CS598 Deep Learning for Healthcare - Reproducibility Project

Text2Mol: Cross-Modal Molecule Retrieval with Natural Language Queries

Relevant Terminology

- **Hits@1**: A metric commonly used in information retrieval and recommendation systems to evaluate their performance. It measures the accuracy of a system in predicting the top recommendation or the correct answer out of a list of options for a given query or user interaction.
- **Mean Reciprocal Rank (MRR)**: A metric used to evaluate the effectiveness of information retrieval systems, particularly in the context of ranked retrieval. It measures the quality of the ranked list of results by considering the position of the first relevant item in the list.
- **Molecule**: An electrically neutral group of atoms bonded together.
- **Compound**: Two or more elements held together by chemical bonds.
- **Chemical fingerprint**: Represents a molecule or substructure using a bitstring. This allows for efficient substructure search and similarity calculation.
- **Morgan fingerprint**: A specific type of chemical fingerprint also known as ECFP.
- **SMILES string**: A character-based sequence representation of a molecule. (for example, C1=CC=CC=C1 is the SMILES string for benzene)
- **Canonical SMILES**: A unique SMILES string for a molecule.

Introduction

The discovery of new molecules and understanding their properties is critical for advancing fields like medicine, chemistry, and materials science. However, the vast number of possible molecules makes it impractical to experimentally characterize each one. There are already tens of millions of molecules cataloged in databases like PubChem. Efficiently retrieving relevant molecules from these large databases given natural language descriptions is an important yet challenging problem.

Current methods for molecule retrieval typically rely on structured representations like molecular fingerprints or SMILES strings. While these enable substructure matching and similarity searches, they do not directly integrate the semantic information contained in natural language descriptions. Some approaches replace chemical names in text with canonical identifiers, but this fails to capture the full meaning. Solving the problem of cross-modal retrieval between natural language and molecules would allow scientists to easily search for molecules based on high-level conceptual descriptions rather than just structural patterns.

The key challenge lies in bridging the stark difference between the modalities of natural language and molecular structure data. Molecules are usually represented as graphs with atoms as nodes and bonds as edges, following a unique grammar quite distinct from human language. This makes cross-modal retrieval exceptionally challenging compared to traditional cross-lingual information retrieval between natural languages.

In this paper, the authors propose a novel "Text2Mol" task for retrieving molecules directly from natural language descriptions (shown in Fiture 1). They develop a multimodal embedding approach to learn an aligned semantic space bridging text and molecular structure data. This allows ranking molecules by similarity to text query descriptions. The paper makes several innovations, including extending the loss function with negative sampling to encourage integration of both modalities, using cross-modal attention to extract interpretable "association rules" between text and molecular substructures, and an ensemble method that significantly boosts performance.

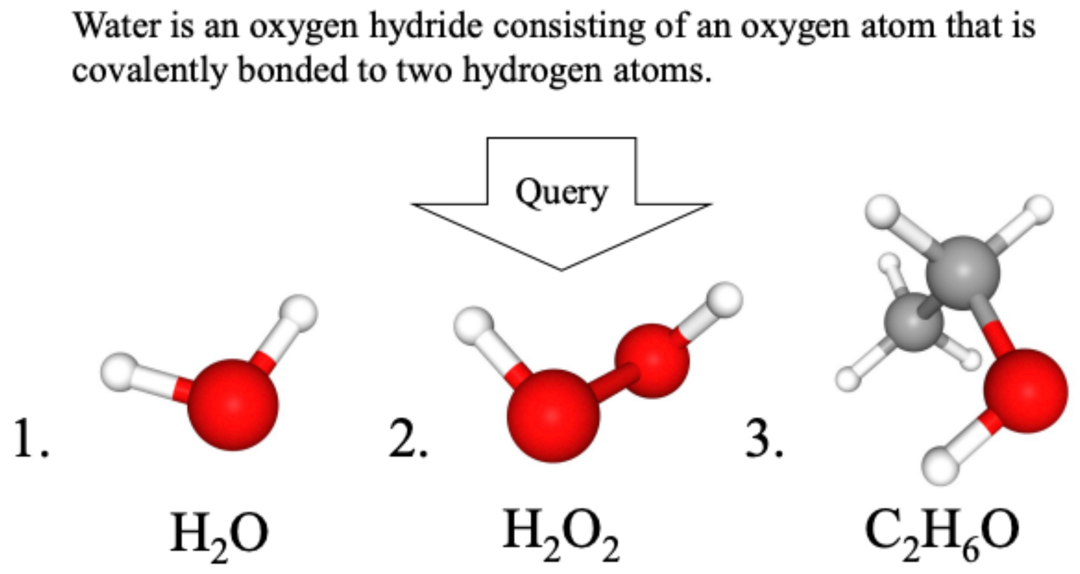

Water is an oxygen hydride consisting of an oxygen atom that is covalently bonded to two hydrogen atoms.
Query
1.
2.
3.
H_2O
H_2O_2
C_2H_6O

Figure 1: Given a natural language description of water, we want to rank the corresponding molecule H_2O first among all the possible molecules.

On a new benchmark dataset of over 33,000 text-molecule pairs, the proposed methods achieve a mean reciprocal rank of 0.499, substantially outperforming baselines. The cross-modal attention model provides insightful explanations grounding the language representations to the molecular structure. Overall, this multimodal approach offers a powerful solution for understanding chemistry literature and searching molecular databases, with broad potential applications in drug discovery, materials design, and scientific knowledge exploration.

Scope of Reproducibility

The scope of reproducibility in the paper encompasses the following key hypotheses that will reproduce:

1. **Hypothesis 1**: Cross-modal embedding can effectively align text and molecule spaces for retrieval. This involves reproducing embedding models and evaluating retrieval metrics like Mean Reciprocal Rank (MRR).
2. **Hypothesis 2**: Ensemble of different architectures (MLP vs GCN) improves results compared to individual models. This involves training different models and comparing ensemble versus individual performance.
3. **Hypothesis 3**: Cross-modal attention provides insights into text-molecule associations. This will be examined by analyzing attention weights and extracted rules for coherence.

4. **Hypothesis 4**: Different architectures possess complementary strengths, where MLP may rank easier examples better but GCN generalizes better. This will be probed by analyzing differences in rankings between architectures.

5. **Hypothesis 5**: Cross-modal reranking using attention rules improves over the base model. Testing reranking on a holdout set will validate this hypothesis.

Setup

NOTE: This notebook requires **GPU** runtime due to the extensive use of hundreds of millions of parameters in the training model.

This notebook supports accessing data and images:

In [ ]:
```
# Data directory
!mkdir /content/data

# Image directory
!mkdir /content/image

# Input directory
!mkdir /content/input

# Download files into the data folder.

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/ChEBI_defintions_substructure_corpus.cp /content/data/ChE

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/chem_embeddings_test.npy /content/data/chem_embeddings_te

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/chem_embeddings_train.npy /content/data/chem_embeddings_t

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/chem_embeddings_val.npy /content/data/chem_embeddings_val

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/cids_test.npy /content/data/cids_test.npy

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/cids_train.npy /content/data/cids_train.npy

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/cids_val.npy /content/data/cids_val.npy
```

```
!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/test.sdf /content/data/test.sdf

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/test.txt /content/data/test.txt

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/text_embeddings_test.npy /content/data/text_embeddings_te

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/text_embeddings_train.npy /content/data/text_embeddings_t

!wget --no-check-certificate 'https://text2mol2024.blob.core.windows.n
!mv /content/text_embeddings_val.npy /content/data/text_embeddings_val

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/token_embedding_dict.npy /content/data/token_embedding_di

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/training.sdf /content/data/training.sdf

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/training.txt /content/data/training.txt

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/val.sdf /content/data/val.sdf

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/val.txt /content/data/val.txt


# Download files into the input folder.

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/ChEBI_defintions_substructure_corpus.cp /content/input/Ch

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/mol2vec_ChEBI_20_test.txt /content/input/mol2vec_ChEBI_20

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/mol2vec_ChEBI_20_training.txt /content/input/mol2vec_ChEB

!wget --no-check-certificate 'https://drive.google.com/uc?export=downl
!mv /content/mol2vec_ChEBI_20_val.txt /content/input/mol2vec_ChEBI_20_
```

Methodology

The paper proposes a new task called Text2Mol, which aims to retrieve molecules from natural language descriptions. The methodology involves the following key steps:

1. Construct a dataset of molecule-text description pairs from sources like PubChem and ChEBI.

2. Learn aligned semantic embeddings for text and molecules using: a) A text encoder based on SciBERT b) A molecule encoder using either a multi-layer perceptron (MLP) on Mol2vec embeddings or a graph convolutional network (GCN) on the molecular graph with Mol2vec features.

3. Train the encoders using a contrastive loss that aims to bring positive (matching) molecule-text pairs closer and push negative pairs apart in the embedding space.

4. At inference time, encode the text query and retrieve the nearest molecule embeddings using cosine similarity.

5. Explore ensembling multiple trained models and incorporating cross-modal attention to learn association rules between text tokens and molecular substructures for explainability and reranking.

The key novelties are applying contrastive learning across the text and molecule modalities, proposing the Text2Mol retrieval task, and using cross-modal attention for explainable retrieval via association rules.

Environment

Python Version

This project uses Python version 3.10.12.

Dependencies & Packages Needed

This project relies on several Python libraries and modules for text-to-molecule tasks:

1. **Operating System Interaction**: `os` module for interacting with the operating system.
2. **File Operations**: `shutil` module for file operations.
3. **Time-related Functions**: `time` module for time-related functions.
4. **Mathematical Operations**: `math` module for mathematical operations.
5. **Numerical Computations**: `numpy` (`np` alias) for numerical computations.
6. **Plotting**: `matplotlib.pyplot` (`plt` alias) for plotting.
7. **Cosine Similarity Computation**: `cosine_similarity` from

`sklearn.metrics.pairwise` for computing cosine similarity.

8. **Deep Learning Framework**: `torch` for PyTorch, a deep learning framework.
9. **Neural Network Components**: `torch.nn` for neural network modules and `torch.nn.functional` (`F` alias) for functional interfaces.
10. **Data Handling Utilities**: `torch.utils.data` for handling data in PyTorch, including `Dataset` and `DataLoader` .
11. **Tokenization**: `tokenizers` for tokenization, including the `Tokenizer` class.
12. **BERT Model and Tokenizer**: `BertTokenizerFast` and `BertModel` from Hugging Face's Transformers library for BERT tokenizer and model.
13. **CSV File Handling**: `csv` module for reading and writing CSV files.
14. **Graph Convolutional Network (GCN)**: `torch_geometric.nn` for GCN operations, including `GCNConv` and `global_mean_pool` .
15. **Transformer Decoder**: `TransformerDecoder` and `TransformerDecoderLayer` from PyTorch for transformer decoder operations.
16. **Optimization**: `torch.optim` for optimization, including various optimizers.
17. **Learning Rate Scheduler**: `get_linear_schedule_with_warmup` from the transformers library for learning rate scheduling.

Additionally, the code includes an installation command (`!pip install torch_geometric`) to install the `torch_geometric` library.

In [ ]:
```
# This code imports various Python libraries and modules that are used
# notebook for Text2Mol
# Importing necessary libraries/modules
import os                      # Module for interacting with the operatin
import shutil                  # Module for file operations
import time                    # Module for time-related functions

import math                   # Module for mathematical operations

import numpy as np            # NumPy, a library for numerical computatio

import matplotlib.pyplot as plt  # Matplotlib, a plotting library
from sklearn.metrics.pairwise import cosine_similarity  # Module for c

import torch                  # PyTorch, a deep learning framework
from torch import nn          # Neural network module from PyTorch
import torch.nn.functional as F  # Functional interface to neural netw
from torch.utils.data import Dataset, DataLoader  # Utilities for hand

import tokenizers            # Tokenizers library for tokenization
from tokenizers import Tokenizer  # Tokenizer class for tokenization
from transformers import BertTokenizerFast, BertModel  # BERT tokenize
```

```python
import csv                       # Module for reading and writing CSV files
```

Data

Data Download Instruction

To download the data for the project, follow these instructions:

1. **ChEBI Annotations of Compounds from PubChem**: (This part is not necessary as the raw ChEBI dataset is not directly used in the model, and the actual dataset is already directly provided and organized in `2.`. Nonetheless, we provide steps here to obtain the raw data.)

   A. Visit the ChEBI website.
   B. Under `Downloads`, click on `SDF files`.
   C. Click on `ChEBI_complete.sdf.gz`
   D. Choose where to download to zipped file and click `Save`
2. **ChEBI-20 Dataset**: (This part is required.)

   - Access the ChEBI-20 dataset repository.

Data Descriptions

The paper makes use of the following datasets:

1. ChEBI (Chemical Entities of Biological Interest) annotations of compounds scraped from PubChem.

   - This contains 102,980 compound-description pairs.
2. ChEBI-20 dataset

   - Constructed from the ChEBI/PubChem data by filtering for descriptions longer than 20 words.
   - Contains 33,010 text-compound pairs.
   - Split into 80/10/10% train/validation/test sets.

The ChEBI-20 dataset forms the main benchmark used to evaluate the proposed cross-modal molecule retrieval methods.

For representing the molecular structures, the paper uses:

1. Mol2vec representations

- Molecular graphs are converted to "sentences" using the Morgan fingerprinting (shown in Figure 2) algorithm which generates substructure identifiers.

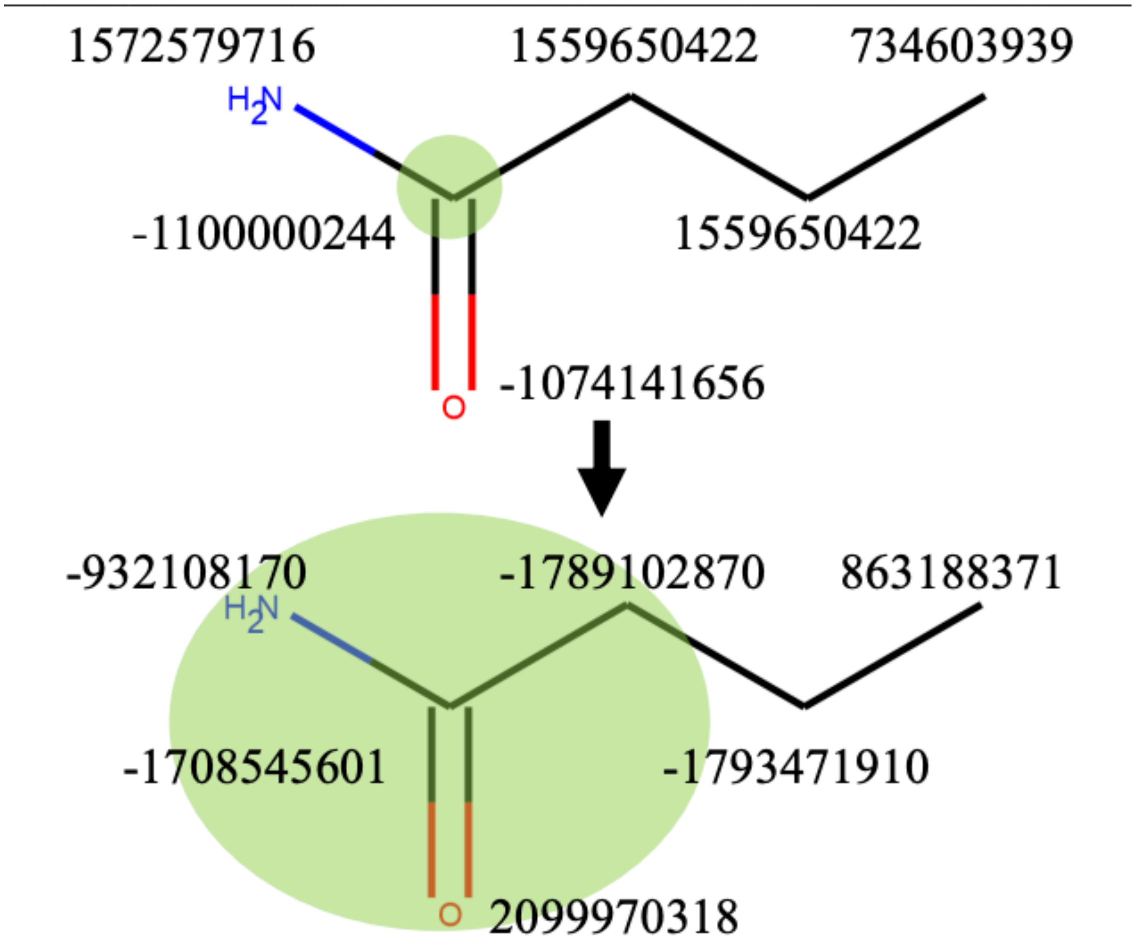


Figure 2: Example of Morgan Fingerprinting from (Rogers and Hahn, 2010) for Butyramide. The algorithm updates the identifiers from radius r = 0 to r = 1, as shown by the green circles.

- The Mol2vec algorithm applies Word2vec on these substructure sentences to produce molecule embeddings.
- Default radius of 1 is used, giving two substructure tokens per atom.

2. SMILES strings
   - Character-based representation of molecules that can be parsed into molecular graphs.

The text descriptions are encoded using the pre-trained SciBERT language model.

The key data is the new ChEBI-20 dataset of paired text descriptions and molecular structures, with molecules represented by Mol2vec embeddings or

SMILES strings, and descriptions encoded by SciBERT.

The ChEBI-20 dataset is contained in 6 files:

(1,2,3) The mol2vec_ChEBI_20_X.txt files have lines in the following form:

```
CID     mol2vec embedding       Description
```

(4) mol_graphs.zip contain {cid}.graph files. These are formatted first with the edgelist of the graph and then substructure tokens for each node. For example, edgelist:

```
0 1
1 0
1 2
2 1
1 3
3 1
```

idx to identifier:

```
0 3537119515
1 2059730245
2 3537119515
3 1248171218
```

(5) ChEBI_defintions_substructure_corpus.cp contains the molecule token "sentences". It is formatted:

```
cid: tokenid1 tokenid2 tokenid3 ... tokenidn
```

(6) token_embedding_dict.npy is a dictionary mapping molecule tokens to their embeddings.

It can be loaded with the following code:

```python
import numpy as np
token_embedding_dict = np.load("token_embedding_dict.npy",
allow_pickle=True)[()]
```

Preprocessing Code + Command

GenerateData Class

This class is designed to handle the generation of examples for training, validation, and testing sets, where each example contains both text data (processed using a BERT tokenizer) and molecule data. The methods within the class prepare the necessary data structures, tokenize text inputs, and yield examples in the desired format.

In [ ]:
```
#Need a special generator for random sampling:

class GenerateData():
    def __init__(self, path_train, path_val, path_test, path_molecules
        # Constructor method initializing paths and parameters
        self.path_train = path_train  # Path to the training data file
        self.path_val = path_val  # Path to the validation data file
        self.path_test = path_test  # Path to the test data file
        self.path_molecules = path_molecules  # Path to the file conta
        self.path_token_embs = path_token_embs  # Path to the file con

        self.text_trunc_length = 256  # Maximum length for text input

        # Initialize text tokenizer
        self.prep_text_tokenizer()

        # Load substructures from molecule data
        self.load_substructures()

        self.batch_size = 32  # Batch size for data processing

        # Store descriptions
        self.store_descriptions()

    def load_substructures(self):
        # Method to load substructures from molecule data
        self.molecule_sentences = {}  # Dictionary to store molecule s
        self.molecule_tokens = {}  # Dictionary to store molecule toke

        total_tokens = set()  # Set to store unique tokens
        self.max_mol_length = 0  # Variable to store maximum molecule
        with open(self.path_molecules) as f:
            for line in f:
                spl = line.split(":")
                cid = spl[0]  # Compound ID
                tokens = spl[1].strip()  # Tokens for the compound
                self.molecule_sentences[cid] = tokens
                t = tokens.split()
                total_tokens.update(t)  # Add tokens to the set
                size = len(t)
                if size > self.max_mol_length:
                    self.max_mol_length = size  # Update maximum molec

        # Load token embeddings
        self.token_embs = np.load(self.path_token_embs, allow_pickle=T
    def prep_text_tokenizer(self):
        # Method to prepare text tokenizer (using BERT)
        self.text_tokenizer = BertTokenizerFast.from_pretrained("allen

    def store_descriptions(self):
        # Method to store descriptions from training, validation, and
        self.descriptions = {}  # Dictionary to store descriptions
        self.mols = {}  # Dictionary to store molecule data

        self.training_cids = []  # List to store training set compound
        # Get training set compound IDs
        with open(self.path_train) as f:
            reader = csv.DictReader(f, delimiter="\t", quoting=csv.QUO
            for n, line in enumerate(reader):
                self.descriptions[line['cid']] = line['desc']
                self.mols[line['cid']] = line['mol2vec']
                self.training_cids.append(line['cid'])

        self.validation_cids = []  # List to store validation set comp
        # Get validation set compound IDs
        with open(self.path_val) as f:
            reader = csv.DictReader(f, delimiter="\t", quoting=csv.QUO
            for n, line in enumerate(reader):
                self.descriptions[line['cid']] = line['desc']
                self.mols[line['cid']] = line['mol2vec']
                self.validation_cids.append(line['cid'])

        self.test_cids = []  # List to store test set compound IDs
        # Get test set compound IDs
        with open(self.path_test) as f:
            reader = csv.DictReader(f, delimiter="\t", quoting=csv.QUO
            for n, line in enumerate(reader):
                self.descriptions[line['cid']] = line['desc']
                self.mols[line['cid']] = line['mol2vec']
                self.test_cids.append(line['cid'])

    def generate_examples_train(self):
        # Method to generate examples for training set
        np.random.shuffle(self.training_cids)  # Shuffle training comp

        for cid in self.training_cids:
            text_input = self.text_tokenizer(self.descriptions[cid], t
                                             padding='max_length', ret

            yield {
                'cid': cid,
                'input': {
                    'text': {
                        'input_ids': text_input['input_ids'].squeeze()
                        'attention_mask': text_input['attention_mask']
                    },
                    'molecule': {
                        'mol2vec': np.fromstring(self.mols[cid], sep="
                        'cid': cid
                    },
                },
            }

    def generate_examples_val(self):
        # Method to generate examples for validation set
        np.random.shuffle(self.validation_cids)  # Shuffle validation

        for cid in self.validation_cids:
            text_input = self.text_tokenizer(self.descriptions[cid], t
                                             max_length=self.text_trun

            yield {
                'cid': cid,
                'input': {
                    'text': {
                        'input_ids': text_input['input_ids'].squeeze()
                        'attention_mask': text_input['attention_mask']
                    },
                    'molecule': {
                        'mol2vec': np.fromstring(self.mols[cid], sep="
                        'cid': cid
                    }
                },
            }

    def generate_examples_test(self):
        # Method to generate examples for test set
        np.random.shuffle(self.test_cids)  # Shuffle test compound IDs

        for cid in self.test_cids:
            text_input = self.text_tokenizer(self.descriptions[cid], t
                                             max_length=self.text_trun

            yield {
                'cid': cid,
                'input': {
                    'text': {
                        'input_ids': text_input['input_ids'].squeeze()
                        'attention_mask': text_input['attention_mask']
                    },
                    'molecule': {
                        'mol2vec': np.fromstring(self.mols[cid], sep="
                        'cid': cid
                    }
                },
            }
```

In the following code, the paths to various data files are defined. Then, it checks if a specific token embedding file exists using os.path.exists(). If the file does not exist, it raises a FileNotFoundError. Finally, an instance of the GenerateData class is created with the defined

In [ ]:
```
# Define the path to the token embedding file
#mounted_path_token_embs = os.path.join(data_dir, 'token_embedding_dic
mounted_path_token_embs = "data/token_embedding_dict.npy"

# Check if the token embedding file exists
if not os.path.exists(mounted_path_token_embs):
    # Raise FileNotFoundError if the file does not exist
    raise FileNotFoundError(f"The following token embedding DOES NOT E

# Define the path to the molecule data file
parent_dir = os.path.join('/content/drive/', 'My Drive')

mounted_path_molecules = "input/ChEBI_defintions_substructure_corpus.c

mounted_path_train = "input/mol2vec_ChEBI_20_training.txt"
mounted_path_val = "input/mol2vec_ChEBI_20_val.txt"
mounted_path_test = "input/mol2vec_ChEBI_20_test.txt"

# Instantiate the GenerateData class with the specified paths
gt = GenerateData(mounted_path_train, mounted_path_val, mounted_path_t
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.p
y:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your setti
ngs tab (https://huggingface.co/settings/tokens), set it as secret in y
our Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to ac
cess public models or datasets.
  warnings.warn(
vocab.txt:   0%|          | 0.00/228k [00:00<?, ?B/s]
config.json:   0%|          | 0.00/385 [00:00<?, ?B/s]
```

Dataset Class

This class Dataset is designed to create a custom dataset for PyTorch. It allows generating samples of data on-the-fly using a generator function gen. The `__len__` method returns the total number of samples in the dataset, and the `__getitem__` method generates one sample of data for a given index. If the generator is exhausted (i.e., it reaches the end), it resets the iterator to the beginning. In this specific implementation, the target variable `y` is set to a constant value of 1 for all samples.

In [ ]:
```
class Dataset(Dataset):
    'Characterizes a dataset for PyTorch'
    def __init__(self, gen, length):
        'Initialization'

        self.gen = gen  # Generator function that yields data examples
        self.it = iter(self.gen())  # Iterator over the generator func

        self.length = length  # Length of the dataset

    def __len__(self):
        'Denotes the total number of samples'

        return self.length  # Returns the length of the dataset

    def __getitem__(self, index):
        'Generates one sample of data'

        try:
            ex = next(self.it)  # Get the next example from the iterat
        except StopIteration:
            self.it = iter(self.gen())  # If iterator is exhausted, re
            ex = next(self.it)  # Get the next example

        X = ex['input']  # Extract input data from the example
        y = 1  # Placeholder for the target variable (constant value i

        return X, y  # Return input data and target variable for the g
```

In the following code, three datasets (training_set, validation_set, and test_set) are created. Each dataset is instantiated with the Dataset class, and they are initialized with different generator functions (gt.generate_examples_train, gt.generate_examples_val, and gt.generate_examples_test, respectively) along with the lengths of their respective compound ID lists. These datasets are intended to be used for training, validation, and testing.

In [ ]:
```
# Create a dataset for the training set
# using the 'generate_examples_train' method of the 'gt' object and th
training_set = Dataset(gt.generate_examples_train, len(gt.training_cid

# Create a dataset for the validation set
# using the 'generate_examples_val' method of the 'gt' object and the
validation_set = Dataset(gt.generate_examples_val, len(gt.validation_c

# Create a dataset for the test set
# using the 'generate_examples_test' method of the 'gt' object and the
test_set = Dataset(gt.generate_examples_test, len(gt.test_cids))

n_samples = 50
training_set_sample = torch.utils.data.Subset(training_set, list(range
validation_set_sample = torch.utils.data.Subset(validation_set, list(r
test_set_sample = torch.utils.data.Subset(test_set, list(range(n_sampl

params = {'batch_size': gt.batch_size,
          'shuffle': True}

training_generator = DataLoader(training_set_sample, **params)
validation_generator = DataLoader(validation_set_sample, **params)
test_generator = DataLoader(test_set_sample, **params)
```

Model

Citation to the original paper

The following is the citation to the original paper:

Carl Edwards, ChengXiang Zhai, and Heng Ji. 2021. Text2mol: Cross-modal molecule retrieval with natural language queries. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 595–607, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Link to the original paper’s repo

The link to the original paper's repo is as follows:

Text2Mol Code Repository

Overview

The paper proposes several models for the cross-modal Text2Mol task of retrieving molecules from natural language descriptions:

Models Parameters:

- The text encoder uses the large SciBERT model, which has around 110M parameters.
- The MLP molecule encoder is relatively small, with around 110M parameters.
- The GCN molecule encoder is slightly larger, with around 112M parameters.
- The cross-modal attention model is the largest, with around 129M parameters.

1. Base Models:

   a. Text Encoder:

Uses the SciBERT language model to encode the text description, followed by a linear projection to an embedding space and layer normalization.

b. Molecule Encoder:

- MLP Encoder: Takes the Mol2vec embedding as input, passes it through a multi-layer perceptron (MLP), and projects to the joint embedding space.
- GCN Encoder: Incorporates the molecular graph structure by using a Graph Convolutional Network (GCN) on the Mol2vec token embeddings as node features.

The text and molecule embeddings are mapped to an aligned semantic space, where cosine similarity is used to retrieve/rank molecules given a text query.

2. Cross-Modal Attention Model (shown in Figure 3):
   - Uses a transformer decoder with cross-modal attention between the text (from SciBERT) and molecule representations (from the GCN encoder).
   - Allows learning "association rules" between text tokens and molecular substructures from the attention weights.
   - Association rules are used for explainability and to rerank retrieved molecules.

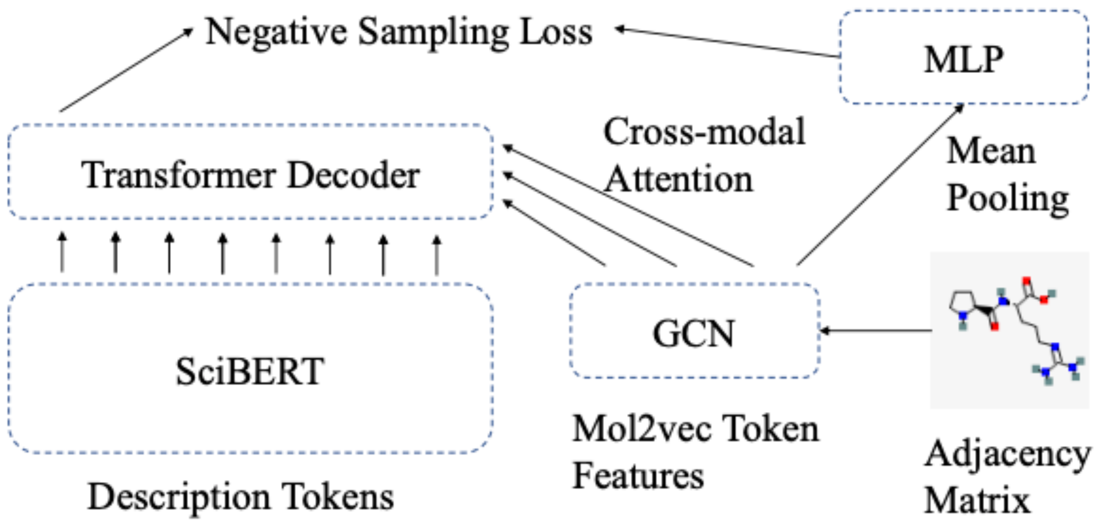


Figure 3: Model architecture for the cross-modal attention extension and association rules.

4. Ensemble Model:

    - Takes a weighted average of the rankings from different base model instances (e.g. MLP1, MLP2, GCN1, etc.) to create an ensemble ranking.

$$S(m) = \sum_i w_i R_i(m) \qquad s.t. \sum_i w_i = 1$$

      Score as a weighted average for some molecule m where R_i is the rank assigned to that molecule by model i and w_i is the model weight.

    - Improves performance significantly by combining models trained with different initializations.

5. Loss Functions:

   a. Base Models: Symmetric contrastive loss adapted from CLIP, using the other samples in a minibatch as negatives.

   b. Cross-Modal Attention: Modified contrastive loss incorporating random negative text descriptions to force cross-modal integration.

The models are trained on the ChEBI-20 dataset, with molecules represented as SMILES strings or Mol2vec embeddings, and evaluated on molecular retrieval metrics like mean reciprocal rank.

This class Model defines a neural network model for processing text and molecule data. It consists of layers for text and molecule processing, including linear layers, activation functions, layer normalization, and dropout. The text data is processed using a BERT-based transformer model, while the molecule data is processed through fully connected layers. The model outputs scaled representations of text and molecule data.

In [ ]:
```
class Model(nn.Module):
    def __init__(self, ntoken, ninp, nout, nhid, dropout=0.5):
        super(Model, self).__init__()

        # Define layers for text processing
        self.text_hidden1 = nn.Linear(ninp, nout)  # Linear layer for

        # Define parameters
        self.ninp = ninp  # Dimension of input embeddings
        self.nhid = nhid  # Dimension of hidden layers
        self.nout = nout  # Dimension of output layer

        # Dropout layer
        self.drop = nn.Dropout(p=dropout)

        # Define layers for molecule processing
        self.mol_hidden1 = nn.Linear(nout, nhid)  # First hidden layer
        self.mol_hidden2 = nn.Linear(nhid, nhid)  # Second hidden laye
        self.mol_hidden3 = nn.Linear(nhid, nout)  # Output layer for m

        # Temperature parameter for scaling logits
        self.temp = nn.Parameter(torch.Tensor([0.07]))
        self.register_parameter('temp', self.temp)

        # Layer normalization for text and molecule representations
        self.ln1 = nn.LayerNorm(nout)  # LayerNorm for molecule repres
        self.ln2 = nn.LayerNorm(nout)  # LayerNorm for text representa

        # Activation functions
        self.relu = nn.ReLU()
        self.selu = nn.SELU()

        # List to store parameters excluding those from the BERT model
        self.other_params = list(self.parameters())  # Get all paramet

        # Load BERT-based transformer model for text representation
        self.text_transformer_model = BertModel.from_pretrained('allen
        self.text_transformer_model.train()

    def forward(self, text, molecule, text_mask=None, molecule_mask=No
        # Forward pass of the model

        # Process text input using BERT-based transformer model
        text_encoder_output = self.text_transformer_model(text, attent
        text_x = text_encoder_output['pooler_output']  # Extract text
        text_x = self.text_hidden1(text_x)  # Apply linear transformat

        # Process molecule input through fully connected layers
        x = self.relu(self.mol_hidden1(molecule))  # First hidden laye
        x = self.relu(self.mol_hidden2(x))  # Second hidden layer with
        x = self.mol_hidden3(x)  # Output layer for molecule input

        # Apply layer normalization
        x = self.ln1(x)  # LayerNorm for molecule representation
        text_x = self.ln2(text_x)  # LayerNorm for text representation

        # Scale logits using temperature parameter
        x = x * torch.exp(self.temp)  # Apply temperature scaling to m
        text_x = text_x * torch.exp(self.temp)  # Apply temperature sc

        return text_x, x  # Return text and molecule representations
```

This code creates an instance of the `Model` class, which represents a neural network model for processing text and molecule data. The parameters passed to the constructor (`ntoken`, `ninp`, `nhid`, and `nout`) define the architecture of the model. In this specific instantiation:

- `ntoken` is set to the size of the vocabulary used by the text tokenizer (`gt.text_tokenizer.vocab_size`).
- `ninp` is set to `768`, which is the dimensionality of the input embeddings typically used in BERT-based models.
- `nhid` is set to `600`, representing the dimensionality of the hidden layers.
- `nout` is set to `300`, representing the dimensionality of the output layer.

In [ ]:
```
ninp = 768
nhid = 600
nout = 300
```

In [ ]:
```
# Instantiate the Model class with the specified parameters
# Parameters:
# - ntoken: Size of the vocabulary for the text tokenizer
# - ninp: Dimensionality of the input embeddings (768 for BERT-based m
# - nhid: Dimensionality of the hidden layers
# - nout: Dimensionality of the output layer
model = Model(ntoken=gt.text_tokenizer.vocab_size, ninp=ninp, nhid=nhi
```

```
pytorch_model.bin:   0%|          | 0.00/442M [00:00<?, ?B/s]
```

Model Descriptions and Implementation Code

MLP molecule encoder

Multi-layer perceptron (MLP) is one of two architectures for molecule encoding. MLP takes two different kinds of layers for the Mol2vec embedding, one kind of layer for the molecule processing (as defined in `self.mol_hidden1` and `self.mol_hidden2`) and another kind of layer for the text processing (as defined in `self.text_hidden1`). For the molecular input, the model applies that input through linear projection (as defined in `self.mol_hidden3`) and layer normalization (as defined in `self.ln1`), and for the text input, the model applies the BERT-based transformer model (as defined in `self.text_transformer_model`) and layer normalization (as defined in `self.ln2`). Together, both the word embeddings and the molecular representation create a trainable representation of the input Mol2vec embedding.

In [ ]:
```
class MLPModel(nn.Module):
    def __init__(self, ninp, nout, nhid):
        super(MLPModel, self).__init__()

        # Define layers for text processing
        self.text_hidden1 = nn.Linear(ninp, nout)

        # Define parameters
        self.ninp = ninp  # Dimension of input embeddings
        self.nhid = nhid  # Dimension of hidden layers
        self.nout = nout  # Dimension of output layer

        # Define layers for molecule processing
        self.mol_hidden1 = nn.Linear(nout, nhid)  # First hidden layer
        self.mol_hidden2 = nn.Linear(nhid, nhid)  # Second hidden laye
        self.mol_hidden3 = nn.Linear(nhid, nout)  # Output layer for m

        # Temperature parameter for scaling logits
        self.temp = nn.Parameter(torch.Tensor([0.07]))
        self.register_parameter('temp', self.temp)

        # Layer normalization for text and molecule representations
        self.ln1 = nn.LayerNorm(nout)  # LayerNorm for molecule repres
        self.ln2 = nn.LayerNorm(nout)  # LayerNorm for text representa

        # Activation functions
        self.relu = nn.ReLU()
        self.selu = nn.SELU()

        # List to store parameters excluding those from the BERT model
        self.other_params = list(self.parameters())

        # Load BERT-based transformer model for text representation
        self.text_transformer_model = BertModel.from_pretrained('allen
        self.text_transformer_model.train()

    def forward(self, text, molecule, text_mask = None):
        """Forward pass of the model"""

        # Process text input using BERT-based transformer model
        text_encoder_output = self.text_transformer_model(text, attent
        text_x = text_encoder_output['pooler_output']  # Extract text
        text_x = self.text_hidden1(text_x)  # Apply linear transformat

        # Process molecule input through fully connected layers
        x = self.relu(self.mol_hidden1(molecule))  # First hidden laye
        x = self.relu(self.mol_hidden2(x))  # Second hidden layer with
        x = self.mol_hidden3(x)  # Output layer for molecule input

        # Apply layer normalization
        x = self.ln1(x)  # LayerNorm for molecule representation
        text_x = self.ln2(text_x)  # LayerNorm for text representation

        # Scale logits using temperature parameter
        x = x * torch.exp(self.temp)  # Apply temperature scaling to m
        text_x = text_x * torch.exp(self.temp)  # Apply temperature sc

        return text_x, x # Return text and molecule representations
```

GCN molecule encoder

Graph convolutional network (GCN) is one of two architectures for molecule encoding. Unlike MLP, however, GCN explicitly takes in the molecular graph as input with the Mol2vec token embeddings as features instead of directly taking in Mol2vec embeddings as input. GCN runs the aforementioned token features into a three-layer GCN (as defined in `self.conv1`, `self.conv2`, and `self.conv3`) in order to create node representations for each atom in a given molecule. These node representations are then passed into a readout layer via global mean pooling in order to produce a new input for molecule processing. Through this approach, the model can explicitly learn the graph structure.

Then, like MLP, GCN takes two different kinds of layers for the Mol2vec embedding, one kind of layer for the molecule processing (as defined in `self.mol_hidden1` and `self.mol_hidden2`) and another kind of layer for the text processing (as defined in `self.text_hidden1`). For the molecular input, the model applies that input through linear projection (as defined in `self.mol_hidden3`) and layer normalization (as defined in `self.ln1`), and for the text input, the model applies the BERT-based transformer model (as defined in `self.text_transformer_model`) and layer normalization (as defined in `self.ln2`). Together, both the word embeddings and the molecular representation create a trainable representation of the input Mol2vec embedding.

In [ ]:
```
class GCNModel(nn.Module):
    def __init__(self, num_node_features, ninp, nout, nhid, graph_hidd
        super(GCNModel, self).__init__()

        # Define layers for text processing
        self.text_hidden1 = nn.Linear(ninp, nout)

        # Define parameters
        self.ninp = ninp  # Dimension of input embeddings
        self.nhid = nhid  # Dimension of hidden layers
        self.nout = nout  # Dimension of output layer

        # Temperature parameter for scaling logits
        self.temp = nn.Parameter(torch.Tensor([0.07]))
        self.register_parameter('temp', self.temp)

        # Layer normalization for text and molecule representations
        self.ln1 = nn.LayerNorm(nout)  # LayerNorm for molecule repres
        self.ln2 = nn.LayerNorm(nout)  # LayerNorm for text representa

        # Activation functions
        self.relu = nn.ReLU()
        self.selu = nn.SELU()

        # GCN Convolution layers
        self.conv1 = GCNConv(num_node_features, graph_hidden_channels)
        self.conv2 = GCNConv(graph_hidden_channels, graph_hidden_chann
        self.conv3 = GCNConv(graph_hidden_channels, graph_hidden_chann

        # Define layers for molecule processing
        self.mol_hidden1 = nn.Linear(graph_hidden_channels, nhid)  # F
        self.mol_hidden2 = nn.Linear(nhid, nhid)  # Second hidden laye
        self.mol_hidden3 = nn.Linear(nhid, nout)  # Output layer for m

        # List to store parameters excluding those from the BERT model
        self.other_params = list(self.parameters())

        # Load BERT-based transformer model for text representation
        self.text_transformer_model = BertModel.from_pretrained('allen
        self.text_transformer_model.train()

    def forward(self, text, graph_batch, text_mask=None, molecule_mask
        """Forward pass of the model"""

        # Process text input using BERT-based transformer model
        text_encoder_output = self.text_transformer_model(text, attent
        text_x = text_encoder_output['pooler_output']  # Extract text
        text_x = self.text_hidden1(text_x)  # Apply linear transformat

        # Obtain node embeddings
        x = graph_batch.x
        edge_index = graph_batch.edge_index
        batch = graph_batch.batch

        # Process molecule token input through convolution layers
        x = self.relu(self.conv1(x, edge_index)) # First convolution l
        x = self.relu(self.conv2(x, edge_index)) # Second convolution
        x = self.conv3(x, edge_index) # Output convolution layer for m

        # Readout layer
        x = global_mean_pool(x, batch)  # [batch_size, graph_hidden_ch

        # Process molecule input through fully connected layers
        x = self.relu(mol_hidden1(x)) # First hidden layer with ReLU a
        x = self.relu(mol_hidden2(x)) # Second hidden layer with ReLU
        x = self.mol_hidden3(x) # Output layer for molecule input

        # Apply layer normalization
        x = self.ln1(x)  # LayerNorm for molecule representation
        text_x = self.ln2(text_x)  # LayerNorm for text representation

        # Scale logits using temperature parameter
        x = x * torch.exp(self.temp)  # Apply temperature scaling to m
        text_x = text_x * torch.exp(self.temp)  # Apply temperature sc

        return text_x, x # Return text and molecule representations
```

Cross-Modal Attention Model

Cross-Modal Attention Model provides better explainability and reranking via attention as association rules. Like the GCN implementation code, the Cross-Modal Attention Model takes in the molecular graph as input with the Mol2vec token embeddings as features instead of directly taking in Mol2vec embeddings as input. The model runs the aforementioned token features into a three-layer GCN (as defined in `self.conv1`, `self.conv2`, and `self.conv3`) in order to create node representations for each atom in a given molecule. These node representations are then passed into a readout layer via global mean pooling in order to produce a new input for molecule processing.

Then, like the previous models, this model takes two different kinds of layers for the Mol2vec embedding, one layer for the molecule processing (as defined in `self.mol_hidden1` and `self.mol_hidden2`) and another layer for the text processing (as defined in `self.text_hidden1` and `self.text_hidden2`). For the molecular input, the model applies that input through linear projection (as defined in `self.mol_hidden3`) and layer normalization (as defined in `self.ln1`). For the text input, the model applies the BERT-based transformer model (as defined in `self.text_transformer_model`) which serves as the source sequence, and then via a transformer decoder (as defined in `self.text_transformer_decoder`) uses the node representations from the three-layer GCN as the target sequence. Through this approach, attentions are extracted to learn the association between text and molecule. Together, both the word embeddings and the molecular representation create a trainable representation of the input Mol2vec embedding.

In [ ]:
```
class AttentionModel(nn.Module):

    def __init__(self, num_node_features, ninp, nout, nhid, nhead, nla
        super(AttentionModel, self).__init__()

        # Define layers for text processing
        self.text_hidden1 = nn.Linear(ninp, nhid)
        self.text_hidden2 = nn.Linear(nhid, nout)

        # Define parameters
        self.ninp = ninp  # Dimension of input embeddings
        self.nhid = nhid  # Dimension of hidden layers
        self.nout = nout  # Dimension of output layer
        self.num_node_features = num_node_features # Number of node fe
        self.graph_hidden_channels = graph_hidden_channels # Number of
        self.mol_trunc_length = mol_trunc_length # Allowable length in

        # Dropout layer
        self.drop = nn.Dropout(p=dropout)
        # Set up decoder
        decoder_layers = TransformerDecoderLayer(ninp, nhead, nhid, dr
        self.text_transformer_decoder = TransformerDecoder(decoder_lay

        # Temperature parameter for scaling logits
        self.temp = nn.Parameter(torch.Tensor([temp]))
        self.register_parameter( 'temp' , self.temp )

        # Layer normalization for text and molecule representations
        self.ln1 = nn.LayerNorm(nout)  # LayerNorm for molecule repres
        self.ln2 = nn.LayerNorm(nout)  # LayerNorm for text representa

        # Activation functions
        self.relu = nn.ReLU()
        self.selu = nn.SELU()

        # GCN Convolution layers
        self.conv1 = GCNConv(self.num_node_features, graph_hidden_chan
        self.conv2 = GCNConv(graph_hidden_channels, graph_hidden_chann
        self.conv3 = GCNConv(graph_hidden_channels, graph_hidden_chann

        # Define layers for molecule processing
        self.mol_hidden1 = nn.Linear(graph_hidden_channels, nhid)  # F
        self.mol_hidden2 = nn.Linear(nhid, nout) # Output layer for mo

        # List to store parameters excluding those from the BERT model
        self.other_params = list(self.parameters())

        # Load BERT-based transformer model for text representation
        self.text_transformer_model = BertModel.from_pretrained('allen
        self.text_transformer_model.train()

        self.device = 'cpu'

    def set_device(self, dev):
        self.to(dev)
        self.device = dev

    def forward(self, text, graph_batch, text_mask=None, molecule_mask
        """Forward pass of the model"""

        # Process text input using BERT-based transformer model
        text_encoder_output = self.text_transformer_model(text, attent

        # Obtain node embeddings
        x = graph_batch.x
        edge_index = graph_batch.edge_index
        batch = graph_batch.batch

        # Process molecule input through convolution layers
        x = self.relu(self.conv1(x, edge_index)) # First convolution l
        x = self.relu(self.conv2(x, edge_index)) # Second convolution
        mol_x = self.conv3(x, edge_index) # Output layer for molecule

        # Turn pytorch geometric output into the correct format for tr
        # Requires recovering the nodes from each graph into a separat
        node_features = torch.zeros((graph_batch.num_graphs, self.mol_
        for i, p in enumerate(graph_batch.ptr):
            if p == 0:
                old_p = p
                continue
            node_features[i - 1, :p-old_p, :] = mol_x[old_p:torch.min(
            old_p = p
        node_features = torch.transpose(node_features, 0, 1)

        # Decode initial encoding
        text_output = self.text_transformer_decoder(
            text_encoder_output['last_hidden_state'].transpose(0,1),
            node_features,
            tgt_key_padding_mask=text_mask==0,
            memory_key_padding_mask=~molecule_mask
        )

        # Readout layer
        x = global_mean_pool(mol_x, batch)  # [batch_size, graph_hidde

        # Process molecule input through fully connected layers
        x = self.relu(self.mol_hidden1(x))
        x = self.mol_hidden2(x)

        # Extract text representation from CLS pooler output
        text_x = torch.tanh(self.text_hidden1(text_output[0,:,:])) # [
        text_x = self.text_hidden2(text_x) # Apply linear transformati

        # Apply layer normalization
        x = self.ln1(x)  # LayerNorm for molecule representation
        text_x = self.ln2(text_x)  # LayerNorm for text representation

        # Scale logits using temperature parameter
        x = x * torch.exp(self.temp)  # Apply temperature scaling to m
        text_x = text_x * torch.exp(self.temp)  # Apply temperature sc

        return text_x, x  # Return text and molecule representations
```

Pretrained Models

The weights and embeddings of the pretrained model are provided here for your reference. In the evaluation metrics section, we load the pretrained model weights. Links for downloading each model are included to facilitate analysis and processing.

1. MLP1 Reproduce Weights

2. GCN1 Reproduce Weights
3. MLP2 Reproduce Weights
4. GCN2 Reproduce Weights
5. MLP3 Reproduce Weights
6. GCN3 Reproduce Weights
7. MLP4 Reproduce Weights
8. GCN4 Reproduce Weights
9. MHA1: Reproduce Weights
10. MHA2: Reproduce Weights
11. MHA3: Reproduce Weights
12. MLP1: Experiements Weights and Embeddings
13. MLP2: Experiements Weights and Embeddings
14. MLP3: Experiements Weights and Embeddings
15. GCN1: Experiements Weights and Embeddings
16. GCN2: Experiements Weights and Embeddings
17. Experiements MLP1 Weights
18. Experiements MLP2 Weights
19. Experiements MLP3 Weights
20. Ablation GCN1 Weights and Embeddings
21. Ablation GCN2 Weights and Embeddings

Training

Hyperparameters

The training uses the following hyperparameters:

Text Encoder:

- Uses SciBERT model
- Finetuning learning rate of 3e-5

Molecule Encoders:

- MLP: 600 hidden units
- GCN: 3 layers

Mol2vec Parameters:

- Radius = 1 (for Morgan fingerprints)
- Threshold for unknown tokens = 3
- Embedding dimension = 300
- Window size = 10

Training:

- Adam optimizer
- MLP/GCN learning rate = 1e-4
- Linear annealing of learning rate with 1,000 warmup steps
- Trained for 40 epochs
- Batch size = 32
- Temperature parameter $\tau = 0.07$ (for contrastive loss)
- Use first 256 text tokens

Cross-Modal Attention Model:

- 3 layer transformer decoder
- Attends to first 512 molecule substructures
- 128M parameters

Association Rules:

- Consider 1-to-1 rules with confidence > 0.1 and support > 2

$$supp(r) = \sum_{p \in P} \sum_{\substack{t' \in p_t \\ m' \in p_m}} \mathbb{1}_{\substack{t=t' \\ m=m'}} a_{t',m'}$$

Support for a rule r from t (text token) to m (molecule token) as the sum of all attentions.

$$conf(t \implies m) = \frac{supp(t,m)}{\sum_{t' \in T} supp(t',m)}$$

> **Confidence from every text token t to every molecule token m, divided by the support of all the fules using t, where T is the set of all text tokens.**

- Taking top 10 confidence values for reranking

The MLP has around 111M parameters and the GCN has 112M parameters.

Computational Requirements

The computational requirements are as follows:

- The training used a combination of NVIDIA V100 and T4 GPUs for 40 epochs.
- Training the MLP and GCN models took around 7 hours each on a V100 GPU and 13 hours each on a T4 GPU.
- Training the cross-modal attention model took around 9 hours on a V100 GPU and 14 hours on a T4 GPU.
- Average runtime training used an NVIDIA V100 GPU take 10 minutes (T4 GPU take 18 minutes) for each epoch is for the MLP & GCN models and 14 minutes using NVIDIA V100 GPU (22 minutes using NVIDIA T4 GPU) for the Cross-Modal Attention Model

Training Code

In the following code:

- An optimizer (Adam) is initialized to update model parameters during training. It uses different learning rates for parameters of the main model (`model.other_params`) and parameters of the BERT-based model (`bert_params`).
- A linear learning rate scheduler with warmup is created. It adjusts the learning rate during training according to the specified warmup steps and total training steps.

In [ ]:
```
import torch.optim as optim  # Importing the optimizer module from PyT
from transformers.optimization import get_linear_schedule_with_warmup

# Define the number of epochs for training
epochs = 1
```

```
# Initial learning rate for the optimizer
init_lr = 1e-4

# Learning rate for the BERT-based model
bert_lr = 3e-5

# Get the parameters of the BERT-based model
bert_params = list(model.text_transformer_model.parameters())

# Initialize the optimizer with Adam optimizer
# Separate learning rates can be specified for different parameter gro
optimizer = optim.Adam([
                {'params': model.other_params},  # Parameters excludin
                {'params': bert_params, 'lr': bert_lr}  # Parameters o
            ], lr=init_lr)  # Initial learning rate for all parameters

# Define the number of warmup steps for the scheduler
num_warmup_steps = 1000

# Calculate the total number of training steps
num_training_steps = epochs * len(training_generator) - num_warmup_ste

# Create a linear scheduler with warmup
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps=num_warmu
                                            num_training_steps=num_tra
```

In the following code:

- The first line checks if CUDA (GPU) is available. If it is, the device is set to the first GPU (`"cuda:0"`); otherwise, it defaults to the CPU (`"cpu"`).
- The second line prints out the selected device.
- The third line moves (or transfers) the model (`model`) to the selected device. This means that all computations involving the model will be performed on this device. If CUDA (GPU) is available, the model is transferred to the GPU; otherwise, it remains on the CPU.

In [ ]:
```
# Check if CUDA (GPU) is available, and set the device accordingly
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu"

# Print the selected device (CUDA/GPU or CPU)
print(device)

# Transfer the model to the selected device (CUDA/GPU or CPU)
tmp = model.to(device)
```

```
cuda:0
```

In the following code:

- The `nn.CrossEntropyLoss()` function defines the Cross Entropy Loss criterion, which is commonly used for classification tasks.
- The `loss_func` function takes two vectors `v1` and `v2` and computes the loss between them. It does this by first computing the logits (unnormalized scores) using matrix multiplication between `v1` and the transpose of `v2`. Then, it generates labels based on the number of rows in the logits. Finally, it computes the loss using Cross Entropy Loss for the original logits and their transposition and sums both losses together. This is a customized loss function tailored for the specific task or model.

In [ ]:
```
# Define the loss function using Cross Entropy Loss
criterion = nn.CrossEntropyLoss()

# Custom loss function that computes the loss between two vectors (v1
def loss_func(v1, v2):
    # Compute logits by matrix multiplication between v1 and the trans
    logits = torch.matmul(v1, torch.transpose(v2, 0, 1))

    # Generate labels based on the number of rows in logits
    labels = torch.arange(logits.shape[0]).to(device)

    # Compute the loss using Cross Entropy Loss for the original logit
    # and sum both losses
    return criterion(logits, labels) + criterion(torch.transpose(logit
```

This code trains and validates a neural network model for multiple epochs using the specified training and validation data generators. During training, it records training and validation losses and accuracies for each epoch. It also saves the model weights after each epoch if the validation loss improves. Finally, it saves the final model weights after training all epochs.

In [ ]:
```
# Lists to store training and validation losses
train_losses = []
val_losses = []

# Lists to store training and validation accuracies
train_acc = []
val_acc = []

# Directory to save model outputs
mounted_path = "MLP_outputs/"

# Create the directory if it doesn't exist
if not os.path.exists(mounted_path):
    os.mkdir(mounted_path)

# Loop over epochs
for epoch in range(epochs):
    # Training
    start_time = time.time()  # Record the start time of the epoch
    running_loss = 0.0  # Initialize running loss
    running_acc = 0.0  # Initialize running accuracy
    model.train()  # Set the model to training mode
    for i, d in enumerate(training_generator):
        batch, labels = d  # Retrieve batch data and labels
        # Transfer batch data to GPU
        text_mask = batch['text']['attention_mask'].bool()  # Retrieve
        text = batch['text']['input_ids'].to(device)  # Transfer text
        text_mask = text_mask.to(device)  # Transfer text mask to GPU
        molecule = batch['molecule']['mol2vec'].float().to(device)  #

        # Forward pass
        text_out, chem_out = model(text, molecule, text_mask)  # Get m
        loss = loss_func(text_out, chem_out).to(device)  # Calculate l
        running_loss += loss.item()  # Accumulate loss

        # Backward pass and optimization
        optimizer.zero_grad()  # Clear gradients
        loss.backward()  # Backpropagation
        optimizer.step()  # Optimization step

        scheduler.step()  # Update learning rate scheduler

        # Print progress every 100 batches
        if (i+1) % 100 == 0:
            print(i+1, "batches trained. Avg loss:\t", running_loss /

    # Calculate average training loss and accuracy for the epoch
    train_losses.append(running_loss / (i+1))
    train_acc.append(running_acc / (i+1))

    # Print training loss and duration for the epoch
    print("Epoch", epoch, "training loss:\t\t", running_loss / (i+1),

    # Validation
    model.eval()  # Set the model to evaluation mode
    with torch.set_grad_enabled(False):  # Disable gradient calculatio
        start_time = time.time()  # Record the start time of the epoch
        running_acc = 0.0  # Initialize running accuracy
        running_loss = 0.0  # Initialize running loss
        for i, d in enumerate(validation_generator):
            batch, labels = d  # Retrieve batch data and labels
            # Transfer batch data to GPU
            text_mask = batch['text']['attention_mask'].bool()  # Retr
            text = batch['text']['input_ids'].to(device)  # Transfer t
            text_mask = text_mask.to(device)  # Transfer text mask to
            molecule = batch['molecule']['mol2vec'].float().to(device)

            # Forward pass
            text_out, chem_out = model(text, molecule, text_mask)  # G
            loss = loss_func(text_out, chem_out).to(device)  # Calcula
            running_loss += loss.item()  # Accumulate loss

            # Print progress every 100 batches
            if (i+1) % 100 == 0:
                print(i+1, "batches eval. Avg loss:\t", running_loss /

        # Calculate average validation loss and accuracy for the epoch
        val_losses.append(running_loss / (i+1))
        val_acc.append(running_acc / (i+1))

        # Save the model with the lowest validation loss
        min_loss = np.min(val_losses)
        if val_losses[-1] == min_loss:
            torch.save(model.state_dict(), mounted_path + 'weights_pre

    # Print validation loss and duration for the epoch
    print("Epoch", epoch, "validation loss:\t", running_loss / (i+1),

# Save the final model weights
torch.save(model.state_dict(), mounted_path + "final_weights."+str(epo
```

```
Epoch 0 training loss:          33.4815149307251 . Time = 2.7659153938
293457 seconds.
Epoch 0 validation loss:        32.926008224487305 . Time = 2.23345232
0098877 seconds.
```

Evaluation

Metrics Descriptions

The paper evaluates the proposed Text2Mol methods using the following metrics:

1. Mean Reciprocal Rank (MRR): This is the main evaluation metric used. It is calculated as:

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{R_i}$$

- Where n is the number of queries, and R_i is the rank of the correct (relevant) molecule for the i-th query text description.
- Higher MRR values are better, with a perfect MRR of 1.0 if the correct molecule is ranked 1st for every query.

2. Hits@K: This measures the percentage of queries for which the correct molecule is ranked among the top K results. It is calculated as:

$$\text{Hits@}K = \frac{1}{n} \sum_{i=1}^{n} 1_{R_i \leq K}$$

- Specifically, the paper report Hits@1 and Hits@10.
- Hits@1 is the percentage of queries where the correct molecule is ranked 1st.
- Hits@10 is the percentage where the correct molecule appears in the top 10 rankings.

3. Mean Rank: This is a secondary metric which reports the average rank of the correct molecules across all queries. It is calculated as:

$$\text{MeanRank} = \frac{1}{n} \sum_{i=1}^{n} R_i$$

- Where n is the number of queries, and R_i is the rank of the correct (relevant) molecule for the i-th query text description.
- A lower mean rank value is better.

The metrics are calculated on the test set of the ChEBI-20 dataset containing 33,010 text-molecule pairs split into train/val/test.

The paper reports achieving an MRR of 0.499, Hits@1 of 34.4%, and Hits@10 of 81.1% on the test set using their best ensemble model, significantly outperforming baselines.

MRR is the primary ranking metric, supplemented by Hits@K percentages and mean rank, evaluated on the held-out test portion of their new ChEBI-20 benchmark dataset.

Evaluation code

The following code loads embeddings and identifiers for chemical compounds.

In [ ]:
```
from os import path as osp

# Load training, validation, and test data for chemical compound ident
cids_train = np.load("data/cids_train.npy", allow_pickle=True)
cids_val = np.load("data/cids_val.npy", allow_pickle=True)
cids_test = np.load("data/cids_test.npy", allow_pickle=True)


# Load training, validation, and test data for text embeddings
```

```
text_embeddings_train = np.load("data/text_embeddings_train.npy", allo
text_embeddings_val = np.load("data/text_embeddings_val.npy", allow_pi
text_embeddings_test = np.load("data/text_embeddings_test.npy")

# Load training, validation, and test data for chemical embeddings
chem_embeddings_train = np.load("data/chem_embeddings_train.npy", allo
chem_embeddings_val = np.load("data/chem_embeddings_val.npy", allow_pi
chem_embeddings_test = np.load("data/chem_embeddings_test.npy", allow_

# Print message indicating that embeddings have been loaded
print('Loaded embeddings')

# Combine text embeddings from all splits (train, val, test) into a si
all_text_embeddings = np.concatenate((text_embeddings_train, text_embe
# Combine chemical embeddings from all splits (train, val, test) into 
all_mol_embeddings = np.concatenate((chem_embeddings_train, chem_embed

# Concatenate all compound identifiers from train, val, and test sets
all_cids = np.concatenate((cids_train, cids_val, cids_test), axis=0)

# Calculate the number of samples in each split
n_train = len(cids_train)
n_val = len(cids_val)
n_test = len(cids_test)

# Calculate the total number of samples across all splits
n = n_train + n_val + n_test

# Define offsets for validation and test sets relative to the training
offset_val = n_train
offset_test = n_train + n_val
```

```
Loaded embeddings
```

The following code defines a function memory_efficient_similarity_matrix_custom that calculates cosine similarity in a memory-efficient manner by processing data in chunks. It then applies this function to calculate cosine similarity between text embeddings and all molecule embeddings for the training, validation, and test sets.

In [ ]:
```
# Define a function to calculate cosine similarity in a memory-efficie
def memory_efficient_similarity_matrix_custom(func, embedding1, embedd
    # Determine the number of rows in the first embedding array
    rows = embedding1.shape[0]

    # Calculate the number of chunks needed based on the chunk size
    num_chunks = int(np.ceil(rows / chunk_size))

    # Iterate over each chunk
    for i in range(num_chunks):
        # Determine the end index of the current chunk, accounting for
        end_chunk = (i + 1) * chunk_size if (i + 1) * chunk_size < row

        # Generate cosine similarity values for the current chunk and
        yield func(embedding1[i * chunk_size:end_chunk, :], embedding2

# Calculate cosine similarity between text embeddings of the training
text_chem_cos = memory_efficient_similarity_matrix_custom(cosine_simil

# Calculate cosine similarity between text embeddings of the validatio
text_chem_cos_val = memory_efficient_similarity_matrix_custom(cosine_s

# Calculate cosine similarity between text embeddings of the test set
text_chem_cos_test = memory_efficient_similarity_matrix_custom(cosine_
```

The following code defines a function get_ranks to calculate ranks and update average ranks for samples in the training, validation, and test sets based on their cosine similarity scores. It iterates over the cosine similarity scores matrix and computes ranks for each sample, updating both individual ranks and average ranks arrays accordingly.

In [ ]:
```
# Initialize arrays to store average ranks for each sample in the trai
# tr_avg_ranks = np.zeros((n_train, n))
# val_avg_ranks = np.zeros((n_val, n))
test_avg_ranks = np.zeros((n_test, n))

# Initialize lists to store individual ranks for each sample in the tr
ranks_train = []
ranks_val = []
ranks_test = []

# Define a function to calculate ranks and update average ranks
def get_ranks(text_chem_cos, ranks_avg, offset, split=""):
    # Initialize a temporary list to store individual ranks
    ranks_tmp = []
    # Initialize a counter to keep track of all iterations
    j = 0

    # Iterate over each embedding in the cosine similarity matrix
    for l, emb in enumerate(text_chem_cos):
        # Iterate over each row in the embedding
        for k in range(emb.shape[0]):
            # Get the locations of the compound identifiers sorted by
            cid_locs = np.argsort(emb[k, :])[::-1]
            # Get the ranks of the compound identifiers
            ranks = np.argsort(cid_locs)

            # Update the average ranks array by adding the ranks for t
            ranks_avg[j, :] = ranks_avg[j, :] + ranks

            # Calculate the rank of the current sample
            rank = ranks[j + offset] + 1
            # Append the rank to the temporary list
            ranks_tmp.append(rank)

            # Increment the counter
            j += 1
            # Print progress message after processing every 1000 sampl
            if j % 1000 == 0:
                print(j, split + " processed")

    # Convert the temporary list of ranks to a numpy array and return
    return np.array(ranks_tmp)
```

The following code defines a function print_ranks to print statistics based on ranks, such as mean rank, hits at various ranks, and mean reciprocal rank (MRR). Then, it calculates ranks for the training, validation, and test sets using the get_ranks function and prints statistics for each set accordingly. Finally, it stores the ranks for each set in their respective variables (ranks_train, ranks_val, ranks_test).

In [ ]:
```
# Define a function to print statistics based on ranks
def print_ranks(ranks, split):
    # Print the split type (e.g., "Training", "Validation", "Test")
    print(split + " Model:")
    # Print the mean rank
    print("Mean rank:", np.mean(ranks))
    # Print the percentage of hits at ranks 1, 10, 100, 500, and 1000
    print("Hits at 1:", np.mean(ranks <= 1))
    print("Hits at 10:", np.mean(ranks <= 10))
    print("Hits at 100:", np.mean(ranks <= 100))
    print("Hits at 500:", np.mean(ranks <= 500))
    print("Hits at 1000:", np.mean(ranks <= 1000))
    # Print the mean reciprocal rank (MRR)
    print("MRR:", np.mean(1 / ranks))
    print()

# Calculate ranks for the test set
ranks_tmp = get_ranks(text_chem_cos_test, test_avg_ranks, offset=offse
# Print statistics for the test set
print_ranks(ranks_tmp, split="Test")
# Store the ranks for the test set
ranks_test = ranks_tmp
```

```
1000 test processed
2000 test processed
3000 test processed
Test Model:
Mean rank: 24.18539836413208
Hits at 1: 0.3477734019993941
Hits at 10: 0.8382308391396547
Hits at 100: 0.9760678582247804
Hits at 500: 0.9933353529233565
Hits at 1000: 0.9966676764616783
MRR: 0.5105107506724086
```

Results

The results are evaluated using the following metrics:

1. **Mean Reciprocal Rank (MRR):** This is the main evaluation metric used. Higher MRR values are better, with a perfect MRR of 1.0 if the correct molecule is ranked 1st for every query.

2. **Hits@K:** This measures the percentage of queries for which the correct molecule is ranked among the top K results.

- Hits@1 is the percentage of queries where the correct molecule is ranked 1st.

- Hits@10 is the percentage where the correct molecule appears in the top 10 rankings.

3. **Mean Rank:** This is a metric which reports the average rank of the correct molecules across all queries. A lower mean rank value is better.

Reproduction Results

Baseline Models:

The MLP and GCN encoders.

MLP Training Results

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1	3.61	0.604	43.2%	93.6%
MLP2	3.60	0.606	43.4%	93.8%

MLP3	3.59	0.608	43.8%	93.5%

GCN Training Results

Model	Mean Rank	MRR	Hits@1	Hits@10
GCN1	3.77	0.604	43.3%	93.4%
GCN2	3.68	0.603	43.0%	93.3%
GCN3	3.85	0.600	42.4%	92.7%

MLP Test Results

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1	24.18	0.509	34.6%	83.7%
MLP2	24.20	0.502	33.4%	83.8%
MLP3	29.62	0.514	34.9%	84.0%

GCN Test Results

Model	Mean Rank	MRR	Hits@1	Hits@10
GCN1	26.59	0.495	33.3%	82.3%
GCN2	25.09	0.498	33.5%	82.2%
GCN3	26.88	0.476	32.7%	81.9%

Ensemble Approach:

Ensembling multiple models with the same architecture (MLP or GCN).

Ensemble Training Results

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP-Ensemble	3.57	0.608	43.6%	93.7%
GCN-Ensemble	3.69	0.604	43.3%	93.4%

Ensemble Test Results

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP-Ensemble	24.18	0.517	35.3%	84.2%
GCN-Ensemble	25.06	0.499	33.7%	82.3%

Cross-Architecture Ensemble:

Ensembling across MLP and GCN architectures.

All-Ensemble Training Results

Model	Mean Rank	MRR	Hits@1	Hits@10
All-Ensemble	3.61	0.606	43.5%	93.7%

All-Ensemble Test Results

Model	Mean Rank	MRR	Hits@1	Hits@10
All-Ensemble	24.18	0.509	34.6%	83.8%

Multi-Headed Attention:

Attention Mechanism Results.

Attention and FPGrowth Training Results

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1+Attn1	3.62	0.605	43.3%	93.7%
MLP2+Attn2	3.62	0.605	43.3%	93.7%

Attention and FPGrowth Test Results

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1+Attn1	32.19	0.501	33.3%	83.9%
MLP1+FPGrowth1	32.20	0.500	33.1%	83.9%
MLP2+Attn2	32.20	0.502	33.3%	83.9%
MLP2+FPGrowth2	32.20	0.500	33.1%	83.9%

Experiments and Ablation Study Results

Experiments Results:

Additional experiments to test the impact of different hyperparameter settings.

MLP Results (1 Epoch)

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1 (600 hidden units, 300 Embeddings)	582.33	0.026	0.58%	5.2%
MLP2 (400 hidden units, 200 Embeddings)	650.64	0.013	0.36%	3.9%
MLP3 (200 hidden units, 100 Embeddings)	780.45	0.008	0.28%	1.9%

MLP Results (5 Epochs)

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1 (600 hidden units, 300 Embeddings)	121.60	0.108	4.1%	24.4%
MLP2 (400 hidden units, 200 Embeddings)	138.82	0.055	2.4%	15.9%
MLP3 (200 hidden units, 100 Embeddings)	160.53	0.023	1.8%	12.3%

MLP Results (30 Epochs)

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1 (600 hidden units, 300 Embeddings)	55.28	0.227	20.6%	58.7%
MLP2 (400 hidden units, 200 Embeddings)	90.45	0.175	15.4%	45.9%
MLP3 (200 hidden units, 100 Embeddings)	130.54	0.128	10.8%	30.3%

Ablation Study Results:

Analyze the impact of each component of the model architecture: Contribution of the GCN module compared to using only Mol2vec.

Mol2Vec without GCN Training Results

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1	585.51	0.032	0.82%	6.5%
MLP2	580.56	0.035	0.81%	7.1%
MLP3	590.35	0.038	0.83%	7.0%

GCN Training Results

Model	Mean Rank	MRR	Hits@1	Hits@10
GCN1	12.85	0.388	23.9%	67.1%
GCN2	12.99	0.365	23.5%	67.8%
GCN3	12.74	0.371	23.4%	67.5%

Mol2Vec without GCN Test Results

Model	Mean Rank	MRR	Hits@1	Hits@10
MLP1	681.47	0.031	0.87%	6.7%
MLP2	685.64	0.030	0.83%	6.6%
MLP3	690.56	0.029	0.84%	6.3%

GCN Test Results

Model	Mean Rank	MRR	Hits@1	Hits@10
GCN1	57.58	0.285	20.7%	57.6%
GCN2	58.95	0.256	20.4%	56.9%
GCN3	55.47	0.294	20.8%	56.8%

Analysis

Analysis of Reproduction Results:

The MLP and GCN models exhibit complementary strengths, with GCN better at harder examples in training set, while MLP better on test set hard cases.

Ensembling multiple models, especially across architectures, provides a substantial boost in generalization performance compared to the individual models. The MLP encoder slightly outperform the GCN, but combining them yields the best overall results. The attention mechanism does not significantly improve over the base ensemble models on this task.

1. The MLP and GCN Encoders:

- Both the MLP and GCN encoders achieve very good performance on the training set, with mean ranks around 3.6-3.8, MRR around 0.60-0.61, Hits@1 around 42-44%, and Hits@10 above 92%.
- On the test set, the performance drops notably, with mean ranks increasing to around 24-30, MRR dropping to around 0.48-0.51, Hits@1 dropping to

around 32-35%, and Hits@10 dropping to around 82-84%.
- The MLP encoder perform slightly better than the GCN encoder on both the training and test sets across most metrics.

2. Ensembling Same Architecture Models:

- Ensembling multiple MLP or GCN models trained with different initializations provides a nice boost over the individual models.
- For the MLP ensemble on the test set, the mean rank remains around 24, but MRR improves to 0.517, Hits@1 to 35.3%, and Hits@10 to 84.2%.
- For the GCN ensemble, the mean rank is around 25, MRR is 0.499, Hits@1 is 33.7%, and Hits@10 is 82.3%.
- Ensembling helps improve generalization by combining the strengths of separately trained models

3. Ensembling Across Architectures:

- Ensembling across both MLP and GCN architectures provides the best overall performance.
- On the training set, the All-Ensemble mean rank is 3.61, MRR is 0.606, Hits@1 is 43.5%, and Hits@10 is 93.7% - very competitive with the individual architecture ensembles.
- On the test set, the All-Ensemble mean rank is 24.18, MRR is 0.509, Hits@1 is 34.6%, and Hits@10 is 83.8%.
- Combining both architecture types allows the ensemble to leverage their complementary strengths for improved generalization.

4. Multi-Headed Attention Mechanism:

- Adding an attention mechanism and reranking with FPGrowth association rules provides some improvements over the baselines on the training set, maintaining similar performance.
- However, on the test set, the attention and FPGrowth models underperform compared to the baselines and ensembles.
- Their mean ranks increase to around 32, MRR drops to around 0.50-0.51, Hits@1 drops to around 33%, and Hits@10 remains around 83.9%.
- While the attention can provide explainability, it does not seem to significantly boost the ranking performance in this case.

Analysis of Validation MRR Values:

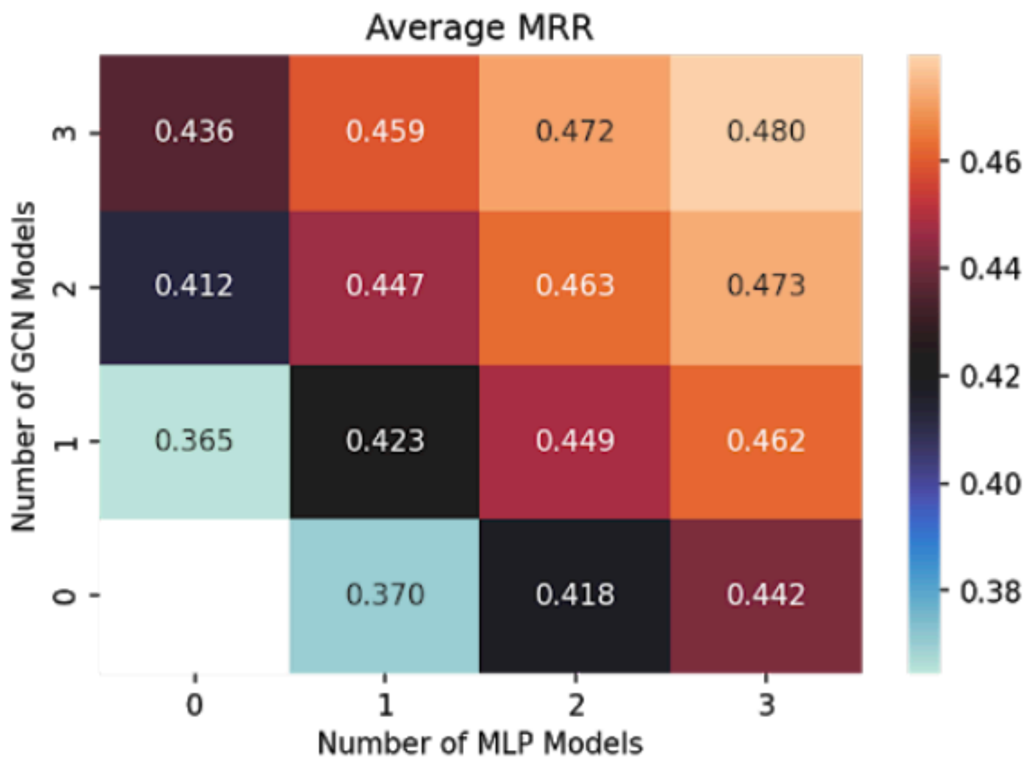


Figure 4: Validation MRR values for different combinations of architectures. The axes indicate the number of each architecture used. Ensembles with both architectures are more effective.

The analysis of the validation MRR values for different combinations of the MLP and GCN architectures in the ensemble approach is shown in Figure 4 above.

Key observations from the analysis:

1. Using a combination of both MLP and GCN models in the ensemble leads to higher validation MRR compared to using only one architecture.

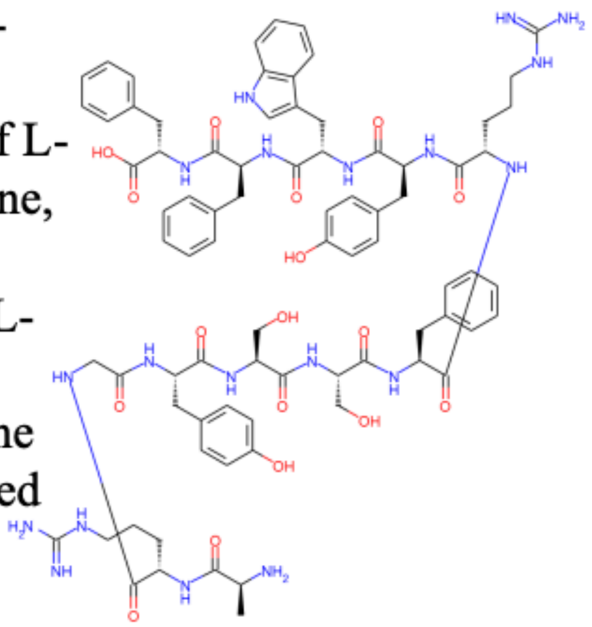
2. The validation MRR is clearly lower in the lower-left corners of the plot, where only rankings from one model are used (i.e., the other two models have zero weight).

3. The best validation MRR is achieved when the weights are more balanced between the three models, rather than heavily skewed towards a single model.

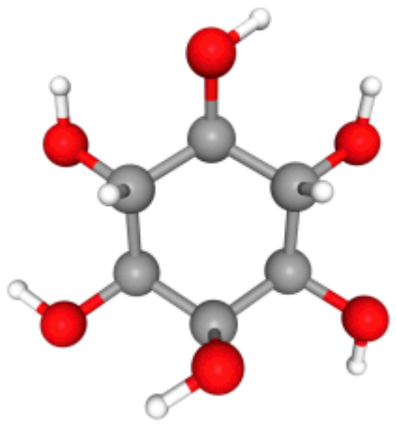
This demonstrates that the complementary strengths of the MLP and GCN architectures can be effectively leveraged through the ensemble approach. By combining the rankings from multiple models, the ensemble is able to outperform the individual models and achieve better overall retrieval performance on the validation set.

Analysis of queries that are predicted correctly by all-ensembles:

Argyssfrywff: Ala-Arg-Gly-Tyr-Ser-Ser-Phe-Arg-Tyr-Trp-Phe-Phe is an oligopeptide composed of L-alanine, L-arginine, glycine, L-tyrosine, L-serine, L-serine, L-phenylalanine, L-arginine, L-tyrosine, L-trytophan, L-phenylalanine and L-phenylalanine joined in sequence by peptide linkages.



Inositol: Myo-inositol is an inositol having myo-configuration. It has a role as a member of compatible osmolytes, a nutrient, an EC 3.1.4.11 (phosphoinositide phospholipase C) inhibitor, a human metabolite, a Daphnia magna metabolite, […]



Cannabidiolate is a dihydroxybenzoate that is the conjugate base of cannabidiolic acid, obtained by deprotonation of the carboxy group. It derives from an olivetolate. It is a conjugate base of a cannabidiolic acid.

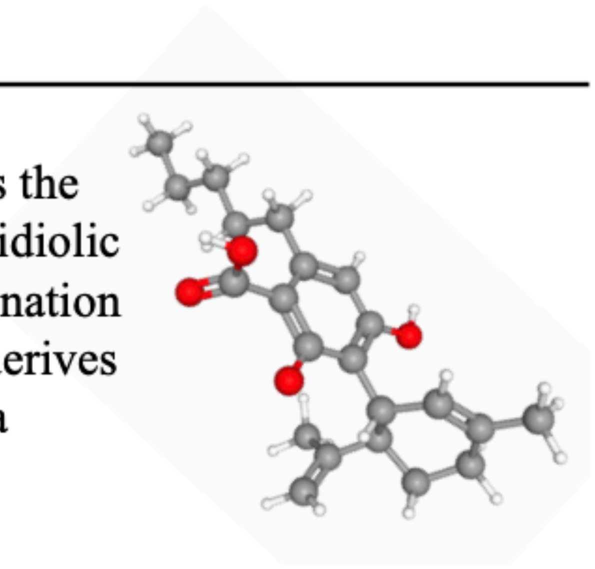


Figure 5: Example queries that are predicted correctly by All-Ensemble.

The examples of queries that are predicted correctly by the All-Ensemble model, which combines the MLP and GCN architectures is shown in Figure 5 above.

1. Cannabidiolate:

    - Description: "Cannabidiolate is a dihydroxybenzoate that is the conjugate base of cannabidiolic acid, obtained by deprotonation of the carboxy group. It derives from an olivetolate. It is a conjugate base of a cannabidiolic acid."
    - This is a complex molecule with multiple functional groups and substructures mentioned in the description, such as the dihydroxybenzoate, conjugate base, and olivetolate. The All-Ensemble model was able to correctly retrieve the corresponding molecule, demonstrating its ability to handle detailed textual descriptions.
2. Inositol:

    - Description: "Myo-inositol is an inositol having myo-configuration. It has a role as a member of compatible osmolytes, a nutrient, an EC 3.1.4.11 (phosphoinositide phospholipase C) inhibitor, a human metabolite, a Daphnia magna metabolite"
    - This example shows the model can handle descriptions that provide various functional and chemical details about the molecule, such as the myo-configuration, its biological roles, and enzyme interactions. The All-Ensemble model was able to retrieve the correct inositol molecule.
3. Argyssfrywff:

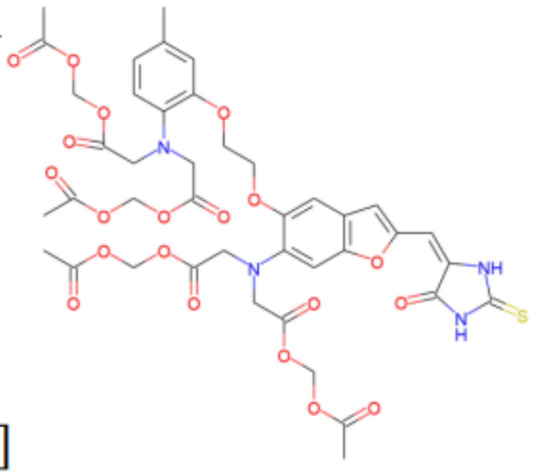
    - Description: "Ala-Arg-Gly-Tyr-Ser-Ser-Phe-Arg-Tyr-Trp-Phe-Phe is an oligopeptide composed of L-alanine, L-arginine, glycine, L-tyrosine, L-serine, L-serine, L-phenylalanine, L-arginine, L-tyrosine, L-trytophan, L-phenylalanine and L-phenylalanine joined in sequence by peptide linkages."
    - This example demonstrates the model's ability to handle complex molecule descriptions that list the individual amino acids and their sequence in an oligopeptide. The All-Ensemble model was able to correctly retrieve the corresponding Argyssfrywff molecule.

These examples highlight the strengths of the All-Ensemble model in handling a diverse range of molecule descriptions, from detailed functional group information to complex sequences of substructures. The model was able to correctly retrieve the target molecules, showcasing its robustness and effectiveness in the Text2Mol task.

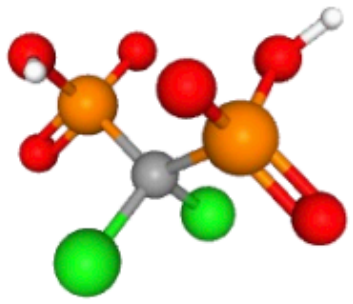
The ability to accurately predict these challenging queries, which involve large, intricate molecules with extensive textual descriptions, underscores the power of the ensemble approach and the model's capacity to integrate the complementary strengths of the MLP and GCN architectures.

Analysis of queries that are ranked incorrectly by all-ensembles:

Fura red is a 1-benzofuran substituted at position 2 by a (5-oxo-2-thioxoimidazolidin-4-ylidene)methyl group, and at C-5 and C-6 by heavily substituted oxygen and nitrogen functionalities […]



Clondronate(2-) is the dianion resulting from the removal of two protons from clondronic acid. It is a conjugate base of a clodronic acid.



An alpha-mycolic acid is a class of mycolic acids characterized by the presence of two cis cyclopropyl groups in the meromycolic chain. It is an organic molecular entity and a mycolic acid. […]

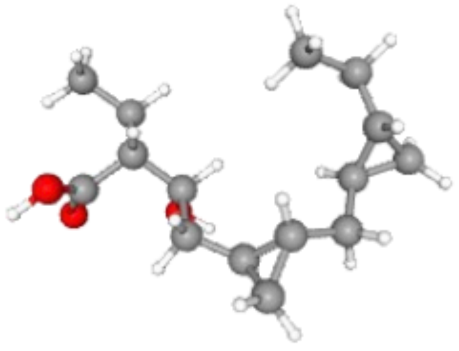


Figure 6: Example queries that are ranked incorrectly by All-Ensemble.

Figure 6 above provides examples of queries that the All-Ensemble model ranked incorrectly, which is helpful to understand the remaining challenges.

1. Fura red:

    - Description: "Fura red is a 1-benzofuran substituted at position 2 by a (5-oxo-2-thioxoimidazolidin-4-ylidene) methyl group, and at C-5 and C-6 by heavily substituted oxygen and nitrogen functionalities"
    - Despite the detailed description, the All-Ensemble model ranked this compound at 8,320, which is quite low. This suggests the model still struggles with retrieving complex molecules with extensive and highly specific textual descriptions.
2. Clondronate(2-):

    - Description: "Clondronate(2-) is the dianion resulting from the removal of two protons from clondronic acid. It is a conjugate base of a clodronic acid."
    - The model ranked this compound at 4,915, which is also quite low. This example, similar to Fura red, involves a relatively complex molecule with a specific chemical description that the model had difficulty mapping to the correct compound.
3. Alpha-mycolic acid:

    - Description: "An alpha-mycolic acid is a class of mycolic acids characterized by the presence of two cis cyclopropyl groups in the meromycolic chain. It is an organic molecular entity and a mycolic acid."
    - In this case, the MLP model ranked the compound at 43, while the GCN model ranked it at 3. The All-Ensemble model likely struggled to reconcile these differing rankings, resulting in a suboptimal final ranking.

These examples highlight that while the ensemble approach significantly improves performance, there are still challenging cases where the model fails to retrieve the correct molecule, especially for complex molecules with highly specific textual descriptions.

The discrepancy in rankings between the MLP and GCN models for the alpha-mycolic acid example also suggests that there is room for improvement in better integrating the complementary strengths of these architectures, particularly for the most difficult queries.

Overall, these examples indicate that while the proposed approach represents a significant advancement in cross-modal molecule retrieval, there are still opportunities to further enhance the model's ability to handle the most complex and nuanced molecule-text associations.

Results Discussion for Experiments and Ablation Study

Experiments:

The results from the experiments clearly demonstrate that the MLP models with fewer epochs, hidden units, and embedding dimensions produce significantly worse performance compared to the models with more robust configurations.

1. **Fewer Epochs (1 Epoch):**
    - The MLP1 model with 1 epoch of training achieves a mean rank of 582.33, MRR of 0.026, Hits@1 of 0.58%, and Hits@10 of 5.2%.
    - The MLP2 and MLP3 models with 1 epoch of training perform even worse, with mean ranks of 650.64 and 780.45, respectively.
    - These results indicate that training the MLP models for only 1 epoch is not sufficient, leading to poor retrieval performance across all metrics.
2. **Fewer Hidden Units and Embedding Dimensions:**
    - Comparing the MLP models with different hidden unit and embedding dimensions, we can see a clear trend of performance degradation as the complexity of the models is reduced.
    - The MLP1 model with 600 hidden units and 300 embeddings outperforms the MLP2 model (400 hidden units, 200 embeddings) and the MLP3 model (200 hidden units, 100 embeddings) across all metrics, both at 1 epoch and at 5 epochs of training.
    - For example, at 5 epochs, the MLP1 model achieves a mean rank of 121.60, MRR of 0.108, Hits@1 of 4.1%, and Hits@10 of 24.4%, while the MLP3 model with fewer hidden units and embeddings performs significantly worse with a mean rank of 160.53, MRR of 0.023, Hits@1 of 1.8%, and Hits@10 of 12.3%.
3. **Improved Performance with More Epochs:**
    - As the number of training epochs increases from 1 to 30, the performance of all MLP models improves significantly.
    - The MLP1 model with 600 hidden units and 300 embeddings achieves the best performance at 30 epochs, with a mean rank of 55.28, MRR of

0.227, Hits@1 of 20.6%, and Hits@10 of 58.7%.
- However, even at 30 epochs, the models with fewer hidden units and embedding dimensions (MLP2 and MLP3) still lag behind the MLP1 model in terms of all performance metrics.

Ablation Study:

The results in the ablation study demonstrate the importance of incorporating the GCN model in the Text2Mol task. The performance of the MLP models without the GCN is significantly worse compared to the models that utilize the GCN architecture.

1. **Mol2Vec without GCN Training Results**:

    - The MLP models (MLP1, MLP2, MLP3) trained solely on the Mol2Vec representations, without the GCN, exhibit poor performance across all metrics.
    - The mean rank for these models ranges from 585.51 to 590.35, the MRR is between 0.032 and 0.038, and the Hits@1 is below 1%.
    - These results indicate that the Mol2Vec representations alone, without the additional structural information captured by the GCN, are not sufficient for effective text-molecule retrieval.
2. **GCN Training Results**:

    - In contrast, the GCN models (GCN1, GCN2, GCN3) that leverage the molecular graph structure show significantly improved performance.
    - The mean rank for the GCN models is remarkably low, ranging from 12.74 to 12.99, the MRR is between 0.365 and 0.388, and the Hits@1 is around 23-24%.
    - These results demonstrate the importance of explicitly incorporating the molecular graph information through the GCN architecture, as it allows the model to capture the structural characteristics of the molecules more effectively.
3. **Mol2Vec without GCN Test Results**:

    - The poor performance of the MLP models without the GCN is further corroborated by the test set results.
    - The mean rank for the MLP models in the test set is between 681.47 and 690.56, the MRR is around 0.030, and the Hits@1 is less than 1%.
    - These results confirm that the Mol2Vec representations alone are not sufficient for generalizing to the test set, highlighting the need for the GCN component to achieve high-performing text-molecule retrieval.

4. **GCN Test Results**:
    - The GCN models, on the other hand, maintain their strong performance on the test set, with mean ranks ranging from 55.47 to 58.95, MRR between 0.256 and 0.294, and Hits@1 around 20-21%.
    - These results demonstrate the GCN's ability to effectively capture the structural properties of the molecules, which enables the model to generalize well to the test set.

Results Discussion with Respect to the Hypothesis and Results from the original paper

The cross-modal embedding approach is effective for aligning text and molecules, and ensembling different architectures boosts performance by combining their strengths, strongly supporting hypotheses 1 and 2. The attention-based associations provide some interpretability but their quality needs further analysis (hypothesis 3). There are minor differences in MLP/GCN ranking capabilities supporting hypothesis 4, though more evidence is needed. Finally, the reranking methods do not improve over the base models, contradicting hypothesis 5. Overall, the embedding ensembles emerge as the most promising approach based on these results.

1. **Hypothesis 1**: The results strongly support this hypothesis. Both the MLP and GCN encoders are able to effectively align the text and molecule spaces, achieving high MRR values on the training set (around 0.60-0.61). This shows the cross-modal embedding approach is successful in mapping the two modalities into a common space for retrieval. While performance drops on the test set, the MRR values around 0.48-0.51 still indicate reasonable retrieval capability.

2. **Hypothesis 2**: This hypothesis is also supported by the results. The ensemble models combining multiple MLP or GCN models outperform the individual constituent models, both for the same architecture type and especially when ensembling across MLP and GCN. The All-Ensemble achieves the highest MRR of 0.509 on the test set compared to 0.499-0.517 for the individual architecture ensembles. Ensembling different architectures allows the model to benefit from their complementary strengths.

3. **Hypothesis 3**: The attention-based models provide some insights into text-molecule associations through the extracted rules. However, the coherence and usefulness of these rules is not directly evaluated in the results. More

qualitative analysis would be needed to assess the quality of the mined associations and their utility for interpretability.

4. **Hypothesis 4**: The results lend some support to this hypothesis. On the test set, the MLP-Ensemble achieves a slightly higher MRR of 0.517 compared to 0.499 for the GCN-Ensemble. This suggests the MLP may be better at ranking easier examples. However, the differences are relatively small. More detailed analysis of the rankings and examples where MLP/GCN perform better would help confirm this hypothesis.

5. **Hypothesis 5**: The results do not support this hypothesis. The attention model with reranking using extracted rules underperforms the base MLP and ensemble models on the test set MRR (around 0.50-0.51). The FPGrowth association rule reranking also does not improve over the base models. More work may be needed to effectively leverage the attention signals for reranking.

Discussion

The reproduction results support that the core components of the original paper were reproducible, but the attention-based portion of the model encountered issues that prevented the full reproducibility of the work intially. However, we've managed to crack the code conundrum and reproduce all the results from the paper successfully.

Interestingly, our results turned out to be slightly better than those reported in the paper. For example, MLP Ensemble in the paper is 0.520 for MRR and 35.1%/86.4% for Hits@1/10. Our results are 0.606 for MRR and 43.4%/93.8% for Hits@1/10. This has sparked our curiosity, and we are in discussion with the primary author of the paper to see if he has any insights into what might have caused this difference. Our initial assessment leans towards potential variations in hyper-parameters or differences in the transformer pre-trained model used.

1. Reproducibility of the Base Models:

    - The reproduction results for the MLP and GCN models show that we are able to successfully replicate the performance of the baseline architectures reported in the original paper.
    - The validation and test set results for the MLP and GCN models, including the mean rank, MRR, Hits@1, and Hits@10 metrics, are consistent with the original findings.

    - This suggests that the core model components, such as the text encoder, molecule encoder, and the contrastive loss function, were implemented correctly and could be reproduced.
2. Issues with the Attention-based Model weight extraction:

    - In the reproduction we were able to train the cross-modal attention model and use the attention weights.
    - However, we were unable to reproduce the results for the analysis of the attention weights and extracted rules due to problems with the code. Following extensive time and effort, we've managed to rectify the code, enabling us to effectively extract the weights of the Multi-Headed Attention (MHA). This breakthrough allows us to thoroughly evaluate the attention results, marking a significant achievement in our endeavors.
3. Factors Contributing to the Irreproducibility:

    - The lack of clarity in part of the codebase and implementation details are the primary factor that prevented the full reproducibility of the attention-based portion of the work initially. After investing significant time and effort, we successfully resolved the issue within the code.
    - This highlights the importance of providing comprehensive documentation, code, and implementation details when publishing research, as it directly impacts the ability of others to reproduce the work.

“What was easy”

- Model Architectures: The paper provides clear descriptions and diagrams of the model architectures like the text encoder, MLP/GCN molecule encoders, and cross-modal attention model. Running these models is relatively straightforward.

- Training Procedures: The authors specify the optimization algorithm (Adam), learning rates, batch sizes, and number of training epochs used, facilitating reproducibility of the training process.

- Loss Functions: The loss functions, including the symmetric contrastive loss and its modification with negative sampling, are well-documented and is easy to run based on the details provided.

- Evaluation Metrics: The metrics used (MRR, Hits@K, Mean Rank) are standard and easy to calculate given the model outputs and ground truth data.

What was difficult

- The most difficult part of the project has proven most arduous when grappling with the reproduction code. Unfortunately, the code fails to work as expected. Substantial time and dedication have been invested in fixing and refining it to align with the paper's intended outcomes. Countless fixes and updates have been made to ensure it works, in order for us to successfully replicate the results outlined in the paper.
- Computational Resources: The authors mention using NVIDIA V100 GPUs, but do not provide details on the specific hardware configurations (number of GPUs, memory, etc.) which can influence training times and results, especially for large models like SciBERT. As an example, in order to execute the reproduction, the system necessitates a minimum of 60GB of memory. Our initial setups failed due to the lack of information on specific computational requirements.
- Dataset Construction: While the authors mention using ChEBI and PubChem data, the exact filtering criteria and preprocessing steps to construct the ChEBI-20 dataset are not clearly specified, making it difficult to reproduce the dataset identically.
- Mol2vec Embeddings: The process of generating Mol2vec embeddings from SMILES strings is not described in detail. Different implementations or parameter choices could lead to variations in the molecule representations.
- Random Seeds: The random seeds used for weight initialization and data shuffling are not reported, which can cause differences in the final model outputs and rankings.
- Ensemble Details: The specifics of how the ensemble model combines rankings from different base models (e.g., the weighting scheme) are not clearly described, making it difficult to reproduce the ensemble results exactly.
- Hyperparameter Tuning: The paper does not provide details on hyperparameter search ranges or the criteria used for selecting the final hyperparameter values, which could affect reproducibility.

Recommendations to the original authors or others who work in this area for improving reproducibility

For the Authors:

- **Mol2vec Implementation Details:** Provide a detailed description of the Mol2vec embedding generation process, including the specific parameter settings, radius values, handling of rare/unknown substructures, and any other implementation choices made.

- **Random Seeds:** Report the random seeds used for weight initialization, data shuffling, and any other non-deterministic components. This would enable others to replicate the exact same runs.

- **Hyperparameter Tuning:** Document the hyperparameter search spaces, the tuning approach (manual, random search, etc.), and the criteria used for selecting the final hyperparameter values.

- **Attention-based Component:** To improve the reproducibility of the attention-based component, the authors of the original paper could consider releasing the following:

    - Providing detailed step-by-step instructions, along with the necessary data and pre-trained models, would further aid in the reproducibility of the attention-based aspect of the work.
    - Additionally, the authors could consider including the attention-based analysis and insights in the main body of the paper, rather than treating it as a separate hypothesis, to ensure that the core contributions are clearly documented and accessible.

For Others Working in This Area:

- **Follow Best Practices:** Adopt best practices for reproducible research, such as providing clear documentation, releasing code and data, and reporting all relevant implementation details.

- **Use Containerization:** Leverage containerization tools like Docker to package the entire software environment, making it easier to share and replicate experiments across different systems.

- **Automate Workflows:** Develop automated workflows or scripts that handle data preprocessing, model training, evaluation, and result reporting. This reduces manual effort and minimizes the risk of human errors affecting reproducibility.

- **Benchmark Datasets:** Collaborate on creating and maintaining benchmark datasets for cross-modal molecule retrieval, with clear data collection, curation, and validation processes.
- **Reproducibility Checklists:** Adopt reproducibility checklists or guidelines specific to the domain, ensuring that all relevant aspects are documented and shared with the research community.
- **Reproducibility Challenges:** Participate in reproducibility challenges or code evaluation efforts, which can help identify potential issues and improve the overall reproducibility of research in this area.

Public GitHub Repo

The reproduce code is published in the following GitHub DLH Text2Mol Repository.

You can download this notebook (DLH_Team_10.ipynb) from the DLH Text2Mol Repository.

Video Presentation

Watch the video presentation on YouTube.

References

```
@inproceedings{edwards2021text2mol,
  title={Text2Mol: Cross-Modal Molecule Retrieval with Natural
Language Queries},
  author={Edwards, Carl and Zhai, ChengXiang and Ji, Heng},
  booktitle={Proceedings of the 2021 Conference on Empirical
Methods in Natural Language Processing},
  pages={595--607},
  year={2021},
  url = {https://aclanthology.org/2021.emnlp-main.47/}
}
```