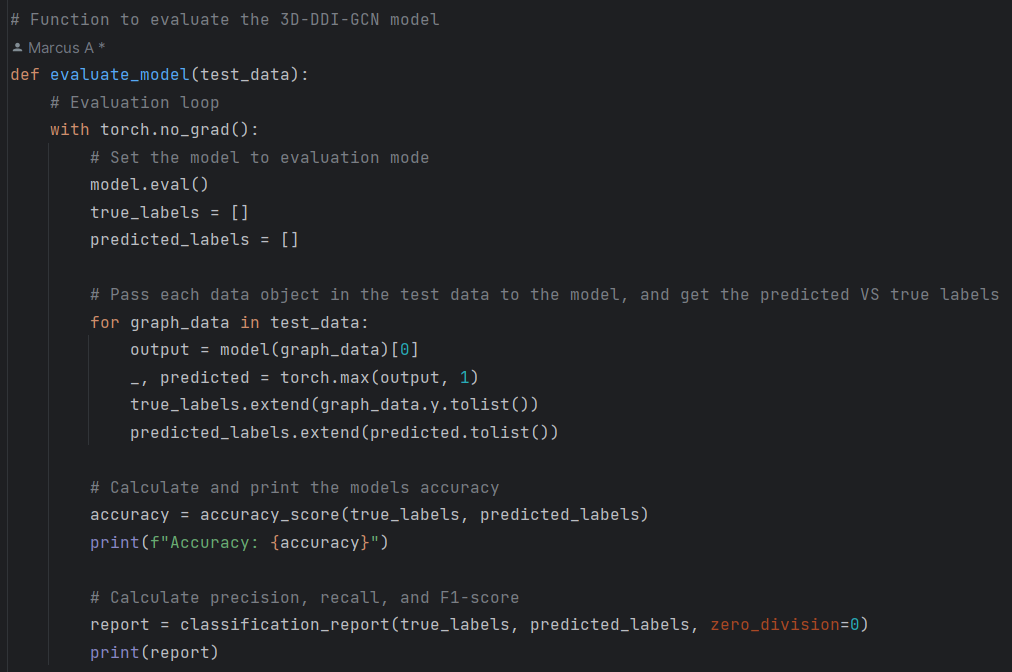


Figure 23: Model Evaluation Code

## **4.5 User Presentation**

The full code for the entire user presentation process can be found in Appendix C, in *“gnn\_3d.py”*, and the version which tested the functions with dummy data is stored in *“visualise\_3d”*. PyQT Designer was utilised to create the GUI that interacts with the model, seen in *Figure 24*. The code which defines the GUI’s class, to be instantiated and called after training the model, is shown in *Figure 25*.

Give a reference for PyQT?

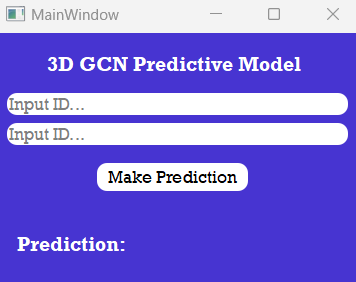


Figure 24: Model GUI

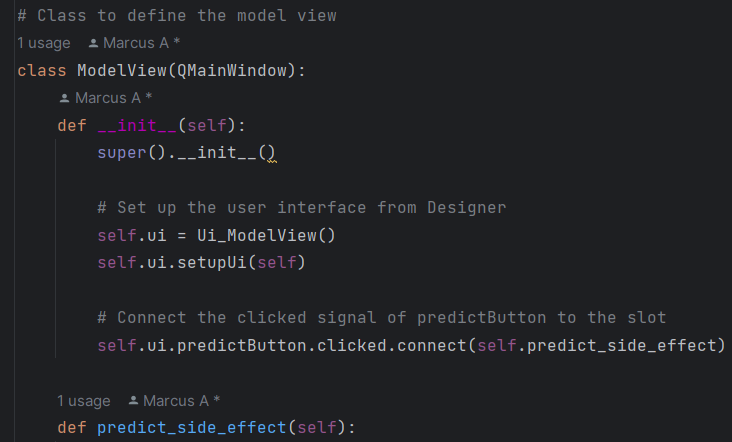


Figure 25: GUI Code

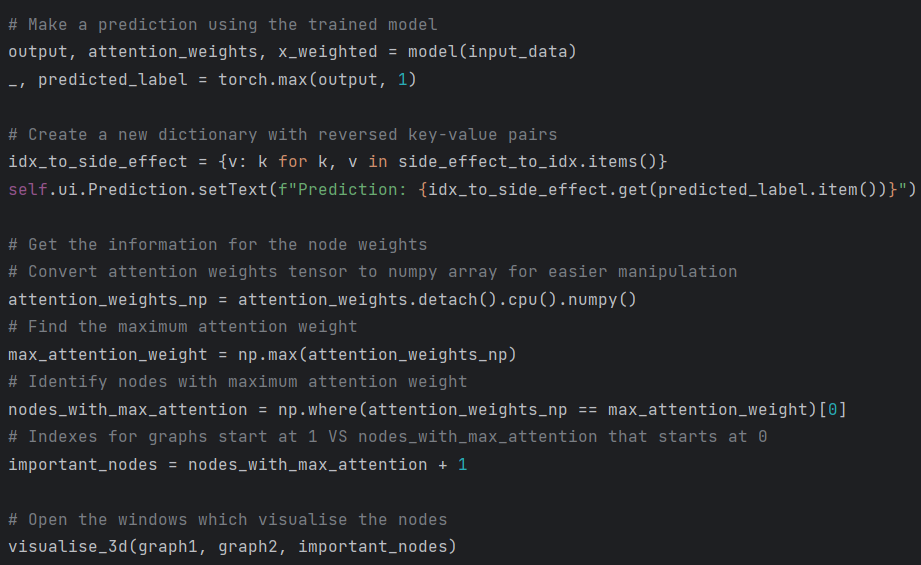
When the user inputs two valid ID’s, and clicks the *“Make Prediction*” button, it triggers the *“predict\_side\_effect”* function. This checks the validity of the ID’s and, assuming they are valid, gathers their 3D information from the PUG REST API and turns them into graphs, just like in the data gathering process. It then calculates the length of the compounds bonds, adding them to the graphs. Like in the data pre-processing implementation, these graphs are turned into representations, concatenated, and then put in a data object. This data object is then input to the 3D-DDI-GCN model, as seen in *Figure 26*.

Figure 26: Prediction Code

Error checking for incorrect or missing ID’s can be seen in *Figure 27*. This is necessary not just in case the user enters the information incorrectly, but also because some drugs may only have 2D representations in the PubChem database.

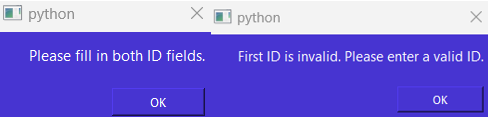


Figure 27: GUI Error Checking

Once the prediction has been made it is displayed in the GUI, as shown in *Figure 28*. As previously discussed, it will predict for one of three side effects. In future iterations with larger datasets, it would learn to predict for the full set of 1308 unique side effects. *Figure 29* shows how graphs are visualised after the prediction has been made, with a fully 3D rotatable plot. The top graph shows how important nodes are highlighted in yellow. The bond and node properties are shown when hovering over them, which can be seen in the bottom graph.

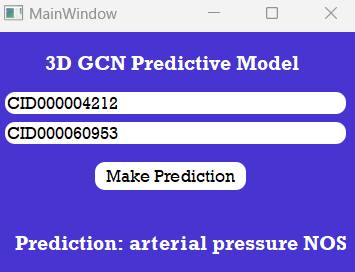


Figure 28: GUI Prediction

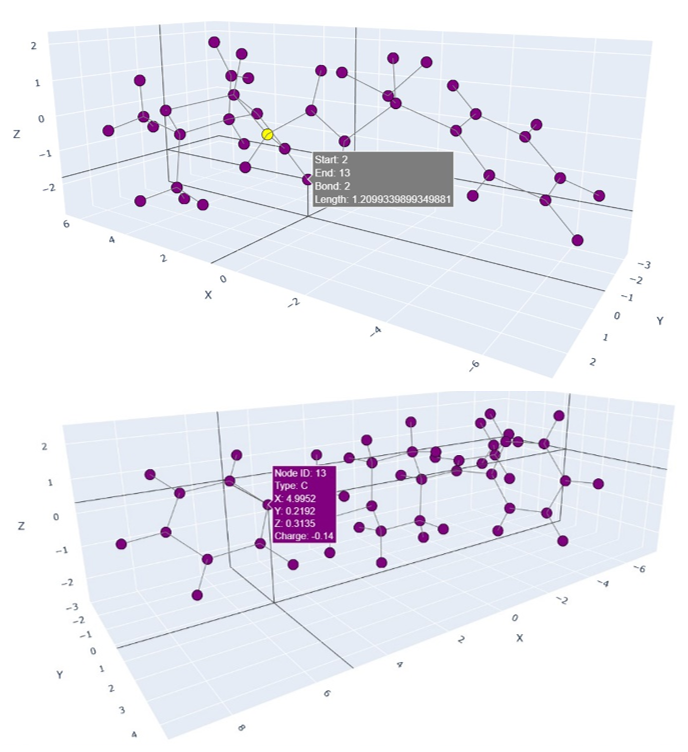


Figure 29: Prediction Visualisation

## **4.6 Conclusion**

To conclude, the successful implementation of the methodology involved several steps, going through data gathering, cleaning, and pre-processing. The 3D-DDI-GCN model architecture was transformed from the methodology diagram to code, with its training and evaluation loops. Code screenshots to support all explanations were provided. The next section shows the implementation of the benchmark models that the 3D-DDI-GCN model was compared to.

# **Chapter 5 – Benchmarking**

## **5.1 Introduction**

Given that the accuracy of deep learning models can be significantly changed depending on the dataset being used, and that different models often have different metrics for evaluation, it was decided that one of the main methods of evaluation for the 3D-DDI-GCN model would be benchmarking. Two models were selected for benchmarking, one traditional method, and one to mimic the DDI-GCN model.

## **5.2 LSTM Model**

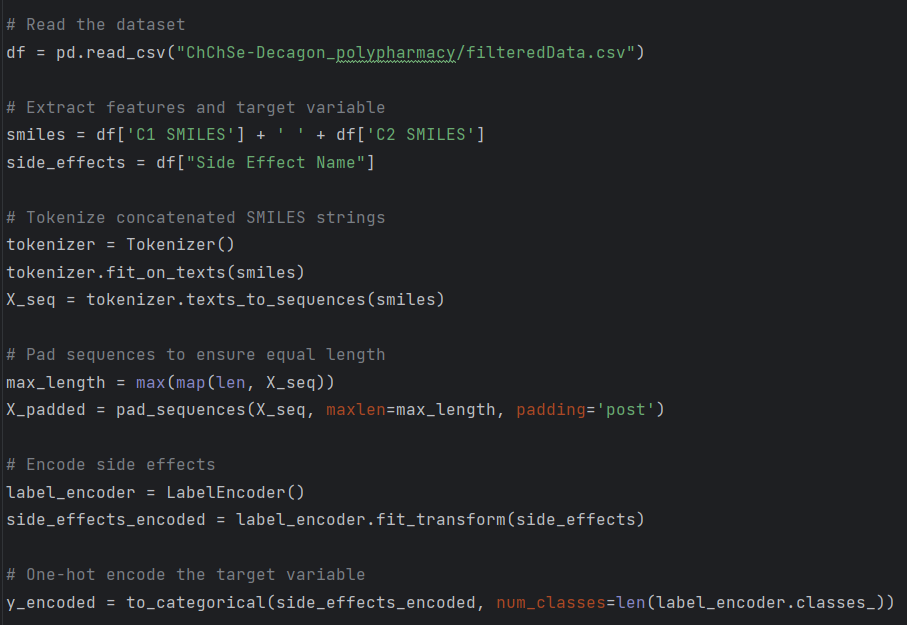
The full code for benchmarking can be found in Appendix C, split across *“lstm.py”* and *“gnn\_smiles.py”*. It was decided that one of the models used for benchmarking would be an LSTM model, which takes concatenated SMILES strings as input. The process of preparing the data for the model is shown in *Figure 30*. The SMILES string representations for each compound are concatenated and tokenized, converting it from a string to sequences of numerical tokens. These sequences are padded to the size of the largest pair, as LSTMs require the inputs to be of equal length. The side effect labels are one hot encoded to represent them as binary vectors which can be understood by the model.

Figure 30: LSTM SMILES Concatenation Code

Like with the 3D-DDI-GCN model, the data is randomly split into training/testing data, as seen in *Figure 31*. The LSTM is built with Bi-directional layers, and dropout layers to prevent overfitting on the data. It is evaluated with accuracy/loss metrics, as well as precision/recall/f1 score/confusion matrix.

*Figure 31: LSTM Model Code*

## **5.3 2D-GCN Model**

The 2D-GCN model architecture shown in *Figure 32*, and like the 3D-DDI-GCN model it consists of 3 convolution layers, with ReLu operations in-between. Given that it is only being used as a benchmark, attention mechanisms were not needed, and explainable methods were not used. The model also only takes the graph features and edge index, as no edge weights were retrievable from the SMILES strings.

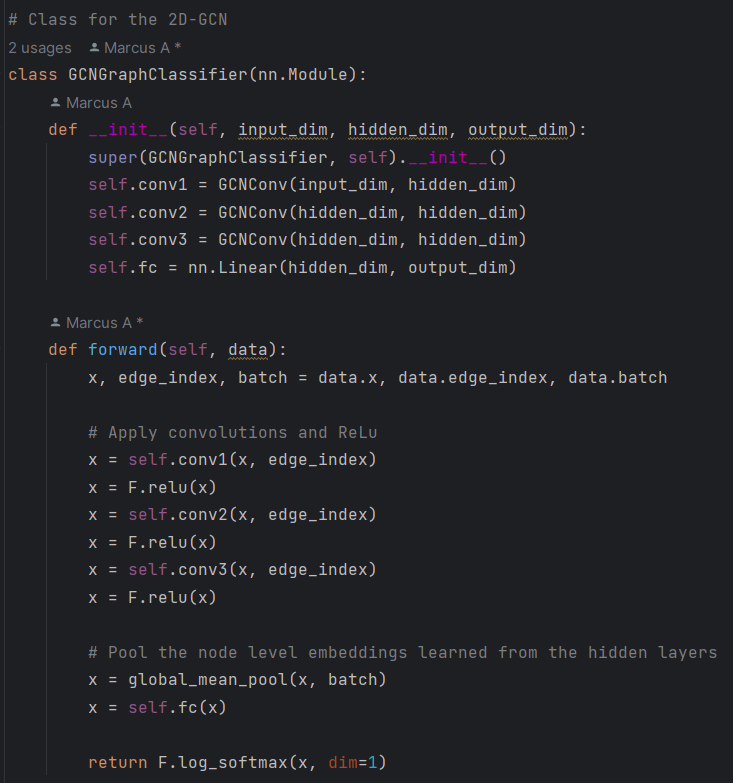


Figure 32: 2D-GCN Model Code

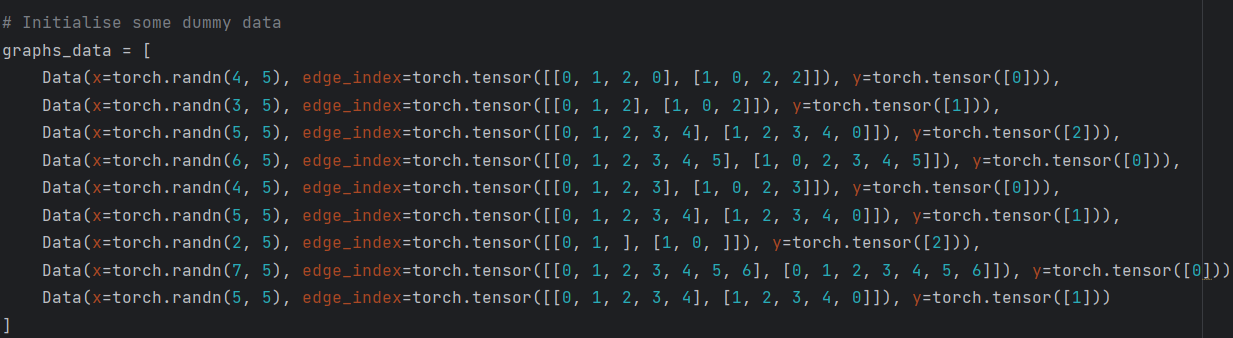
The dummy data shown in *Figure 33* was used for the *“gnn\_smiles\_dummypy”* file, to see how the model would perform on dummy data in comparison to the training data.

Figure 33: 2D-GCN Dummy Data Code

For each drug pair, their SMILES strings are converted graph representations with feature lists and adjacency matrices using the function seen in *Figure 34*. This information is then padded, concatenated, and put into data objects, like with the 3D-DDI-GCN data preparation. This process is shown in *Figure 35*.

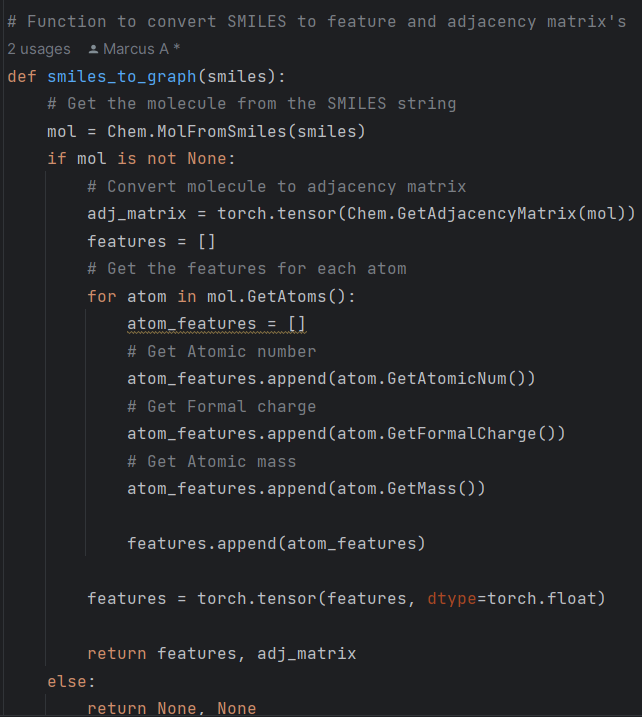


Figure 34: 2D-GCN SMILES To Graph Code

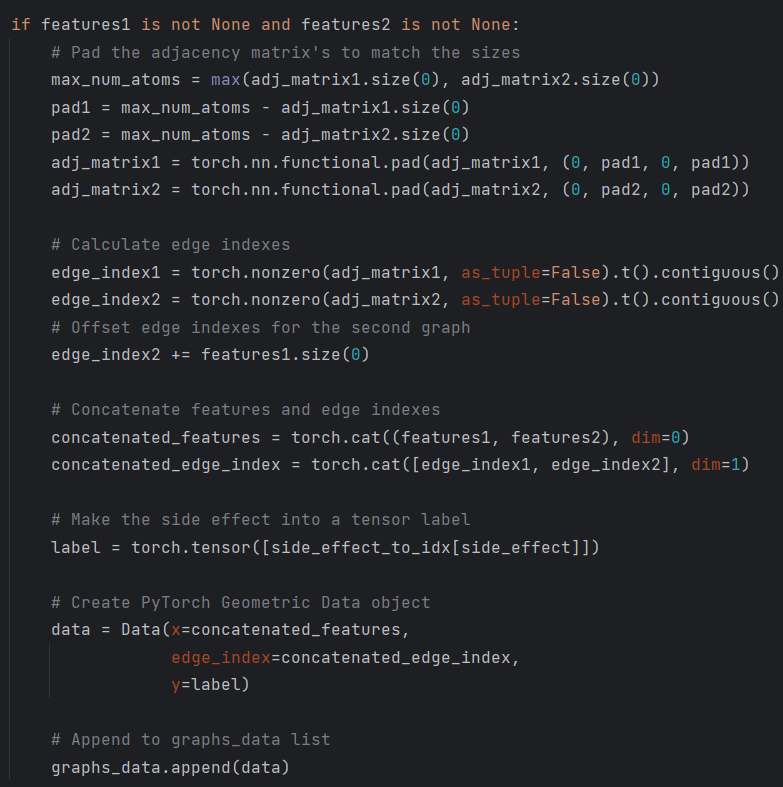


Figure 35: 2D-GCN Data Creation

*Figure 37* and *Figure 37* show the training and evaluation loops for the 2D-GCN model, which uses the same evaluation metrics as the 3D-DDI-GCN model, that were previously discussed.

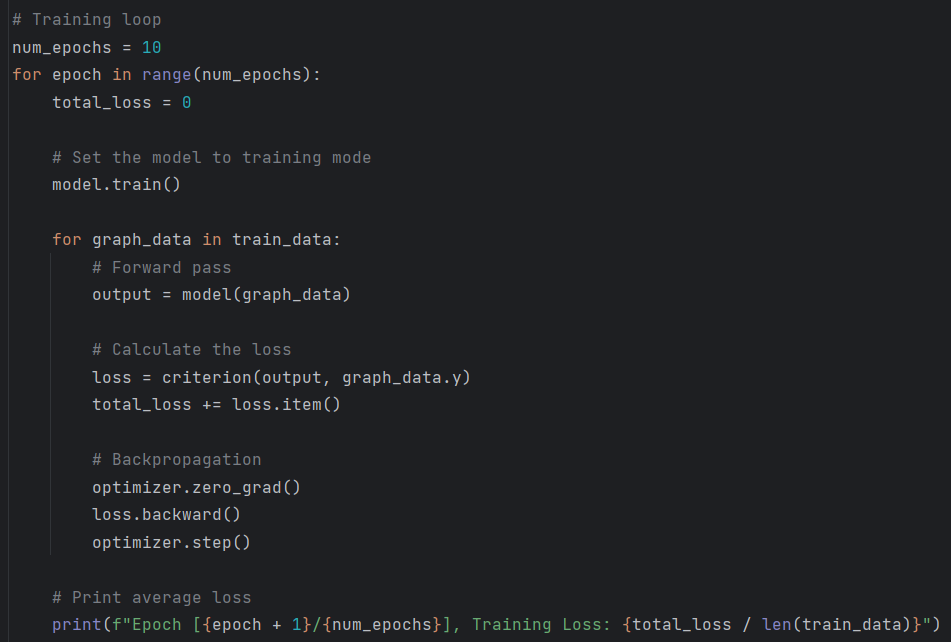


Figure 36: 2D-GCN Training Code

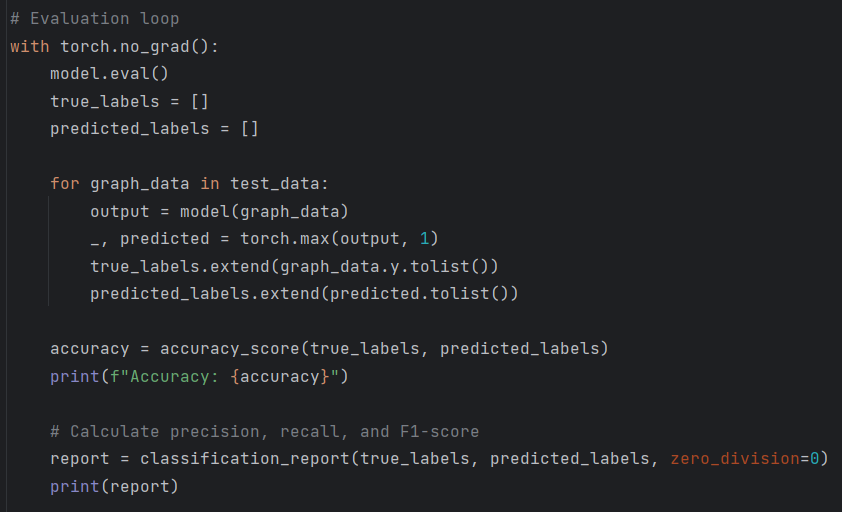


Figure 37: 2D-GCN Evaluation Code

## **5.1 Conclusion**

In summary, the benchmarking implementation was successful, implementing an RNN and a 2D-GCN model. The benchmark models provide a point of the comparisons for the 3D-DDI-GCN model. This lays out the foundation for the following analysis chapter, where all three models are compared.

Top of Form

# **Chapter 6 – Analysis**

## **6.1 Introduction**

Top of Form

To better understand the results gathered from the 3D-DDI-GCN model, and benchmark models, they were analysed and compared in the following analysis chapter. This was done with the aim to unpick the reasons behind the model’s behaviours, and how they could be improved in the future.

## **6.2 LSTM Results**

The LSTM models loss and accuracy can be seen in *Figure 38*. While an accuracy of 0.33 is low, with a high loss of 0.65, initial assessment of these values may be that the model is beginning to fit the data and simply needs fine tuning. However an important piece of information is that the filtered dataset given to the model contains only three side effects, meaning it is only learning to classify between three classes. This means that an accuracy of roughly 1/3 suggests its results are equivalent to if the model simply guessed the same class each time. It is because of this that we can safely conclude that the model is severely underfitting.

****

Figure 38: LSTM Loss/Accuracy

To support this conclusion, the models precision, recall and f1 scores were calculated, as shown in *Figure 39*. This shows that out of all the models predictions, only around 33% of them are correct. A confusion matrix was also generated, with the results seen in *Figure 40*. It’s values further support that the model is underfitting, and suggest that class imbalance is a factor in this.



Figure 39: LSTM Precision/Recall/F1

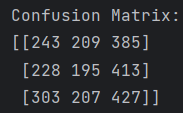


Figure 40: LSTM Confusion Matrix

Given the results discussed so far, it appears that the LSTM model has a significant underfitting problem with the given data. This is mostly due to the relatively small size of the dataset, with 13049 rows of drug pairs, split between 3 classes. As shown in the data gathering section of the methodology chapter, and previously discussed in the data gathering section of the implementation chapter, the decision to filter the data for the 3 classes with the most samples was made to reduce noise in the data from classes with only a handful of samples. Regardless of the underfitting issue, this idea still holds, and introducing more classes with even less samples would not increase the accuracy.

Consideration should also be given to the way that the SMILES strings were analysed by the LSTM model, with each SMILES pair being concatenated and padded to the size of the biggest pair. This was done because LSTM’s require the lengths of all inputs to match the maximum input length. The introduction of padding, while it should have been ignored by the model, could have introduced noise into the data and ultimately contributed to the models poor performance.

Beyond obtaining a larger and more in-depth dataset, a few improvements could be made to the LSTM model to attempt to boost its accuracy and prevent it from underfitting. The way that the SMILES strings are given to the model could be changed, utilising Siamese networks rather than just concatenating the pairs. Siamese networks are an architecture where two identical models handle inputs, and then compare their similarity. However by modifying this a Siamese network could be used to generate embeddings for each SMILES string in a pair, before then concatenating those embeddings, and feeding them into further layers which can learn to predict their label. Hybrid models like the ones discussed in the literature review chapter may also help make up for the reduced accuracy from the lack of data.

## **6.2 2D-GCN Results**

The 2D-GCN’s training loss progression and final accuracy are shown in *Figure 41*. Most notably, the training loss fails to reduce over 10 epochs suggesting that, like with the LSTM model, the 2D-GCN is underfitting. The accuracy of 0.36 also supports this conclusion, with similar reasons being likely, given both models utilise the same dataset with the same 3-class split.

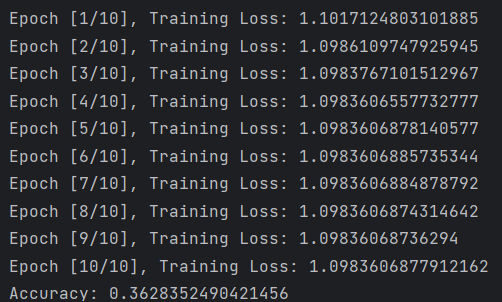


Figure 41: 2D-GCN Accuracy/Loss

To provide a comparison, a test was run with the same pre-processing methods and 2D-GCN architecture, only this time on dummy data. The results are shown in *Figure 42*, and provide an interesting comparison to the results from when the model is run on the actual dataset. This shows how the actual dataset lacks enough samples for a deep learning model to capture each drug pair’s unique relationships with the side effect labels, leading models to guess at random and achieve the same result as if they were using dummy data.

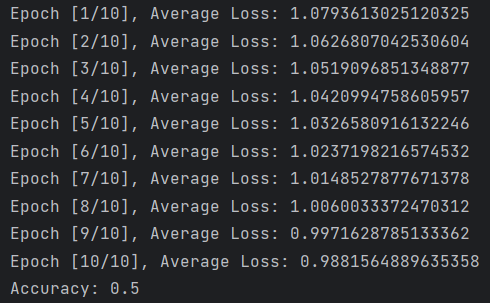


Figure 42: 2D-GCN Dummy Data Comparison

The precision, recall, and f1 values in *Figure 43* provide a new insight into the reasons behind the models underfitting, by showing how the 2D-GCN model seems to only be predicting for the class which has the highest number of samples. While potentially confusing at first, this is somewhat intuitive, given the model initially fails to fit the data due to a lack of samples, opting to pick the class with the most samples and therefore highest likelihood of being correct.

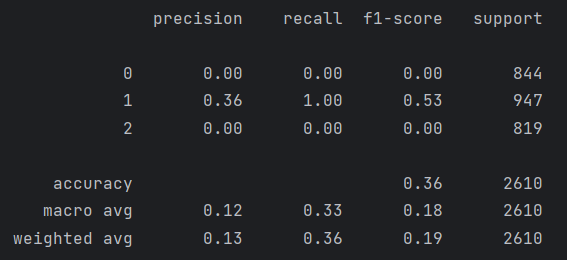


Figure 43: 2D-GCN Precision/Recall/F1

It is worth also noting that the underfitting issues faced by the LSTM and 2D-GCN models highlight exactly why there is a need for more interpretability and explainability in deep learning models. Without being able to trace the decision-making processes of these models, it is extremely difficult to determine with 100% certainty the reasons that they behave in a given way.

## **6.3 3D-DDI-GCN**

The results of the 3D-DDI-GCN model’s training can be seen in *Figure 44*, and are almost identical to the results of the 2D-GCN model (look closely at the beginning loss values). Like with the LSTM and 2D-GCN models, the loss fails to reduce over epochs, and the accuracy is around 1/3. *Figure 45* shows the reason for these near identical results, given the 3D-DDI-GCN model also underfits the data and defaults to always choosing the class with the largest number of samples (which would be the same class that the 2D-GCN defaulted to).

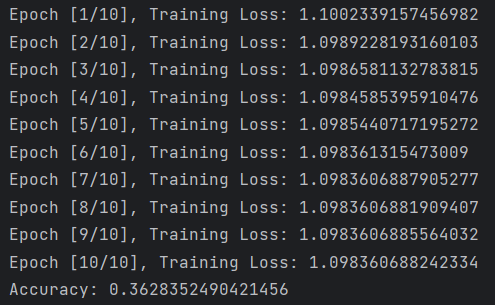


Figure 44: 3D-DDI-GCN Accuracy/Loss

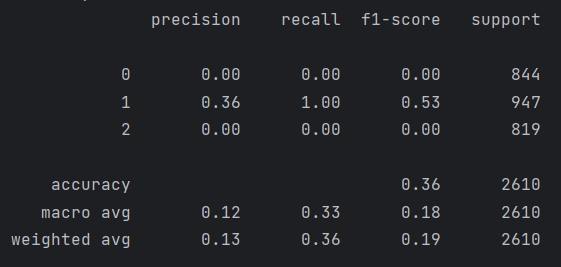


Figure 45: 3D-DDI-GCN Precision/Recall/F1

As briefly touched upon in the previous section, the explainable visualisation generated by the 3D-DDIGCN model actually helps to debug the reasoning behind the behaviour of the model when underfitting. *Figure 46* shows an example of a compound being visualised with its important nodes by a version of the model that has not been through its training loop. Some compounds have no nodes highlighted, while others have random single nodes highlighted. This is expected when not using a training loop, as the model has not had the time to learn the representations. What might be expected when adding in a training loop and underfitting is for the result to be somewhat the same, with either no nodes or random disconnected nodes to be highlighted as important. However instead *Figure 47* shows how all nodes in the graph get highlighted as important for the prediction. This speaks to the behaviour of the model, which lacks enough data to learn the patterns associated with each class, which causes it to generalise those patterns too much and assign all nodes an equal level of importance for its prediction.

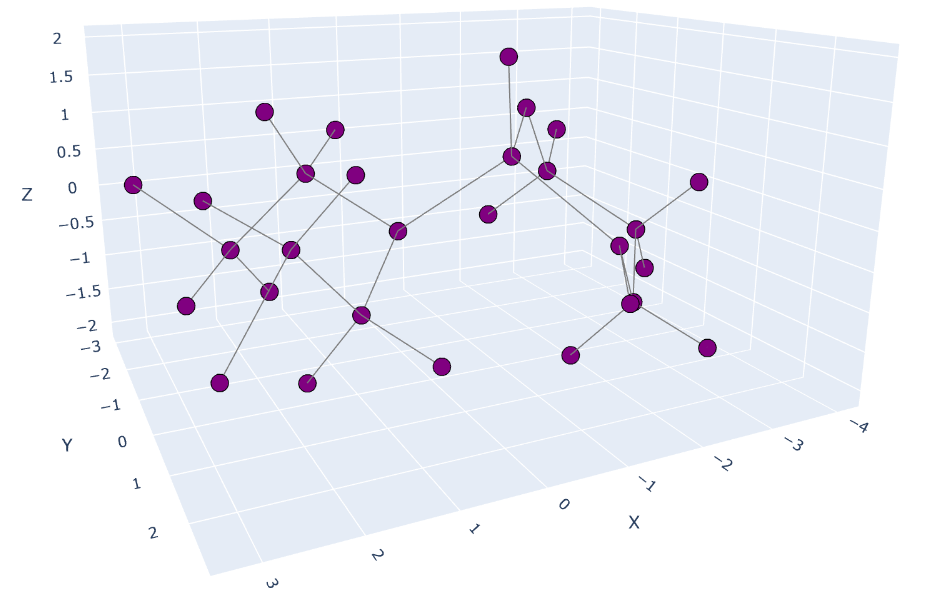


Figure 46: 3D-DDI-GCN Visualisation Before Training

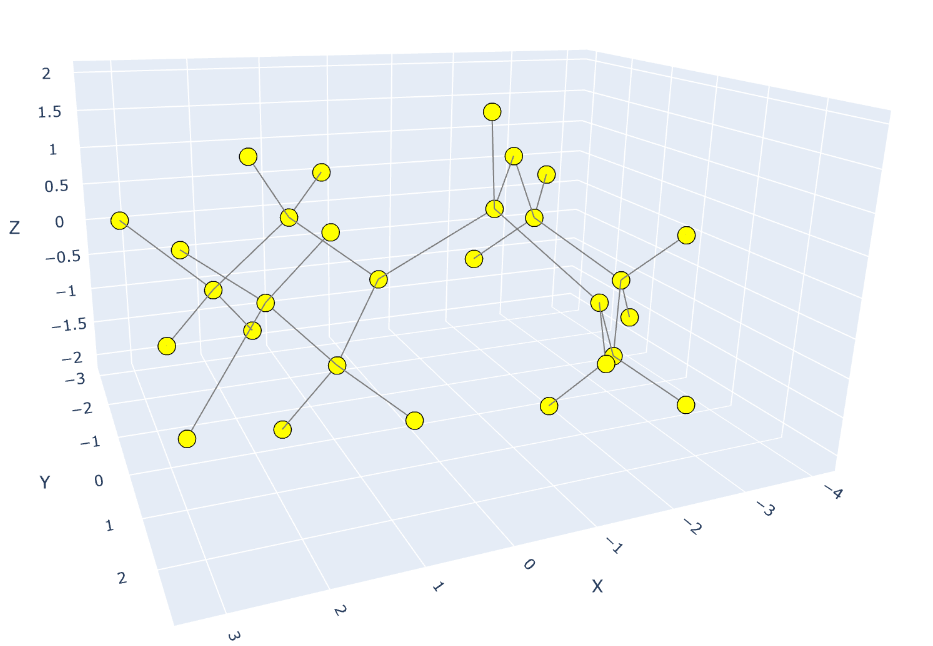


Figure 47: 3D-DDI-GCN Visualisation After Training

If the model were to have succeeded, the expected result is that it would learn the specific subgraph patterns within graph pairs that combine to create each side effect. It would then be able to highlight these subgraphs in the visualisations after making its prediction for a drug pair, to provide better explainability for what combinations of chemical structure can cause those ADRs.

## **6.4 Conclusion**

While the LSTM model provided a considerably low accuracy, and still suffered from underfitting, it is notable that it was still able to fit the data better than the GCN models, demonstrating that a lower complexity of data for a smaller dataset is beneficial. Additionally, hybrid methods may be applied to improve accuracy. The 2D and 3D-DDI-GCN models both failed to fit the data, with a lack of data being the most likely reason. The explainability of the 3D-DDI-GCN model provided valuable insights to support this line of reasoning.

# **Chapter 7 – Discussion and Conclusion**

While the 3D-DDI-GCN model failed to correctly identify sub-structures associated with ADR’s, the research gaps identified in the literature review still present great potential for future work. Based off of the research and results in this dissertation it is believed that, with a larger dataset, representing chemical compounds structures with GCNs is the best path towards simulating them as realistically as possible. This opens up opportunity’s not just for ADR/DDI prediction, but also for other drug research that analyses their structure. Combining GCN’s with proper attention mechanisms allows for greater explainability of their decision making process.

While the Decagon dataset remains a valuable resource for polypharmacy related research, it lacks the necessary data for the specific use-case of 3D compound modelling. Further research would therefore need to be done into the available datasets to determine a more suitable fit.

Several areas of improvement can be worked on to further 3D-DDI-GCN’s model, beyond data quality. Treating the problem as a multi-label classification issue rather than single-label would allow for a more accurate assessment of interactions between drugs, which may cause multiple side effects. Being able to predict side effects for multiple drug interactions would be a large advancement in polypharmacy prediction, allowing medical practitioners to input whole treatment plans, and receive real-time feedback on these plans. Additionally, incorporating patient data into the model would allow for treatment plans to be personalised based on a given patient’s unique biological profile.

# **References**

Alomar, M.J. (2013) ‘Factors affecting the development of adverse drug reactions (Review article)’, *Saudi Pharmaceutical Journal : SPJ*, 22(2), pp. 83–94. Available at: https://doi.org/10.1016/j.jsps.2013.02.003.

Askr, H. *et al.* (2023) ‘Deep learning in drug discovery: an integrative review and future challenges’, *Artificial Intelligence Review*, 56(7), pp. 5975–6037. Available at: https://doi.org/10.1007/s10462-022-10306-1.

Dey, S. *et al.* (2018) ‘Predicting adverse drug reactions through interpretable deep learning framework’, *BMC Bioinformatics*, 19(21), p. 13. Available at: https://doi.org/10.1186/s12859-018-2544-0.

Dubey, S.A. and Pandit, A.A. (2022) ‘A Comprehensive Review and Application of Interpretable Deep Learning Model for ADR Prediction’, *International Journal of Advanced Computer Science and Applications*, 13(9), pp. 204–213. Available at: https://doi.org/10.14569/IJACSA.2022.0130924.

Heberle, H. *et al.* (2023) ‘XSMILES: interactive visualization for molecules, SMILES and XAI attribution scores’, *Journal of Cheminformatics*, 15(1), p. 12. Available at: https://doi.org/10.1186/s13321-022-00673-w.

Hong, E., Jeon, J. and Kim, H.U. (2023) ‘Recent development of machine learning models for the prediction of drug-drug interactions’, *The Korean Journal of Chemical Engineering*, 40(2), pp. 276–285. Available at: https://doi.org/10.1007/s11814-023-1377-3.

Ji, Z.L. *et al.* (2012) ‘Prediction of Putative Adverse Drug Reaction-Related Proteins from Primary Sequence by Support Vector Machines’, *International Journal of Pharmaceutical Medicine*, 19(5), pp. 317–322. Available at: https://doi.org/10.2165/00124363-200519050-00009.

Kim, H.R. *et al.* (2022) ‘Analyzing adverse drug reaction using statistical and machine learning methods’, *Medicine*, 101(25), p. e29387. Available at: https://doi.org/10.1097/MD.0000000000029387.

Kim, S. *et al.* (2023) ‘PubChem 2023 update’, *Nucleic Acids Research*, 51(D1), pp. D1373–D1380. Available at: https://doi.org/10.1093/nar/gkac956.

Kuhn, M. *et al.* (2016) ‘The SIDER database of drugs and side effects’, *Nucleic Acids Research*, 44(Database issue), pp. D1075–D1079. Available at: https://doi.org/10.1093/nar/gkv1075.

Lee, C.Y. and Chen, Y.-P.P. (2021a) ‘Descriptive prediction of drug side-effects using a hybrid deep learning model’, *International Journal of Intelligent Systems*, 36(6), pp. 2491–2510. Available at: https://doi.org/10.1002/int.22389.

Lee, C.Y. and Chen, Y.-P.P. (2021b) ‘Prediction of drug adverse events using deep learning in pharmaceutical discovery’, *Briefings in Bioinformatics*, 22(2), pp. 1884–1901. Available at: https://doi.org/10.1093/bib/bbaa040.

Liu, M. *et al.* (2012) ‘Large-scale prediction of adverse drug reactions using chemical, biological, and phenotypic properties of drugs’, *Journal of the American Medical Informatics Association*, 19(e1), pp. 28–35. Available at: https://doi.org/10.1136/amiajnl-2011-000699.

Nguyen, D.A., Nguyen, C.H. and Mamitsuka, H. (2021) ‘A survey on adverse drug reaction studies: data, tasks and machine learning methods’, *Briefings in Bioinformatics*, 22(1), pp. 164–177. Available at: https://doi.org/10.1093/bib/bbz140.

Platt, E.L. (2019) *Network Science with Python and NetworkX Quick Start Guide: Explore and visualize network data effectively*. Packt Publishing Ltd.

Qian, J. *et al.* (2023) ‘An Attentive LSTM based approach for adverse drug reactions prediction’, *Applied Intelligence*, 53(5), pp. 4875–4889. Available at: https://doi.org/10.1007/s10489-022-03721-y.

Saifuddin, K.M. *et al.* (2023) ‘HyGNN: Drug-Drug Interaction Prediction via Hypergraph Neural Network’. arXiv. Available at: https://doi.org/10.48550/arXiv.2206.12747.

Sovrano, F. and Vitali, F. (2023) ‘An objective metric for Explainable AI: How and why to estimate the degree of explainability’, *Knowledge-Based Systems*, 278, p. 23. Available at: https://doi.org/10.1016/j.knosys.2023.110866.

Suwanda, R., Syahputra, Z. and Zamzami, E.M. (2020) ‘Analysis of Euclidean Distance and Manhattan Distance in the K-Means Algorithm for Variations Number of Centroid K’, *Journal of Physics: Conference Series*, 1566(1), p. 012058. Available at: https://doi.org/10.1088/1742-6596/1566/1/012058.

Valeanu, A. *et al.* (2020) ‘The development of a scoring and ranking strategy for a patient-tailored adverse drug reaction prediction in polypharmacy’, *Scientific Reports*, 10, p. 9552. Available at: https://doi.org/10.1038/s41598-020-66611-8.

Vamathevan, J. *et al.* (2019) ‘Applications of machine learning in drug discovery and development’, *Nature Reviews Drug Discovery*, 18(6), pp. 463–477. Available at: https://doi.org/10.1038/s41573-019-0024-5.

Vargesson, N. and Stephens, T. (2021) ‘Thalidomide: history, withdrawal, renaissance, and safety concerns’, *Expert Opinion on Drug Safety*, 20(12), pp. 1455–1457. Available at: https://doi.org/10.1080/14740338.2021.1991307.

Vo, T.H. *et al.* (2022) ‘On the road to explainable AI in drug-drug interactions prediction: A systematic review’, *Computational and Structural Biotechnology Journal*, 20, pp. 2112–2123. Available at: https://doi.org/10.1016/j.csbj.2022.04.021.

Yang, S. and Kar, S. (2023) ‘Application of artificial intelligence and machine learning in early detection of adverse drug reactions (ADRs) and drug-induced toxicity’, *Artificial Intelligence Chemistry*, 1(2), p. 18. Available at: https://doi.org/10.1016/j.aichem.2023.100011.

Zhong, Y. *et al.* (2023) ‘DDI-GCN: Drug-drug interaction prediction via explainable graph convolutional networks’, *Artificial Intelligence in Medicine*, 144, p. 9. Available at: https://doi.org/10.1016/j.artmed.2023.102640.

Zitnik, M., Agrawal, M. and Leskovec, J. (2018) ‘Modeling polypharmacy side effects with graph convolutional networks’, *Bioinformatics*, 34(13), pp. i457–i466. Available at: https://doi.org/10.1093/bioinformatics/bty294.

# **Appendix**

## **A – Literature Review Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Date | What | How | Why |
| [Large-scale prediction of adverse drug reactions using chemical, biological, and phenotypic properties of drugs](https://academic.oup.com/jamia/article/19/e1/e28/2909247) | June 2012 | Proposes a data fusion approach to ADR prediction, common machine learning methods, and introduces some evaluation methods. Also discusses feature importance of chemical properties for prediction. | The approach they propose uses phenotypic characteristics, known ADRs, chemical structures and biological properties (protein targets and pathway information). They compared 5 machine learning algorithms with fivefold cross-validation, when predicting 1385 known ADRs of 832 approved drugs. They found that phenotypic data was the most useful for predictions. | Aim to increase accuracy of prediction models for ADR’s. |
| [Prediction of Putative Adverse Drug Reaction-Related Proteins from Primary Sequence by Support Vector Machines](https://link.springer.com/article/10.2165/00124363-200519050-00009) | August 2012 | The first case found where they apply machine learning for ADR research. They classify protein structures related to ADR’s. | Using SVM classification. 93.9% of the ADR-related proteins and 98.2% of non-ADR-related proteins were correctly classified. | Identifying ADR-related proteins facilitates the design of drugs with fewer adverse effects by avoiding unwanted interaction with these proteins. |
| [Factors affecting the development of adverse drug reactions (Review article)](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3950535/#:~:text=Pharmacological%2C%20immunological%2C%20and%20genetic%20factors,pharmacodynamic%20abnormalities%2C%20and%20drug%20interactions) | February 2013 | Gives an overview of what factors can cause an ADR, and how important each one is. Also discusses polypharmacy as a factor. | Reviewed literature between 1991 and 2012 from PubMed, the Cochrane database of systematic reviews, EMBASE and IDIS to determine the main factors influencing ADR’s. Categorised factors into patient/social/drug/disease related. | To collate information around ADR’s, to aid future research. The chosen literature sources were the commonly used sources for ADR’s at the time. |
| [The SIDER database of drugs and side effects](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4702794/) | January 2016 | Paper detailing when the SIDER database was created, and an overview of everything it contains. | Released an update to SIDER, which had over 40% more drugs, ADRs and drug–ADR pairs compared to the previous version and more than 2-fold as many drug–ADR pairs. | To continue providing up to date information on ADRs, SIDER is a heavily used ADR dataset. |
| [Modelling polypharmacy side effects with graph convolutional networks](https://academic.oup.com/bioinformatics/article/34/13/i457/5045770?login=false) | July 2018 | Introduces the Decagon model, which is the GNN model that was used to gather the BioSNAP dataset that was used in the methodology. They applied it to polypharmacy ADR prediction. | Pulled data from SIDER, OFFSIDES and TWOSIDES. Created a non-linear, multi-layer convolutional graph neural network model. Decagon accurately predicts polypharmacy side effects, outperforming baselines by up to 69%. | Decagon is developed specifically to handle multimodal graphs with a large number of edge types.  Polypharmacy gives a significant increased risk of ADRs. Knowledge of drug interactions is often limited because the relationships are rare, and often go unnoticed in small clinical testing. |
| [Predicting adverse drug reactions through interpretable deep learning framework](https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-018-2544-0) | December 2018 | Tested multiple machine learning models, as well as an interpretable deep learning approach. | Used chemical structures to predict. Compared 10 different state-of-the-art fingerprint models and found that neural fingerprints from the convolutional deep learning model outperformed all other methods. | The main challenge in representing the molecular graphs of drugs into features is how to represent the varying sizes of each drug molecule into a fixed-size feature representation. The proposes model can simultaneously construct chemical fingerprint features and assess their associations with ADRs. |
| [Applications of machine learning in drug discovery and development](https://www.nature.com/articles/s41573-019-0024-5) | June 2019 | Covers how machine learning is used during the drug discovery pipeline. Identifies the issue of a lack of interpretability and repeatability in deep learning prediction methods for ADR’s. | Provides a breakdown of each part of the drug discovery pipeline, and how ML algorithms are used. Gives detailed diagrams that show how each type of algorithm can be applied. | Aim to improve drug discovery pipelines, to save costs and reduce risk. |
| [Machine learning on adverse drug reactions for pharmacovigilance](https://www.sciencedirect.com/science/article/pii/S1359644618303672) | July 2019 | Discusses the issue of interpretability, and proposes a two stage deep learning framework to try and tackle that issue. | Reviews traditional machine learning methods for pharmacovigilance, and proposes a new two-stage LSTM deep learning framework. The first stage is used to predict the association between drugs and their adverse reactions by using deep learning methods running on big data. The second stage integrates an individual’s biological data into the system. To increase interpretability SMILES was used. | Aim to increase accuracy of prediction models for ADR’s. |
| [The development of a scoring and ranking strategy for a patient-tailored adverse drug reaction prediction in polypharmacy](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7293306/) | June 2020 | Introduces a patient-tailored polypharmacy ADR prediction model. | Trained on 734 drugs from SIDER, designed in Python. Produces an overall severity profile (hospitalization and mortality risk), risk on specific ADR groups and a sorted list of the most important ADRs depending on frequency and severity. Uses a Neural Network (Multi-Layer Perceptron, MLP) machine learning classifier. | Far fewer applications deal with personalized adverse drug reactions (ADRs) prediction in the case of polypharmacy, which is important because patient specific factors do influence ADRs. |
| [A survey on adverse drug reaction studies: data, tasks and machine learning methods](https://academic.oup.com/bib/article/22/1/164/5678053) | January 2021 | Provides a summary of popular ADR datasets, factors involved in ADR’s, common machine learning methods for prediction, and suggests further improvements to the field. Improvements such as polypharmacy prediction (DDI). | Summarized ADR data sources and review ADR studies in three tasks: drug-ADR benchmark data creation, drug–ADR prediction and ADR mechanism analysis. | Attempting to identify current gaps in the field of ADR prediction, and future work. |
| [Prediction of drug adverse events using deep learning in pharmaceutical discovery](https://academic.oup.com/bib/article/22/2/1884/5826453) | March 2021 | Discusses why we need ADR prediction with machine learning. Gives an overview of current methods. Discusses DDI and methods used. Discusses future improvements, such as increasing black box model interpretability with explainable AI. | Doesn’t specify how it selected its literature for review. | Attempting to identify current gaps in the field of ADR prediction, and future work. |
| [Descriptive prediction of drug side-effects using a hybrid deep learning model](https://onlinelibrary.wiley.com/doi/full/10.1002/int.22389?casa_token=wFyukhmhKR8AAAAA%3AaUR1UKic6M0-KVysyeR4GLpuOiXFWigvZcZqZj7NjjfEAqrPDx_a2ACmI1_2PfPC-Ux2tuEtRdScx6rU) | March 2021 | Introduces a new hybrid model for ADR prediction, shows a higher accuracy than baseline methods, despite a small dataset. Future suggestions include work with larger datasets, and incorporating drug information other than chemical structure. | They use a hybrid GCNN-BiLSTM model. Its predictions provide word descriptions of the ADRs, which is not seen in many other models. | CNNs are good at reducing frequency variations; LSTMs are good at temporal modelling. It has been proven that the performance of LSTM could be improved by augmenting it with CNNs. |
| [On the road to explainable AI in drug-drug interactions prediction: A systematic review](https://pubmed.ncbi.nlm.nih.gov/35832629/#:~:text=In%20this%20review%2C%20a%20comprehensive,prediction%2C%20the%20modeling%20methods%2C%20is) | April 2022 | Overview of DDI prediction, data sources, and methods. Also discusses current promising methods of XAI. | Searched five databases up to December 2021: Cochrane Library, PubMed, EMBASE, IEEE, and Scopus. The eligibility criteria consisted of DDI predictive models that were built up using ML - and/or DL-based algorithms. The articles were screened and selected independently by two reviewers, and disagreements were resolved by the third reviewer. Ended up reviewing 94 different research studies. | Attempting to identify data sources and current methods, as well as current gaps in the field of ADR prediction, and future work. |
| [Analysing adverse drug reaction using statistical and machine learning methods](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9276413/) | June 2022 | Overview of various drug databases, machine learning models, visualisation tools and evaluation tools. | Literature review was conducted based on articles published between 2015 and 2020. The keywords used were statistical, machine learning, and deep learning methods for detecting ADR signals. Reviewed 72 articles, of which 51 and 21 addressed statistical and machine learning methods, respectively. | Attempting to identify data sources and existing models, to aid future work. |
| [XSMILES: interactive visualization for molecules, SMILES and XAI attribution scores](https://jcheminf.biomedcentral.com/articles/10.1186/s13321-022-00673-w) | January 2023 | Proposes XSMILES, an interactive visualization technique, to explore explainable artificial intelligence attributions scores and support the interpretation of SMILES. | Has a visual representation of feature importance from each of the SMILES strings.  Users can input any type of score attributed to atom and non-atom tokens and visualize them on top of a 2D molecule diagram coordinated with a bar chart that represents a SMILES string. | Made to support data scientists to develop, improve, and communicate their models by making it easier to identify patterns and compare attributions through interactive exploratory visualization. |
| [Recent development of machine learning models for the prediction of drug-drug interactions](https://pubmed.ncbi.nlm.nih.gov/36748027/) | February 2023 | Covers recent developments in DDI prediction, including the best models that have been made, and new data sources. Mentions Decagon.Discusses further improvements in the field. | Doesn’t specify how it selected its literature for review, but it reviews models that have been created since 2018. | Attempting to identify gaps in current research, specifically for polypharmacy. |
| [An Attentive LSTM based approach for adverse drug reactions prediction](https://link.springer.com/article/10.1007/s10489-022-03721-y) | March 2023 | Introduces a novel approach for ADR prediction, focused on association between ADR’s, and not relying on chemical property data. Also introduces more methods of evaluation model accuracy. For future work, it suggests combining the LSTM model that evaluates ADR data, with a model that evaluates structure data. | Proposes an encoder-decoder framework based on attention mechanism and the long short-term memory (LSTM) model to predict potential ADRs. | To learn better embeddings, they optimize the traditional decoder structure so that the LSTM can receive more information from the attention layer. They were the first to consider ADR prediction as a sequence-to-sequence problem. The model is also based solely on ADR data, which are independent of other classical methods utilizing molecular drug structures. It is therefore easily combined with other methods to provide even more accurate predictions with unknown ADRs. |
| [HyGNN: Drug-Drug Interaction Prediction via Hypergraph Neural Network](https://arxiv.org/abs/2206.12747) | April 2023 | Demonstrates a model that analyses SMILES strings with high accuracy. | Uses a hypergraph GNN model. | Wanted to analyse specifically just structures because not all datasets have extensive information, and most only have structure. GNN has shown promising performance, as well as using SMILES string for prediction. |
| [Deep learning in drug discovery: an integrative review and future challenges](https://link.springer.com/article/10.1007/s10462-022-10306-1) | July 2023 | Gives a review of recent articles and deep learning methods for DDI and DTI, as well as the datasets available. Also discusses the importance of explain-ability in drug discovery, types of XAI, and techniques for it. | Reviews more than 300 articles between 2000 and 2022. The benchmark data sets, the databases, and the evaluation measures are also presented. | Drug discovery has received a lot of attention since it significantly shortens the time and cost of developing new drugs. Deep learning approaches are increasingly being used in all stages of drug development as DL technology advances, and drug-related data grows. |
| [DDI-GCN: Drug-drug interaction prediction via explainable graph convolutional networks](https://www.sciencedirect.com/science/article/abs/pii/S0933365723001549) | October 2023 | [Proposes](http://wengzq-lab.cn/ddi/) an explainable deep learning model for DDI prediction | Utilises a GCN with attention mechanisms. Hosted the resulting model on a web server for anybody to access. | To more accurately predict DDIs, and improve explainability of deep learning DDI models. |
| [An objective metric for Explainable AI: How and why to estimate the degree of explain ability](https://www.sciencedirect.com/science/article/pii/S0950705123006160) | October 2023 | Presents a model agnostic measure of explain ability in machine learning models. | Introduces a metric called Degree of Explain-ability (DoX). It assumes that the degree of explain-ability is directly proportional to the number of relevant questions that a piece of information can correctly answer. They operationalized this concept by formalizing the DoX metric through a mathematical formula. | Aims to fill the gaps that many of the previous review papers in this table highlighted. By giving a measure of explain-ability it’s now easier to asses an interpretable deep learning model. |
| [Application of artificial intelligence and machine learning in early detection of adverse drug reactions (ADRs) and drug-induced toxicity](https://www.sciencedirect.com/science/article/pii/S2949747723000118) | December 2023 | Provides a more recent review of the current state of ADR prediction as a field. Goes into detail about types of ADR’s. Also provides a useful history of ADR’s, talking about major ADR events that began the field. Mentions a lack of interpretability in deep learning methods. Discuses future improvements, including improved accuracy with deep learning methods, and interpretability of black box models. | Doesn’t specify how it selected its literature for review. | Attempting to identify gaps in current research, specifically for polypharmacy. |

Table 2: Literature Review Table

## **B – Data Gathering and Cleaning Code**

### **data\_gathering.py**

import math  
import requests  
import csv  
import re  
  
  
# Function to parse an SDF file  
def parse\_sdf(sdf\_text):  
 # Split the SDF text into lines  
 lines = sdf\_text.strip().split('\n')  
  
 # Initialize a list to store the parsed data  
 parsed\_coords = []  
 parsed\_bonds = []  
 parsed\_charges = []  
 # Flag to indicate parsing the partial charges section  
 parsing\_charges = False  
  
 # Parse the SDF text line by line  
 index = 1  
 for line in lines:  
 # Check if it's an atom line  
 if len(line) >= 60:  
 # Extract atom type and coordinates  
 x = float(line[:10])  
 y = float(line[10:20])  
 z = float(line[20:30])  
 atom\_type = line[31:34].strip()  
 # Append parsed data to the list  
 parsed\_coords.append((index, atom\_type, x, y, z))  
 index += 1  
  
 # Check if it's a bond line  
 elif 20 <= len(line) <= 22:  
 try:  
 # Extract bond information  
 atom1\_idx = int(line[:3])  
 atom2\_idx = int(line[4:7])  
 bond\_type = int(line[8])  
 parsed\_bonds.append((atom1\_idx, atom2\_idx, bond\_type))  
 except ValueError:  
 # Skip lines that don't contain valid data  
 continue  
  
 # Check if it's a charge line  
 elif parsing\_charges:  
 # Check to see if it's the end of the charge information  
 if line.startswith('>'):  
 parsing\_charges = False  
 else:  
 try:  
 # Extract atom index and partial charge  
 atom\_idx, charge = map(float, line.strip().split())  
 parsed\_charges.append((int(atom\_idx), charge))  
 except ValueError:  
 # Skip lines that don't contain valid charge data  
 continue  
 # Check for the beginning of the charges information  
 elif line.startswith('> <PUBCHEM\_MMFF94\_PARTIAL\_CHARGES>'):  
 # Start parsing partial charges  
 parsing\_charges = True  
  
 return parsed\_coords, parsed\_bonds, parsed\_charges  
  
  
# Function to retrieve 3d compound information for a given ID  
def get\_compound\_info(cid):  
 # Set the URL for the PUG REST API  
 smiles\_url = f"https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/cid/{cid}/property/CanonicalSMILES/JSON"  
 # Make a request to the API with that URL  
 smiles\_response = requests.get(smiles\_url)  
 canonical\_smiles = None  
 parsed\_coords = None  
 parsed\_bonds = None  
 parsed\_charges = None  
 # If there is a response then save the SMILES string as canonical\_smiles  
 if smiles\_response.status\_code == 200:  
 smiles\_data = smiles\_response.json()  
 if 'PropertyTable' in smiles\_data:  
 smiles\_property\_table = smiles\_data['PropertyTable']  
 if 'Properties' in smiles\_property\_table and len(smiles\_property\_table['Properties']) > 0:  
 canonical\_smiles = smiles\_property\_table['Properties'][0]['CanonicalSMILES']  
  
 # Make a request to the API for the same compounds 3D information  
 try:  
 # Receive the file as an SDF file  
 sdf\_url = f"https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/cid/{cid}/SDF?record\_type=3d"  
 sdf\_response = requests.get(sdf\_url)  
 if sdf\_response.status\_code == 200:  
 sdf\_text = sdf\_response.text  
 # Parse SDF text to extract atoms, bonds, coordinates, and charge  
 parsed\_coords, parsed\_bonds, parsed\_charges = parse\_sdf(sdf\_text)  
 except:  
 return None  
 return canonical\_smiles, parsed\_coords, parsed\_bonds, parsed\_charges  
  
  
# Function to gather and write all compound info for given compound ID pairs  
def write\_compound\_info(filename, output\_filename):  
 # Open the given file, the name given in 'filename'  
 with open(filename, mode='r') as file:  
 reader = csv.DictReader(file)  
 # Retrieve a list of column names  
 fieldnames = reader.fieldnames[:]  
  
 # Find the index of the "STITCH 1" column  
 stitch1\_index = fieldnames.index('# STITCH 1')  
 # Insert new fieldnames to the right of "STITCH 1" - to hold new 3d values  
 fieldnames.insert(stitch1\_index + 1, 'C1 SMILES')  
 fieldnames.insert(stitch1\_index + 2, 'C1 Coords')  
 fieldnames.insert(stitch1\_index + 3, 'C1 Bonds')  
 fieldnames.insert(stitch1\_index + 4, 'C1 Charges')  
  
 # Find the index of the "STITCH 2" column  
 stitch2\_index = fieldnames.index('STITCH 2')  
 # Insert new fieldnames to the right of "STITCH 2" - to hold new 3d values  
 fieldnames.insert(stitch2\_index + 1, 'C2 SMILES')  
 fieldnames.insert(stitch2\_index + 2, 'C2 Coords')  
 fieldnames.insert(stitch2\_index + 3, 'C2 Bonds')  
 fieldnames.insert(stitch2\_index + 4, 'C2 Charges')  
  
 # Decide which row of the csv to start writing from  
 start\_row = 0  
  
 # Write the new data to a new file, the name given in 'output\_filename'  
 with open(output\_filename, mode='a', newline='') as output\_file:  
 writer = csv.DictWriter(output\_file, fieldnames=fieldnames)  
 # Write the header row only if the file is empty  
 if output\_file.tell() == 0:  
 writer.writeheader()  
  
 last\_stitch1\_id = None  
 last\_stitch2\_id = None  
  
 # Iterate over all rows in the input file  
 for index, row in enumerate(reader):  
 # If the starting row is higher, don't gather data for this row  
 if index < start\_row:  
 continue  
  
 # Get the first compounds ID  
 stitch1\_id = row['# STITCH 1']  
  
 # Gets the compound ID of varying length, removes trailing 0's  
 compound\_id1 = re.search(r'\d+', stitch1\_id).group()  
 compound\_id1 = str(int(compound\_id1))  
  
 # Check to see if the last requested ID was the same as the current requested ID  
 # If it is then it will just use the value from the last time get\_compound\_info was called  
 if stitch1\_id != last\_stitch1\_id:  
 # If it is not then retrieve the compounds information  
 stitch1\_canonical\_smiles, stitch1\_parsed\_coords, stitch1\_parsed\_bonds, stitch1\_parsed\_charges = get\_compound\_info(compound\_id1)  
 if stitch1\_canonical\_smiles is None:  
 # If no data is returned from the API then print an error  
 print("No data found for", stitch1\_id)  
 continue  
 else:  
 # Set the last requested ID  
 last\_stitch1\_id = stitch1\_id  
  
 # Insert the information to the new columns  
 row['C1 SMILES'] = stitch1\_canonical\_smiles  
 row['C1 Coords'] = stitch1\_parsed\_coords  
 row['C1 Bonds'] = stitch1\_parsed\_bonds  
 row['C1 Charges'] = stitch1\_parsed\_charges  
  
 # Get the first compounds ID  
 stitch2\_id = row['STITCH 2']  
  
 # Gets the compound ID of varying length, removes trailing 0's  
 compound\_id2 = re.search(r'\d+', stitch2\_id).group()  
 compound\_id2 = str(int(compound\_id2))  
  
 # Check to see if the last requested ID was the same as the current requested ID  
 # If it is then it will just use the value from the last time get\_compound\_info was called  
 if stitch2\_id != last\_stitch2\_id:  
 # If it is not then retrieve the compounds information  
 stitch2\_canonical\_smiles, stitch2\_parsed\_coords, stitch2\_parsed\_bonds, stitch2\_parsed\_charges = get\_compound\_info(compound\_id2)  
 if stitch2\_canonical\_smiles is None:  
 # If no data is returned from the API then print an error  
 print("No data found for", stitch2\_id)  
 continue  
 else:  
 # Set the last requested ID  
 last\_stitch2\_id = stitch2\_id  
  
 # Insert the information to the new columns  
 row['C2 SMILES'] = stitch2\_canonical\_smiles  
 row['C2 Coords'] = stitch2\_parsed\_coords  
 row['C2 Bonds'] = stitch2\_parsed\_bonds  
 row['C2 Charges'] = stitch2\_parsed\_charges  
  
 # Write the compound pair to the output file  
 writer.writerow(row)  
 print("File complete!")  
  
  
# Function to calculate distance between two atoms  
def calculate\_distance(atom1\_coords, atom2\_coords):  
 x1, y1, z1 = atom1\_coords  
 x2, y2, z2 = atom2\_coords  
 # Euclidian distance formula  
 distance = math.sqrt((x2 - x1)\*\*2 + (y2 - y1)\*\*2 + (z2 - z1)\*\*2)  
 return distance  
  
  
# Function to parse string and convert to list of tuples  
def parse\_list(str):  
 # Remove brackets and parentheses  
 str = str.strip("[]()")  
 # Split by "), ("  
 str\_list = str.split("), (")  
 # Use eval to convert each coordinate string to a tuple  
 str\_list = [eval(item) for item in str\_list]  
 return str\_list  
  
  
# Function to calculate bond lengths for all compounds pairs in cleanedData.csv  
def calculate\_bond\_lengths(filename, output\_filename):  
  
 # Open the given file, the name given in 'filename'  
 with open(filename, mode='r') as file:  
 reader = csv.DictReader(file)  
 # Retrieve a list of column names  
 fieldnames = reader.fieldnames[:]  
  
 # Find the index of the "STITCH 1" column (first compound ID column)  
 stitch\_index1 = fieldnames.index('# STITCH 1')  
 # Insert new fieldnames for the computed lengths  
 fieldnames.insert(stitch\_index1 + 5, 'C1 Computed Lengths')  
 fieldnames.insert(stitch\_index1 + 10, 'C2 Computed Lengths')  
  
 # Write the new data to a new file, the name given in 'output\_file'  
 with open(output\_filename, mode='a', newline='') as output\_file:  
 writer = csv.DictWriter(output\_file, fieldnames=fieldnames)  
 # Write the header row only if the file is empty  
 if output\_file.tell() == 0:  
 writer.writeheader()  
  
 # Iterate over each row in the file being read  
 for index, row in enumerate(reader):  
  
 # Get the coordinates of the compounds  
 c1\_atom\_coords = parse\_list(row['C1 Coords'])  
 c2\_atom\_coords = parse\_list(row['C2 Coords'])  
  
 # Get the bonds between the compounds atoms  
 c1\_atom\_bonds = parse\_list(row['C1 Bonds'])  
 c2\_atom\_bonds = parse\_list(row['C2 Bonds'])  
  
 # Calculate the first compounds bond lengths  
 c1\_distances = []  
 # Loop over each bond in the compounds list of bonds  
 for bond1 in c1\_atom\_bonds:  
 # Retrieve the indexes of the atoms for the given bond, retrieving their coordinates  
 atom1\_index, atom2\_index, \_ = bond1  
 atom1\_coords = c1\_atom\_coords[atom1\_index - 1][2:]  
 atom2\_coords = c1\_atom\_coords[atom2\_index - 1][2:]  
  
 # Calculate their euclidian distance, and add it to the list of distances  
 distance = calculate\_distance(atom1\_coords, atom2\_coords)  
 c1\_distances.append((atom1\_index, atom2\_index, distance))  
  
 # Calculate the second compounds bond lengths  
 c2\_distances = []  
 # Loop over each bond in the compounds list of bonds  
 for bond2 in c2\_atom\_bonds:  
 # Retrieve the indexes of the atoms for the given bond, retrieving their coordinates  
 atom1\_index, atom2\_index, \_ = bond2  
 atom1\_coords = c2\_atom\_coords[atom1\_index - 1][2:]  
 atom2\_coords = c2\_atom\_coords[atom2\_index - 1][2:]  
  
 # Calculate their euclidian distance, and add it to the list of distances  
 distance = calculate\_distance(atom1\_coords, atom2\_coords)  
 c2\_distances.append((atom1\_index, atom2\_index, distance))  
  
 # Insert the information to the new columns  
 row['C1 Computed Lengths'] = c1\_distances  
 row['C2 Computed Lengths'] = c2\_distances  
  
 # Write the whole row to the new file  
 writer.writerow(row)  
 print("Distances calculated!")  
  
  
'''  
# Testing with Aspirin:  
canonical\_smiles, parsed\_coords, parsed\_bonds, parsed\_charges = get\_compound\_info(2244)  
print("Aspirin Canonical SMILES:", canonical\_smiles)  
print("Aspirin Atom Info:", parsed\_coords)  
print("Aspirin Bond Info:", parsed\_bonds)  
print("Paracetamol Charge Info:", parsed\_charges)  
print("\n")  
'''

'''WARNING: The below statements will allow for data gathering and calculating. However, this file is also imported by  
 gnn\_3d. If you leave these statements uncommented, then gnn\_3d will attempt to run them when it imports them.  
 It is therefore recommended to avoid leaving them uncommented when not directly testing these functions.'''

# RUN THIS TO WRITE THE COMPOUND INFO  
write\_compound\_info("ChChSe-Decagon\_polypharmacy/ChChSe-Decagon\_polypharmacy.csv", "ChChSe-Decagon\_polypharmacy/gatheredData.csv")  
  
# ADD THE BOND LENGTHS  
calculate\_bond\_lengths("ChChSe-Decagon\_polypharmacy/cleanedData.csv", "ChChSe-Decagon\_polypharmacy/computedData.csv")

### **descriptive\_analytics.py**

import pandas as pd  
  
  
# Function to count the number of unique side effects in the dataset  
def count\_side\_effects(filename):  
 # Load the CSV file into a DataFrame  
 df = pd.read\_csv(filename)  
  
 # Count the number of unique values in the side effect column  
 unique\_side\_effects = df["Side Effect Name"].nunique()  
  
 print(f"Unique side effects: {unique\_side\_effects}")  
  
  
# Function to count the amount of drug pairs for each side effect  
def count\_unique\_values(filename):  
 # Load the CSV file into a DataFrame  
 df = pd.read\_csv(filename)  
  
 # Count the number of samples for each unique value in the side effect column  
 unique\_values\_count = df["Side Effect Name"].value\_counts()  
  
 # Display unique values and their counts  
 print("Unique Values and Their Counts:")  
 for id, (value, count) in enumerate(unique\_values\_count.items(), start=0):  
 print(f"{value} (Count: {count})")  
  
  
# Function to filter the side effects with few samples  
def filter\_data(filename):  
 # Read CSV file into a DataFrame  
 df = pd.read\_csv(filename)  
  
 # Count unique values in the side effect column  
 counts = df.iloc[:, 13].value\_counts()  
  
 # Filter for the top 3 side effects  
 filtered\_df = df[df.iloc[:, 13].isin(counts[counts >= 4200].index)]  
  
 # Write filtered data to a new CSV file  
 filtered\_df.to\_csv("ChChSe-Decagon\_polypharmacy/filteredData.csv", index=False)  
  
  
# Call the functions to count unique values  
count\_side\_effects("ChChSe-Decagon\_polypharmacy/computedData.csv")  
count\_unique\_values("ChChSe-Decagon\_polypharmacy/computedData.csv")  
  
#Filter the data  
filter\_data("ChChSe-Decagon\_polypharmacy/computedData.csv")

## **C – Model Code**

### **graph\_creation.py**

import ast  
import networkx as nx  
import torch  
from torch\_geometric.data import Data  
  
  
# Function to create atom graph structures from coordinate lists of compounds  
def create\_coordinate\_graphs(compound\_list):  
 # List to store the atom graphs for each compound  
 atom\_graphs = []  
  
 # Iterate over each compound string in the list  
 for compound in compound\_list:  
 # Create an empty graph for the current compound  
 atom\_graph = nx.Graph()  
  
 # Parse the string into a list of tuples to get each atom and their coordinates  
 atom\_list = ast.literal\_eval(compound)  
  
 # Iterate over each atom in the compound  
 for atom in atom\_list:  
 # Get the atoms information  
 index, atom\_type, x, y, z = atom  
 # Add node for each atom to the current atom graph  
 atom\_graph.add\_node(index, atom\_type=atom\_type, x=x, y=y, z=z)  
  
 # Append the current atom graph to the list of atom graphs  
 atom\_graphs.append(atom\_graph)  
  
 return atom\_graphs  
  
  
# Function to add charges to the atom graphs  
def add\_charges(atom\_graphs, compound\_charges):  
 # Iterate over each pair of graph and charges (they start from the same index, same amount of rows)  
 for graph, charges in zip(atom\_graphs, compound\_charges):  
 # Parse the string into a list of tuples to get each atom index/charge pair  
 charges\_list = ast.literal\_eval(charges)  
  
 # Iterate over each atom charge pair  
 for index, charge in charges\_list:  
 # Check if the node with given index exists in the graph  
 if index in graph.nodes:  
 # Add charge attribute to the node  
 graph.nodes[index]['charge'] = charge  
  
 return atom\_graphs  
  
  
# Function to add bonds to the atom graphs  
def add\_bonds(atom\_graphs, compound\_bonds):  
 # Iterate over each pair of graph and bonds (they start from the same index, same amount of rows)  
 for graph, bonds in zip(atom\_graphs, compound\_bonds):  
 # Parse the string into a list of tuples to get all atom index's/bond type pairs  
 bonds\_list = ast.literal\_eval(bonds)  
  
 # Iterate over each bond  
 for source\_index, target\_index, bond\_value in bonds\_list:  
 # Add edge between nodes with the specified bond value  
 graph.add\_edge(source\_index, target\_index, bond=bond\_value)  
 return atom\_graphs  
  
  
# Function to add lengths to the bonds of the atom graphs  
def add\_lengths(atom\_graphs, compound\_lengths):  
 # Iterate over each pair of graph and lengths (they start from the same index, same amount of rows)  
 for graph, lengths in zip(atom\_graphs, compound\_lengths):  
 # Parse the string into a list of tuples to get all atom index's/bond length pairs  
 lengths\_list = ast.literal\_eval(lengths)  
  
 # Iterate over each length  
 for source\_index, target\_index, length\_value in lengths\_list:  
 # Check if the edge exists in the graph  
 if graph.has\_edge(source\_index, target\_index):  
 # Add 'length' attribute to the edge  
 graph[source\_index][target\_index]['length'] = length\_value  
 return atom\_graphs  
  
  
# Function to turn graphs into data representations  
def graph\_to\_data(graph):  
 # Extract node features and coordinates  
 node\_features = []  
 # Iterate over each node of the graph and their properties  
 for node\_id, data in graph.nodes(data=True):  
 # Append the nodes properties to the list  
 node\_features.append([data.get('x', 0), data.get('y', 0), data.get('z', 0), data.get('charge', 0)])  
  
 # Assuming graph is a NetworkX graph  
 edge\_index = []  
 edge\_attributes = []  
  
 # Iterate over edges and extract weights, lengths, and edge indices  
 for idx, (source, target, data) in enumerate(graph.edges(data=True)):  
 # Extract length if available, otherwise default to 0  
 # We're only adding length to the edge attributes because GCNConv only accepts a tensor with 1 dimension  
 # --So no bond type--  
 length = data.get('length', 0)  
  
 # Append a tuple containing both weight and length to edge\_attributes  
 edge\_attributes.append(length)  
  
 # Append edge indices to edge\_index  
 edge\_index.append([source, target])  
  
 # Convert lists to tensors  
 node\_features = torch.tensor(node\_features, dtype=torch.float)  
 edge\_index = torch.tensor(edge\_index, dtype=torch.long).t().contiguous()  
 edge\_attributes = torch.tensor(edge\_attributes, dtype=torch.float)  
  
 return node\_features, edge\_index, edge\_attributes  
  
  
# Function to concatenate data representations and turn them into data objects  
def concatenate\_data(atom\_graphs1, atom\_graphs2, side\_effects, side\_effect\_to\_idx):  
  
 # Create list to store graph data  
 graphs\_data = []  
 # For each drug pair, create Data objects for each drug pair  
 for idx, (graphs1, graphs2, side\_effect) in enumerate(zip(atom\_graphs1, atom\_graphs2, side\_effects)):  
  
 # Convert graphs to data representations  
 features1, edge\_index1, edge\_attributes1 = graph\_to\_data(graphs1)  
 features2, edge\_index2, edge\_attributes2 = graph\_to\_data(graphs2)  
  
 if features1 is not None and features2 is not None:  
 # Concatenate features and edge indexes  
 concatenated\_features = torch.cat((features1, features2), dim=0)  
 # Concatenate the padded edge indices  
 concatenated\_edge\_index = torch.cat([edge\_index1, edge\_index2], dim=1)  
 # Concatenate edge attributes  
 concatenated\_edge\_attributes = torch.cat([edge\_attributes1, edge\_attributes2], dim=0)  
  
 # Make the side effect into a tensor label  
 label = torch.tensor([side\_effect\_to\_idx[side\_effect]])  
  
 # Create PyTorch Geometric Data object  
 data = Data(x=concatenated\_features,  
 edge\_index=concatenated\_edge\_index,  
 edge\_attr=concatenated\_edge\_attributes,  
 y=label)  
  
 # Append to graphs\_data list  
 graphs\_data.append(data)  
  
 return graphs\_data

### **gnn\_3d.py**

import sys  
import random  
import re  
import numpy as np  
import pandas as pd  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from PyQt5.QtWidgets import QMainWindow, QApplication, QMessageBox  
from sklearn.metrics import accuracy\_score, classification\_report  
from torch\_geometric.graphgym import optim  
from torch\_geometric.nn import GCNConv, global\_mean\_pool  
from torch\_geometric.data import Data  
import plotly.graph\_objs as go  
from ModelView import Ui\_ModelView  
from data\_gathering import get\_compound\_info, calculate\_distance  
import graph\_creation  
  
  
# Class for the 3D-DDI-GCN Model  
class GCNGraphClassifier(nn.Module):  
 def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):  
 super(GCNGraphClassifier, self).\_\_init\_\_()  
 self.conv1 = GCNConv(input\_dim, hidden\_dim)  
 self.conv2 = GCNConv(hidden\_dim, hidden\_dim)  
 self.conv3 = GCNConv(hidden\_dim, hidden\_dim)  
 self.attention = nn.Linear(hidden\_dim, 1)  
 self.fc = nn.Linear(hidden\_dim, output\_dim)  
  
 def forward(self, data):  
 x, edge\_index, edge\_attributes, batch = data.x, data.edge\_index, data.edge\_attr, data.batch  
  
 # Apply convolutions and ReLu  
 x = self.conv1(x, edge\_index, edge\_attributes)  
 x = F.relu(x)  
 x = self.conv2(x, edge\_index, edge\_attributes)  
 x = F.relu(x)  
 x = self.conv3(x, edge\_index, edge\_attributes)  
 x = F.relu(x)  
  
 # Compute attention weights  
 attention\_weights = F.softmax(self.attention(x), dim=0)  
 # Weighted sum of node embeddings  
 x\_weighted = torch.matmul(x.t(), attention\_weights).squeeze()  
  
 # Pool the node level embeddings learned from the hidden layers  
 x = global\_mean\_pool(x, batch)  
 x = self.fc(x)  
  
 return F.log\_softmax(x, dim=1), attention\_weights, x\_weighted  
  
  
# Class to define the model view  
class ModelView(QMainWindow):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 # Set up the user interface from PyQT Designer  
 self.ui = Ui\_ModelView()  
 self.ui.setupUi(self)  
  
 # Connect the clicked signal of predictButton to 'predict\_side\_effect'  
 self.ui.predictButton.clicked.connect(self.predict\_side\_effect)  
  
 # Function to predict the side effect for a given drug pair  
 def predict\_side\_effect(self):  
 # Retrieve input from Id1 and Id2 input fields  
 Id1 = self.ui.Id1.text()  
 Id2 = self.ui.Id2.text()  
  
 # Check if either of the input fields is empty, if so send an error message  
 if not Id1 or not Id2:  
 msg = QMessageBox()  
 msg.setText("Please fill in both ID fields.")  
 msg.setStyleSheet("background-color: rgb(71, 52, 209);color:white;")  
 msg.exec()  
 else:  
 # Otherwise, begin the prediction  
 # Get the compound ID of varying length, remove trailing 0's  
 try:  
 Id1 = str(int(re.search(r'\d+', Id1).group()))  
 Id2 = str(int(re.search(r'\d+', Id2).group()))  
 except:  
 print("User entered invalid Ids")  
  
 # Gather data for both Ids  
 result1 = get\_compound\_info(Id1)  
 result2 = get\_compound\_info(Id2)  
 smiles1, coords1, bonds1, charges1 = result1  
 smiles2, coords2, bonds2, charges2 = result2  
  
 # Check if 3d data is returned for both Id's  
 if coords1 is None or coords2 is None:  
 # If not, prompt an error  
 if coords1 is None:  
 msg = QMessageBox()  
 msg.setText("First ID is invalid. Please enter a valid ID.")  
 msg.setStyleSheet("background-color: rgb(71, 52, 209);color:white;")  
 msg.exec()  
 if coords2 is None:  
 msg = QMessageBox()  
 msg.setText("Second ID is invalid. Please enter a valid ID")  
 msg.setStyleSheet("background-color: rgb(71, 52, 209);color:white;")  
 msg.exec()  
 else:  
 # Calculate the first compounds bond lengths  
 lengths1 = []  
 for bond1 in bonds1:  
 atom1\_index, atom2\_index, \_ = bond1  
 atom1\_coords = coords1[atom1\_index - 1][2:]  
 atom2\_coords = coords1[atom2\_index - 1][2:]  
 distance = calculate\_distance(atom1\_coords, atom2\_coords)  
 lengths1.append((atom1\_index, atom2\_index, distance))  
  
 # Calculate the second compounds bond lengths  
 lengths2 = []  
 for bond2 in bonds2:  
 atom1\_index, atom2\_index, \_ = bond2  
 atom1\_coords = coords2[atom1\_index - 1][2:]  
 atom2\_coords = coords2[atom2\_index - 1][2:]  
 distance = calculate\_distance(atom1\_coords, atom2\_coords)  
 lengths2.append((atom1\_index, atom2\_index, distance))  
  
 # Construct atom graphs for compound 1, and add charges, bonds, length  
 graph1 = graph\_creation.create\_coordinate\_graphs([str(coords1)])  
 graph1 = graph\_creation.add\_charges(graph1, [str(charges1)])  
 graph1 = graph\_creation.add\_bonds(graph1, [str(bonds1)])  
 graph1 = graph\_creation.add\_lengths(graph1, [str(lengths1)])[0]  
  
 # Construct atom graphs for compound 2, and add charges, bonds, length  
 graph2 = graph\_creation.create\_coordinate\_graphs([str(coords2)])  
 graph2 = graph\_creation.add\_charges(graph2, [str(charges2)])  
 graph2 = graph\_creation.add\_bonds(graph2, [str(bonds2)])  
 graph2 = graph\_creation.add\_lengths(graph2, [str(lengths2)])[0]  
  
 # Convert the graphs to data objects  
 features1, edge\_index1, edge\_attributes1 = graph\_creation.graph\_to\_data(graph1)  
 features2, edge\_index2, edge\_attributes2 = graph\_creation.graph\_to\_data(graph2)  
  
 input\_data = None  
 # Concatenate the data objects  
 if features1 is not None and features2 is not None:  
 # Concatenate features and edge indexes  
 concatenated\_features = torch.cat((features1, features2), dim=0)  
 # Concatenate the padded edge indices  
 concatenated\_edge\_index = torch.cat([edge\_index1, edge\_index2], dim=1)  
 # Concatenate edge attributes  
 concatenated\_edge\_attributes = torch.cat([edge\_attributes1, edge\_attributes2], dim=0)  
  
 # Create PyTorch Geometric Data object  
 input\_data = Data(x=concatenated\_features,  
 edge\_index=concatenated\_edge\_index,  
 edge\_attr=concatenated\_edge\_attributes)  
  
 # Make a prediction using the trained model - retrieve the predicted label  
 output, attention\_weights, x\_weighted = model(input\_data)  
 \_, predicted\_label = torch.max(output, 1)  
  
 # Create a new dictionary with reversed key-value pairs  
 idx\_to\_side\_effect = {v: k for k, v in side\_effect\_to\_idx.items()}  
  
 # Display the predicted side effect in the UI  
 self.ui.Prediction.setText(f"Prediction: {idx\_to\_side\_effect.get(predicted\_label.item())}")  
  
 # Get the information for the node weights  
 # Convert attention weights tensor to numpy array for easier manipulation  
 attention\_weights\_np = attention\_weights.detach().cpu().numpy()  
 # Find the maximum attention weight  
 max\_attention\_weight = np.max(attention\_weights\_np)  
 # Identify nodes with maximum attention weight  
 nodes\_with\_max\_attention = np.where(attention\_weights\_np == max\_attention\_weight)[0]  
 # Indexes for graphs start at 1 VS nodes\_with\_max\_attention that starts at 0  
 important\_nodes = nodes\_with\_max\_attention + 1  
  
 # Open the windows which visualise the nodes  
 visualise\_3d(graph1, graph2, important\_nodes)  
  
  
# Function to train the 3D-DDI-GCN model  
def train\_model(train\_data):  
 # Training loop  
 num\_epochs = 10  
 for epoch in range(num\_epochs):  
 total\_loss = 0  
  
 # Set the model to training mode  
 model.train()  
  
 # Pass each data object in the training data to the model  
 for graph\_data in train\_data:  
 # Forward pass  
 output = model(graph\_data)[0]  
  
 # Calculate the loss  
 loss = criterion(output, graph\_data.y)  
 total\_loss += loss.item()  
  
 # Backpropagation  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 # Print average loss  
 print(f"Epoch [{epoch + 1}/{num\_epochs}], Training Loss: {total\_loss / len(train\_data)}")  
  
  
# Function to evaluate the 3D-DDI-GCN model  
def evaluate\_model(test\_data):  
 # Evaluation loop  
 with torch.no\_grad():  
 # Set the model to evaluation mode  
 model.eval()  
 true\_labels = []  
 predicted\_labels = []  
  
 # Pass each data object in the test data to the model, and get the predicted VS true labels  
 for graph\_data in test\_data:  
 output = model(graph\_data)[0]  
 \_, predicted = torch.max(output, 1)  
 true\_labels.extend(graph\_data.y.tolist())  
 predicted\_labels.extend(predicted.tolist())  
  
 # Calculate and print the models accuracy  
 accuracy = accuracy\_score(true\_labels, predicted\_labels)  
 print(f"Accuracy: {accuracy}")  
  
 # Calculate precision, recall, and F1-score  
 report = classification\_report(true\_labels, predicted\_labels, zero\_division=0)  
 print(report)  
  
  
# Function to visualise 3D-DDI-GCN's predictions  
def visualise\_3d(first\_graph, second\_graph, important\_nodes):  
 # Extract node positions, types and charges for first\_graph  
 node\_type\_first = [data['atom\_type'] for \_, data in first\_graph.nodes(data=True)]  
 node\_x\_first = [data['x'] for \_, data in first\_graph.nodes(data=True)]  
 node\_y\_first = [data['y'] for \_, data in first\_graph.nodes(data=True)]  
 node\_z\_first = [data['z'] for \_, data in first\_graph.nodes(data=True)]  
 node\_charge\_first = [data.get('charge', '0') for \_, data in first\_graph.nodes(data=True)]  
  
 # Define colors for nodes based on whether their 'index' property is in the important nodes list  
 node\_colors\_first = ['yellow' if node\_id in important\_nodes else 'purple' for node\_id, \_ in  
 first\_graph.nodes(data=True)]  
  
 # Create node traces for first\_graph with color based on node index  
 node\_trace\_first = go.Scatter3d(  
 x=node\_x\_first,  
 y=node\_y\_first,  
 z=node\_z\_first,  
 mode='markers',  
 marker=dict(  
 size=8,  
 color=node\_colors\_first,  
 line=dict(color='rgb(0,0,0)', width=1)  
 ),  
 hoverinfo='text',  
 hovertext=[f"Node ID: {node\_id}<br>Type: {atom\_type}<br>X: {x}<br>Y: {y}<br>Z: {z}<br>Charge: {charge}"  
 for node\_id, atom\_type, x, y, z, charge in  
 zip(first\_graph.nodes(), node\_type\_first, node\_x\_first, node\_y\_first, node\_z\_first,  
 node\_charge\_first)]  
 )  
  
 # Create edge traces for first\_graph  
 edge\_trace\_first = []  
 for edge in first\_graph.edges(data=True):  
 start = edge[0]  
 end = edge[1]  
 bond\_value = edge[2].get('bond', 'N/A')  
  
 # Adjust node IDs to start from 0-based indexing  
 start\_index = start - 1  
 end\_index = end - 1  
  
 # Ensure node IDs are within the valid range  
 if start\_index in range(len(node\_x\_first)) and end\_index in range(len(node\_x\_first)):  
 edge\_trace = go.Scatter3d(  
 x=[node\_x\_first[start\_index], node\_x\_first[end\_index]],  
 y=[node\_y\_first[start\_index], node\_y\_first[end\_index]],  
 z=[node\_z\_first[start\_index], node\_z\_first[end\_index]],  
 mode='lines',  
 line=dict(color='rgb(125,125,125)', width=2),  
 hoverinfo='text',  
 hovertext=f"Start: {start}<br>End: {end}<br>Bond: {bond\_value}<br>Length: {edge[2].get('length', 'N/A')}"  
 )  
 edge\_trace\_first.append(edge\_trace)  
 else:  
 print(f"Issue with node IDs in edge: {start} -> {end}")  
  
 # Extract node positions, types and charges for second\_graph  
 node\_type\_second = [data['atom\_type'] for \_, data in second\_graph.nodes(data=True)]  
 node\_x\_second = [data['x'] for \_, data in second\_graph.nodes(data=True)]  
 node\_y\_second = [data['y'] for \_, data in second\_graph.nodes(data=True)]  
 node\_z\_second = [data['z'] for \_, data in second\_graph.nodes(data=True)]  
 node\_charge\_second = [data.get('charge', '0') for \_, data in second\_graph.nodes(data=True)]  
  
 # Account for the length of the first graph (the important nodes assumes both graphs are concatenated  
 important\_nodes\_second = [node - len(first\_graph.nodes(data=True)) for node in important\_nodes]  
  
 # Define colors for nodes based on whether their 'index' property matches the number  
 node\_colors\_second = ['yellow' if node\_id in important\_nodes\_second else 'purple' for node\_id, \_ in  
 second\_graph.nodes(data=True)]  
  
 # Create node traces for second\_graph with color based on node index  
 node\_trace\_second = go.Scatter3d(  
 x=node\_x\_second,  
 y=node\_y\_second,  
 z=node\_z\_second,  
 mode='markers',  
 marker=dict(  
 size=8,  
 color=node\_colors\_second,  
 line=dict(color='rgb(0,0,0)', width=1)  
 ),  
 hoverinfo='text',  
 hovertext=[f"Node ID: {node\_id}<br>Type: {atom\_type}<br>X: {x}<br>Y: {y}<br>Z: {z}<br>Charge: {charge}"  
 for node\_id, atom\_type, x, y, z, charge in  
 zip(second\_graph.nodes(), node\_type\_second, node\_x\_second, node\_y\_second, node\_z\_second,  
 node\_charge\_second)]  
 )  
  
 # Create edge traces for second\_graph  
 edge\_trace\_second = []  
 for edge in second\_graph.edges(data=True):  
 start = edge[0]  
 end = edge[1]  
 bond\_value = edge[2].get('bond', 'N/A')  
  
 # Adjust node IDs to start from 0-based indexing  
 start\_index = start - 1  
 end\_index = end - 1  
  
 # Ensure node IDs are within the valid range  
 if start\_index in range(len(node\_x\_second)) and end\_index in range(len(node\_x\_second)):  
 edge\_trace = go.Scatter3d(  
 x=[node\_x\_second[start\_index], node\_x\_second[end\_index]],  
 y=[node\_y\_second[start\_index], node\_y\_second[end\_index]],  
 z=[node\_z\_second[start\_index], node\_z\_second[end\_index]],  
 mode='lines',  
 line=dict(color='rgb(125,125,125)', width=2),  
 hoverinfo='text',  
 hovertext=f"Start: {start}<br>End: {end}<br>Bond: {bond\_value}<br>Length: {edge[2].get('length', 'N/A')}"  
 )  
 edge\_trace\_second.append(edge\_trace)  
 else:  
 print(f"Issue with node IDs in edge: {start} -> {end}")  
  
 # Create the visualisation layout  
 layout = go.Layout(  
 title='3D Graph Visualization',  
 showlegend=False,  
 scene=dict(  
 xaxis=dict(title='X'),  
 yaxis=dict(title='Y'),  
 zaxis=dict(title='Z'),  
 ),  
 margin=dict(  
 l=0,  
 r=0,  
 b=0,  
 t=0  
 ),  
 hovermode='closest'  
 )  
  
 # Combine traces and create figure for first\_graph  
 fig\_first = go.Figure(data=[node\_trace\_first] + edge\_trace\_first, layout=layout)  
 # Combine traces and create figure for second\_graph  
 fig\_second = go.Figure(data=[node\_trace\_second] + edge\_trace\_second, layout=layout)  
  
 # Show plots in two tabs  
 fig\_first.show()  
 fig\_second.show()  
  
  
# Read the dataset  
df = pd.read\_csv("ChChSe-Decagon\_polypharmacy/filteredData.csv")  
  
# Extract 3d compound information and side effects  
coords1 = df['C1 Coords']  
charges1 = df['C1 Charges']  
bonds1 = df['C1 Bonds']  
lengths1 = df['C1 Computed Lengths']  
  
coords2 = df['C2 Coords']  
charges2 = df['C2 Charges']  
bonds2 = df['C2 Bonds']  
lengths2 = df['C2 Computed Lengths']  
  
side\_effects = df["Side Effect Name"]  
  
# Create atom graphs for the first compound  
atom\_graphs1 = graph\_creation.create\_coordinate\_graphs(coords1)  
atom\_graphs1 = graph\_creation.add\_charges(atom\_graphs1, charges1)  
atom\_graphs1 = graph\_creation.add\_bonds(atom\_graphs1, bonds1)  
atom\_graphs1 = graph\_creation.add\_lengths(atom\_graphs1, lengths1)  
  
# Create atom graphs for the second compound  
atom\_graphs2 = graph\_creation.create\_coordinate\_graphs(coords2)  
atom\_graphs2 = graph\_creation.add\_charges(atom\_graphs2, charges2)  
atom\_graphs2 = graph\_creation.add\_bonds(atom\_graphs2, bonds2)  
atom\_graphs2 = graph\_creation.add\_lengths(atom\_graphs2, lengths2)  
  
# Map side effects to class indices  
unique\_side\_effects = side\_effects.unique()  
side\_effect\_to\_idx = {side\_effect: idx for idx, side\_effect in enumerate(unique\_side\_effects)}  
  
# Concatenate the graph data  
graphs\_data = graph\_creation.concatenate\_data(atom\_graphs1, atom\_graphs2, side\_effects, side\_effect\_to\_idx)  
  
# Assign random seeds  
random.seed(42)  
torch.manual\_seed(42)  
  
# Split data into train and test sets  
random.shuffle(graphs\_data)  
split\_index = int(0.8 \* len(graphs\_data)) # 80% train, 20% test  
train\_data = graphs\_data[:split\_index]  
test\_data = graphs\_data[split\_index:]  
  
# Define the dimensions  
input\_dim = 4  
hidden\_dim = 32  
output\_dim = len(unique\_side\_effects)  
  
# Create an instance of the model  
model = GCNGraphClassifier(input\_dim, hidden\_dim, output\_dim)  
# Define the loss function  
criterion = nn.CrossEntropyLoss()  
# Define the optimizer  
optimizer = optim.Adam(model.parameters())  
  
# Test and train the model - WARNING: Train/test can take up to 20 minutes  
train\_model(train\_data)  
evaluate\_model(test\_data)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Load the pyqt view  
 app = QApplication(sys.argv)  
 main\_window = ModelView()  
 main\_window.show()  
 sys.exit(app.exec\_())

### **visualise\_3d.py**

import plotly.graph\_objs as go  
import networkx as nx  
  
# Create a graph  
G = nx.Graph()  
  
# Add nodes with multiple attributes (dummy data)  
G.add\_nodes\_from([(1, {'pos': (0, 0, 0), 'info': {'label': 'Node 1', 'color': 'blue'}}),  
 (2, {'pos': (1, 1, 1), 'info': {'label': 'Node 2', 'color': 'red'}}),  
 (3, {'pos': (2, 2, 2), 'info': {'label': 'Node 3', 'color': 'green'}}),  
 (4, {'pos': (3, 3, 3), 'info': {'label': 'Node 4', 'color': 'orange'}})])  
  
# Add edges with attributes  
G.add\_edges\_from([(1, 2, {'weight': 2}),  
 (2, 3, {'weight': 3}),  
 (3, 4, {'weight': 4}),  
 (4, 1, {'weight': 1})])  
  
# Extract node positions  
node\_pos = nx.get\_node\_attributes(G, 'pos')  
  
# Create node traces  
node\_trace = go.Scatter3d(  
 x=[pos[0] for pos in node\_pos.values()],  
 y=[pos[1] for pos in node\_pos.values()],  
 z=[pos[2] for pos in node\_pos.values()],  
 mode='markers',  
 marker=dict(  
 size=8,  
 line=dict(color='rgb(0,0,0)', width=1)  
 ),  
 hoverinfo='text',  
 hovertext=[node[1]['info'] for node in G.nodes(data=True)]  
)  
  
# Create edge traces  
edge\_traces = []  
for edge in G.edges(data=True):  
 start = node\_pos[edge[0]]  
 end = node\_pos[edge[1]]  
 weight = edge[2]['weight']  
 edge\_trace = go.Scatter3d(  
 x=[start[0], end[0]],  
 y=[start[1], end[1]],  
 z=[start[2], end[2]],  
 mode='lines',  
 line=dict(color='rgb(125,125,125)', width=2),  
 hoverinfo='text',  
 hovertext=f"Start: {edge[0]}, End: {edge[1]}, Weight: {weight}"  
 )  
 edge\_traces.append(edge\_trace)  
  
# Create layout  
layout = go.Layout(  
 title='3D Graph Visualization',  
 showlegend=False,  
 scene=dict(  
 xaxis=dict(title='X'),  
 yaxis=dict(title='Y'),  
 zaxis=dict(title='Z'),  
 ),  
 margin=dict(  
 l=0,  
 r=0,  
 b=0,  
 t=0  
 ),  
 hovermode='closest' # Enables the hover mode closest to the point  
)  
  
# Combine traces  
fig = go.Figure(data=[node\_trace] + edge\_traces, layout=layout)  
  
# Update the figure's layout to include a click event handler  
fig.update\_layout(  
 clickmode='event+select'  
)  
  
# Plot 3d visualisation  
fig.show()

### **lstm.py**

import pandas as pd  
from keras.src.layers import Dropout  
from keras.src.utils import to\_categorical  
from sklearn.metrics import recall\_score, precision\_score, f1\_score, confusion\_matrix  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import LabelEncoder  
from keras.models import Sequential  
from keras.layers import LSTM, Embedding, Dense, Bidirectional  
from keras\_preprocessing.text import Tokenizer  
from keras\_preprocessing.sequence import pad\_sequences  
  
# Read the dataset  
df = pd.read\_csv("ChChSe-Decagon\_polypharmacy/filteredData.csv")  
  
# Extract features and target variable  
smiles = df['C1 SMILES'] + ' ' + df['C2 SMILES']  
side\_effects = df["Side Effect Name"]  
  
# Tokenize concatenated SMILES strings  
tokenizer = Tokenizer()  
tokenizer.fit\_on\_texts(smiles)  
X\_seq = tokenizer.texts\_to\_sequences(smiles)  
  
# Pad sequences to ensure equal length  
max\_length = max(map(len, X\_seq))  
X\_padded = pad\_sequences(X\_seq, maxlen=max\_length, padding='post')  
  
# Encode side effects  
label\_encoder = LabelEncoder()  
side\_effects\_encoded = label\_encoder.fit\_transform(side\_effects)  
  
# One-hot encode the target variable  
y\_encoded = to\_categorical(side\_effects\_encoded, num\_classes=len(label\_encoder.classes\_))  
  
# Split data into train and test sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_padded, y\_encoded, test\_size=0.2, random\_state=42)  
  
# Build LSTM model  
vocab\_size = len(tokenizer.word\_index) + 1  
embedding\_dim = 300  
num\_classes = len(label\_encoder.classes\_)  
  
# Create a Bidirectional LSTM  
model = Sequential()  
model.add(Embedding(input\_dim=vocab\_size, output\_dim=embedding\_dim))  
model.add(Bidirectional(LSTM(units=128, return\_sequences=True)))  
model.add(Dropout(0.1))  
model.add(Bidirectional(LSTM(units=64)))  
model.add(Dropout(0.1))  
model.add(Dense(num\_classes, activation='sigmoid'))  
model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])  
  
# Train the model  
model.fit(X\_train, y\_train, epochs=10, batch\_size=128, validation\_split=0.1)  
  
# Evaluate the model  
loss, accuracy = model.evaluate(X\_test, y\_test)  
print(f"Loss: {loss}, Accuracy: {accuracy}")  
  
y\_pred = model.predict(X\_test).argmax(axis=1)  
precision = precision\_score(y\_test.argmax(axis=1), y\_pred, average='macro')  
recall = recall\_score(y\_test.argmax(axis=1), y\_pred, average='macro')  
f1 = f1\_score(y\_test.argmax(axis=1), y\_pred, average='macro')  
conf\_matrix = confusion\_matrix(y\_test.argmax(axis=1), y\_pred)  
  
print(f"Precision: {precision}, Recall: {recall}, F1-score: {f1}")  
print("Confusion Matrix:")  
print(conf\_matrix)

### **gnn\_smiles\_dummy.py**

import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from torch\_geometric.graphgym import optim  
from torch\_geometric.nn import GCNConv, global\_mean\_pool  
from torch\_geometric.data import Data  
import random  
  
  
# Class for the 2D-GCN  
class GCNClassifier(nn.Module):  
 def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):  
 super(GCNClassifier, self).\_\_init\_\_()  
 self.conv1 = GCNConv(input\_dim, hidden\_dim)  
 self.conv2 = GCNConv(hidden\_dim, hidden\_dim)  
 self.fc = nn.Linear(hidden\_dim, output\_dim)  
  
 def forward(self, data):  
 x, edge\_index, batch = data.x, data.edge\_index, data.batch  
  
 # Apply convolutions and ReLu  
 x = self.conv1(x, edge\_index)  
 x = F.relu(x)  
 x = self.conv2(x, edge\_index)  
 x = F.relu(x)  
  
 # Pool the node level embeddings learned from the hidden layers  
 x = global\_mean\_pool(x, batch)  
 x = self.fc(x)  
  
 return F.log\_softmax(x, dim=1)  
  
  
# Initialise some dummy data  
graphs\_data = [  
 Data(x=torch.randn(4, 5), edge\_index=torch.tensor([[0, 1, 2, 0], [1, 0, 2, 2]]), y=torch.tensor([0])),  
 Data(x=torch.randn(3, 5), edge\_index=torch.tensor([[0, 1, 2], [1, 0, 2]]), y=torch.tensor([1])),  
 Data(x=torch.randn(5, 5), edge\_index=torch.tensor([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]]), y=torch.tensor([2])),  
 Data(x=torch.randn(6, 5), edge\_index=torch.tensor([[0, 1, 2, 3, 4, 5], [1, 0, 2, 3, 4, 5]]), y=torch.tensor([0])),  
 Data(x=torch.randn(4, 5), edge\_index=torch.tensor([[0, 1, 2, 3], [1, 0, 2, 3]]), y=torch.tensor([0])),  
 Data(x=torch.randn(5, 5), edge\_index=torch.tensor([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]]), y=torch.tensor([1])),  
 Data(x=torch.randn(2, 5), edge\_index=torch.tensor([[0, 1, ], [1, 0, ]]), y=torch.tensor([2])),  
 Data(x=torch.randn(7, 5), edge\_index=torch.tensor([[0, 1, 2, 3, 4, 5, 6], [0, 1, 2, 3, 4, 5, 6]]), y=torch.tensor([0])),  
 Data(x=torch.randn(5, 5), edge\_index=torch.tensor([[0, 1, 2, 3, 4], [1, 2, 3, 4, 0]]), y=torch.tensor([1]))  
]  
  
# Split data into train and test sets  
random.seed(42)  
random.shuffle(graphs\_data)  
split\_index = int(0.8 \* len(graphs\_data))  
train\_data = graphs\_data[:split\_index]  
test\_data = graphs\_data[split\_index:]  
  
# Define the dimensions  
input\_dim = 5 # Dimension of node features  
hidden\_dim = 16 # Hidden dimension  
output\_dim = 3 # Number of classes (labels)  
  
# Create an instance of the model  
model = GCNClassifier(input\_dim, hidden\_dim, output\_dim)  
  
# Define the loss function  
criterion = nn.CrossEntropyLoss()  
# Define the optimizer  
optimizer = optim.Adam(model.parameters())  
  
# Training loop  
num\_epochs = 10  
for epoch in range(num\_epochs):  
 total\_loss = 0  
 for graph\_data in train\_data:  
 # Forward pass  
 output = model(graph\_data)  
  
 # Calculate the loss  
 loss = criterion(output, graph\_data.y)  
 total\_loss += loss.item()  
  
 # Backpropagation  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 # Print average loss  
 print(f"Epoch [{epoch + 1}/{num\_epochs}], Average Loss: {total\_loss / len(train\_data)}")  
  
# Evaluation loop  
with torch.no\_grad():  
 model.eval()  
 correct = 0  
 total = 0  
 for graph\_data in test\_data:  
 output = model(graph\_data)  
 \_, predicted = torch.max(output, 1)  
 total += graph\_data.y.size(0)  
 correct += (predicted == graph\_data.y).sum().item()  
 accuracy = correct / total  
 print(f"Accuracy: {accuracy}")

### **gnn\_smiles.py**

import random  
import pandas as pd  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from rdkit import Chem  
from sklearn.metrics import accuracy\_score, classification\_report  
from torch\_geometric.graphgym import optim  
from torch\_geometric.nn import GCNConv, global\_mean\_pool  
from torch\_geometric.data import Data  
  
  
# Class for the 2D-GCN  
class GCNGraphClassifier(nn.Module):  
 def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):  
 super(GCNGraphClassifier, self).\_\_init\_\_()  
 self.conv1 = GCNConv(input\_dim, hidden\_dim)  
 self.conv2 = GCNConv(hidden\_dim, hidden\_dim)  
 self.conv3 = GCNConv(hidden\_dim, hidden\_dim)  
 self.fc = nn.Linear(hidden\_dim, output\_dim)  
  
 def forward(self, data):  
 x, edge\_index, batch = data.x, data.edge\_index, data.batch  
  
 # Apply convolutions and ReLu  
 x = self.conv1(x, edge\_index)  
 x = F.relu(x)  
 x = self.conv2(x, edge\_index)  
 x = F.relu(x)  
 x = self.conv3(x, edge\_index)  
 x = F.relu(x)  
  
 # Pool the node level embeddings learned from the hidden layers  
 x = global\_mean\_pool(x, batch)  
 x = self.fc(x)  
  
 return F.log\_softmax(x, dim=1)  
  
  
# Function to convert SMILES to feature and adjacency matrix's  
def smiles\_to\_graph(smiles):  
 # Get the molecule from the SMILES string  
 mol = Chem.MolFromSmiles(smiles)  
 if mol is not None:  
 # Convert molecule to adjacency matrix  
 adj\_matrix = torch.tensor(Chem.GetAdjacencyMatrix(mol))  
 features = []  
 # Get the features for each atom  
 for atom in mol.GetAtoms():  
 atom\_features = []  
 # Get Atomic number  
 atom\_features.append(atom.GetAtomicNum())  
 # Get Formal charge  
 atom\_features.append(atom.GetFormalCharge())  
 # Get Atomic mass  
 atom\_features.append(atom.GetMass())  
  
 features.append(atom\_features)  
  
 features = torch.tensor(features, dtype=torch.float)  
  
 return features, adj\_matrix  
 else:  
 return None, None  
  
  
# Read the dataset  
df = pd.read\_csv("ChChSe-Decagon\_polypharmacy/filteredData.csv")  
  
# Extract features and target variable  
smiles1 = df['C1 SMILES']  
smiles2 = df['C2 SMILES']  
side\_effects = df["Side Effect Name"]  
  
# Map side effects to class indices  
unique\_side\_effects = side\_effects.unique()  
side\_effect\_to\_idx = {side\_effect: idx for idx, side\_effect in enumerate(unique\_side\_effects)}  
  
# Create list to store graph data  
graphs\_data = []  
  
# Create Data objects for each drug pair  
for idx, (smiles1, smiles2, side\_effect) in enumerate(zip(smiles1, smiles2, side\_effects)):  
 # Convert SMILES to graphs  
 features1, adj\_matrix1 = smiles\_to\_graph(smiles1)  
 features2, adj\_matrix2 = smiles\_to\_graph(smiles2)  
  
 if features1 is not None and features2 is not None:  
 # Pad the adjacency matrix's to match the sizes  
 max\_num\_atoms = max(adj\_matrix1.size(0), adj\_matrix2.size(0))  
 pad1 = max\_num\_atoms - adj\_matrix1.size(0)  
 pad2 = max\_num\_atoms - adj\_matrix2.size(0)  
 adj\_matrix1 = torch.nn.functional.pad(adj\_matrix1, (0, pad1, 0, pad1))  
 adj\_matrix2 = torch.nn.functional.pad(adj\_matrix2, (0, pad2, 0, pad2))  
  
 # Calculate edge indexes  
 edge\_index1 = torch.nonzero(adj\_matrix1, as\_tuple=False).t().contiguous()  
 edge\_index2 = torch.nonzero(adj\_matrix2, as\_tuple=False).t().contiguous()  
 # Offset edge indexes for the second graph  
 edge\_index2 += features1.size(0)  
  
 # Concatenate features and edge indexes  
 concatenated\_features = torch.cat((features1, features2), dim=0)  
 concatenated\_edge\_index = torch.cat([edge\_index1, edge\_index2], dim=1)  
  
 # Make the side effect into a tensor label  
 label = torch.tensor([side\_effect\_to\_idx[side\_effect]])  
  
 # Create PyTorch Geometric Data object  
 data = Data(x=concatenated\_features,  
 edge\_index=concatenated\_edge\_index,  
 y=label)  
  
 # Append to graphs\_data list  
 graphs\_data.append(data)  
  
# Split data into train and test sets  
random.seed(42)  
random.shuffle(graphs\_data)  
split\_index = int(0.8 \* len(graphs\_data))  
train\_data = graphs\_data[:split\_index]  
test\_data = graphs\_data[split\_index:]  
  
# Define the dimensions  
input\_dim = 3 # Dimension of node features  
hidden\_dim = 32 # Hidden dimension  
output\_dim = len(unique\_side\_effects) # Number of classes (labels)  
  
# Create an instance of the model  
model = GCNGraphClassifier(input\_dim, hidden\_dim, output\_dim)  
# Define the loss function  
criterion = nn.CrossEntropyLoss()  
# Define the optimizer  
optimizer = optim.Adam(model.parameters())  
  
# Training loop  
num\_epochs = 10  
for epoch in range(num\_epochs):  
 total\_loss = 0  
   
 # Set the model to training mode  
 model.train()  
  
 for graph\_data in train\_data:  
 # Forward pass  
 output = model(graph\_data)  
  
 # Calculate the loss  
 loss = criterion(output, graph\_data.y)  
 total\_loss += loss.item()  
  
 # Backpropagation  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 # Print average loss  
 print(f"Epoch [{epoch + 1}/{num\_epochs}], Training Loss: {total\_loss / len(train\_data)}")  
  
# Evaluation loop  
with torch.no\_grad():  
 model.eval()  
 true\_labels = []  
 predicted\_labels = []  
  
 for graph\_data in test\_data:  
 output = model(graph\_data)  
 \_, predicted = torch.max(output, 1)  
 true\_labels.extend(graph\_data.y.tolist())  
 predicted\_labels.extend(predicted.tolist())  
  
 accuracy = accuracy\_score(true\_labels, predicted\_labels)  
 print(f"Accuracy: {accuracy}")  
  
 # Calculate precision, recall, and F1-score  
 report = classification\_report(true\_labels, predicted\_labels, zero\_division=0)  
 print(report)