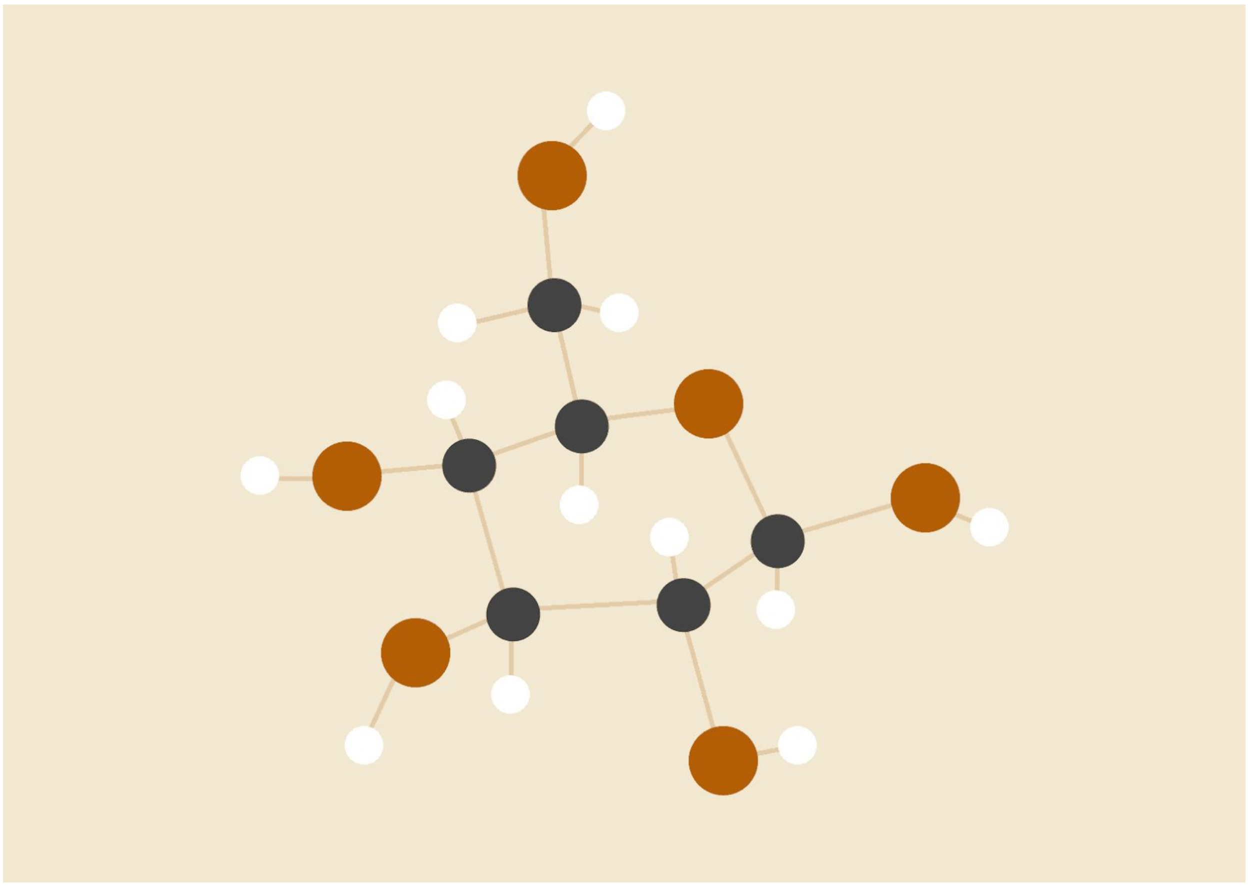
CSE 700: Independent Study
Drug Discovery and Repurposing



Supervised by
Prof. Mingchen Gao

Ankita Das (UB Person No.: 50317491)
Manisha Biswas (UB Person No.: 50317483)
Sriparna Chakraborty (UB Person No.: 50314303)
Upasana Ghosh (UB Person No.: 50317396)

Department of Computer Science
University at Buffalo
Buffalo, NY 14260

ABSTRACT

Given the high attrition rates, substantial costs, and slow pace of new drug discovery, repurposing of 'old' drugs to treat both common and rare diseases are increasingly becoming an essential resolution because it involves the use of non-toxic compounds, with potentially lower overall development costs within shorter development timelines. Various data-driven and experimental approaches have been suggested in this paper for the identification of the proper method; however, there are also major technological and regulatory challenges that need to be addressed. In this Review, we did a comparative study on TOX21 dataset using three different approaches, eg. Graph Convolutional Network, IBM RXN, Generative Adversarial Network, and received a satisfactory result for Graph Convolutional Network.

INTRODUCTION

Early drug discovery is an essential part of the pharmaceutical sector.The entire Drug discovery process during Clinical Trials takes a lot of time because there are multiple phases of testing namely Phase 1, Phase 2 and Phase 3 trials. Most of the time drugs compound fails testing at Phase 2 and Phase 3. The traditional process involves basic research to uncover targets that may be susceptible to attack, such as a disease-related protein receptor on the surface of particular cells. Then, scientists use techniques like high-throughput screening to see which compounds bind the target. After that, various methods of biological and chemical testing are used to fine-tune the structure or test other features, such as a compound's ability to reach the target in an organism.

The starting target is very essential in terms of drug discovery so Scientists and Biologists believe that utilizing Machine Learning techniques will help streamline the process into the more rigorous testing. Using advanced computational tools and simulations to create new molecules as well as faster processing and AI will help us generate more medicines and also allow better medicines to come in.

This methodology will help us to identify the new drugs in terms of Drug discovery and Drug Repurposing. The study here underlines three different approaches - the process of testing the toxicity of a drug using Graph Convolutional Model, identification and classification of new Drug Structure using IBM RXN and analysing the identified drug using Advanced Neural Network.

HYPOTHESIS

We are using Deep Learning, and AI methods to classify and identify new drugs which helps us to make a decision and reduce time for unnecessary walk-throughs for the entire clinical trial process as described above.

The process relates to methods for scientific experimentation especially used in drug discovery and relevant to the fields of biology and chemistry. Using robotics, data processing/control software, liquid handling devices, and sensitive detectors, high-throughput screening allows a researcher to quickly conduct millions of chemical, genetic, or pharmacological tests. Through this process one can rapidly identify active compounds, antibodies, or genes that modulate a particular bio-molecular pathway. The results of these experiments provide starting points for drug design and for understanding the non-interaction or role of a particular location.

Taking this as an inspiration we have tried to initially identify a new drug by combining Azithromycin and Hydroxychloroquine using IBM RXN which can serve as an medicine to Corona Virus and that gave us 40% confidence. Then we tried testing the toxicity of the drug on the TOX-21 dataset using Deep Chem. And ,Finally we tried implementing Advanced Graph convolutional Network to analyse the new drug.

DATASET

We looked through three datasets and did an Exploratory Data Analysis to understand which should be the best dataset to move forward in the study. All of the dataset are as below:

1. TOX21 : It is a dataset to measure qualitative toxicity of drugs on 12 biological targets, including nuclear receptors and stress response pathways
2. ClinTox : It is a dataset with qualitative data of drugs approved by the FDA that have failed clinical trials for toxicity reasons.
3. Chembl : It is a curated database of bioactive molecules with drug-like properties. It brings together chemical, bioactivity and genomic data to aid the translation of genomic information into effective new drugs

Looking into all the above three datasets we concluded that in our current scenario we should move ahead with the TOX-21 dataset because after identification and classification of a new drug, the most important phenomena is to test its toxicity.

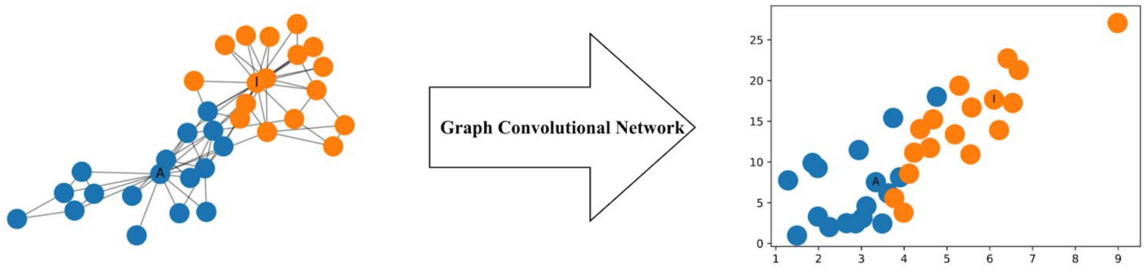
DEEPCHEM

DeepChem, is a framework which aims to provide a high quality open-source toolchain that democratizes the use of deep-learning in drug discovery, materials science, quantum chemistry, and biology.

APPROACH 1: USING GRAPH CONVOLUTIONAL NETWORK MODEL ON TOX21

GRAPH CONVOLUTIONAL NETWORK (GCN)

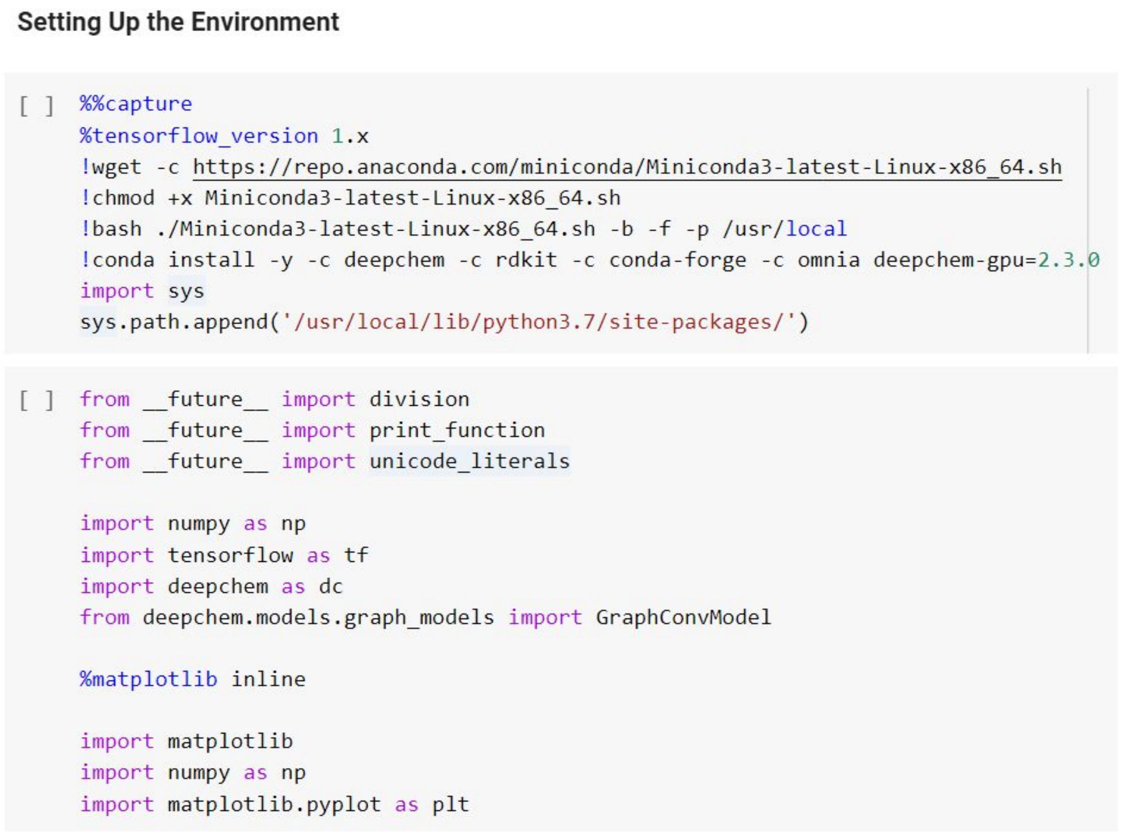
Graph Convolutional Networks are a type of deep learning architectures that are specifically designed to work with life science data. GCN is a powerful neural network architecture for machine learning on molecular data as they can be naturally visualized as graphical structures. GCNs are so powerful that even a randomly initiated 2-layer GCN can produce useful feature representations of the nodes in the networks. The figure below illustrates a 2-dimensional representation of each node in a network produced by such a GCN. We can see from the figure that the 2-dimensional model preserves the relative nearness of the nodes in the network even without any training.



APPLYING DEEPCHEM GCN ON TOX21

EXPERIMENTAL SETUP

We have used Google Collab to run DeepChem and imported the core `GraphConvModel` from the DeepChem `graph_models` library.

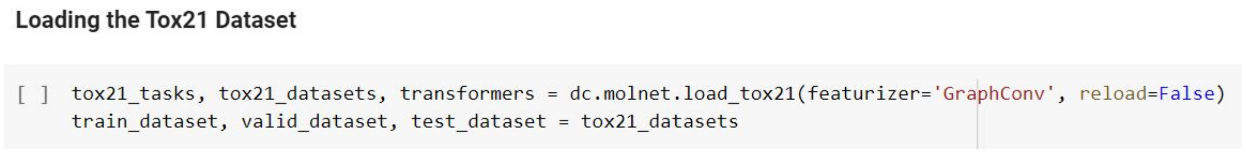


Setting Up the Environment

```
[ ]  %%capture
     %tensorflow_version 1.x
     !wget -c https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
     !chmod +x Miniconda3-latest-Linux-x86_64.sh
     !bash ./Miniconda3-latest-Linux-x86_64.sh -b -f -p /usr/local
     !conda install -y -c deepchem -c rdkit -c conda-forge -c omnia deepchem-gpu=2.3.0
     import sys
     sys.path.append('/usr/local/lib/python3.7/site-packages/')
```

```
[ ]  from __future__ import division
     from __future__ import print_function
     from __future__ import unicode_literals

     import numpy as np
     import tensorflow as tf
     import deepchem as dc
     from deepchem.models.graph_models import GraphConvModel

     %matplotlib inline

     import matplotlib
     import numpy as np
     import matplotlib.pyplot as plt
```

MoleculeNet suite has been used to load and preprocess the Tox21 dataset. The featurizer option has been set to 'GraphConv' in order to preprocess and transform the data to a form that can ensure a stable training of our GCN model. The MoleculeNet call also partitions the data set into training, validation and test sets for training and evaluating our model.



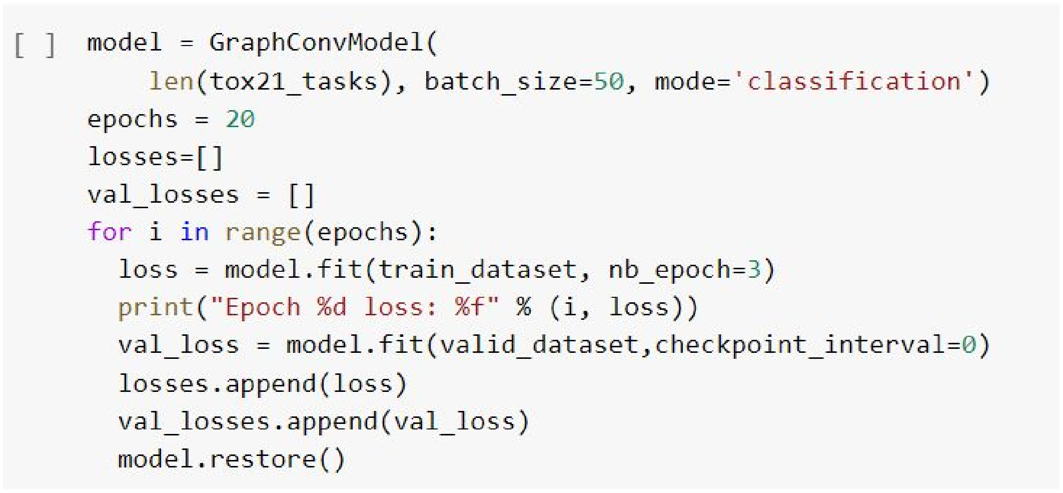
Loading the Tox21 Dataset

```
[ ]  tox21_tasks, tox21_datasets, transformers = dc.molnet.load_tox21(featurizer='GraphConv', reload=False)
     train_dataset, valid_dataset, test_dataset = tox21_datasets
```

MODEL TRAINING

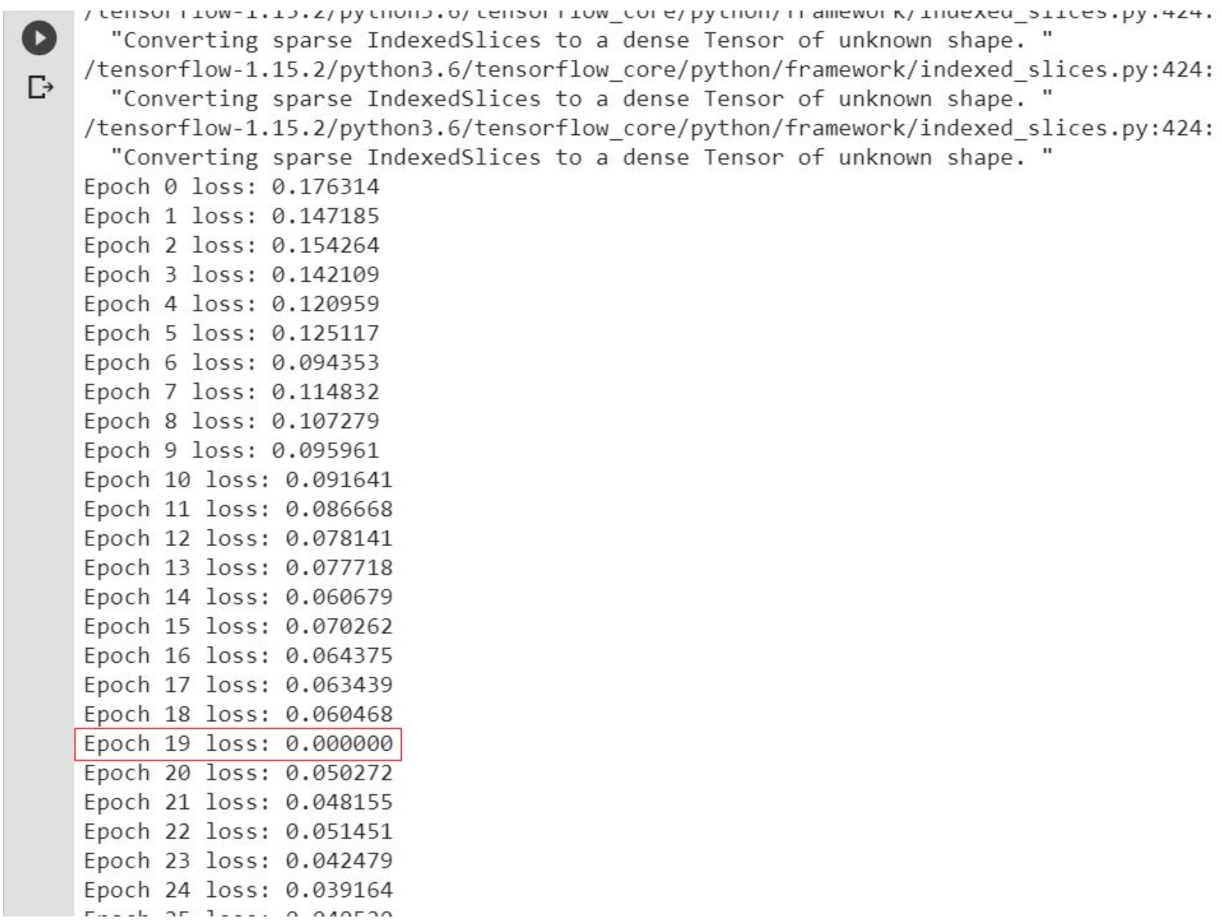
For training the GCN model on the loaded Tox21 train_dataset, we instantiate an object of the DeepChem class `GraphConvModel` that wraps a standard graph convolutional architecture underneath a wrapper for user convenience. We train this model object on the train_dataset and the valid_dataset. We also store the losses per epoch for each of the datasets for visualizing the model evaluation.

Training the model



```
[ ]  model = GraphConvModel(
         len(tox21_tasks), batch_size=50, mode='classification')
     epochs = 20
     losses=[]
     val_losses = []
     for i in range(epochs):
       loss = model.fit(train_dataset, nb_epoch=3)
       print("Epoch %d loss: %f" % (i, loss))
       val_loss = model.fit(valid_dataset,checkpoint_interval=0)
       losses.append(loss)
       val_losses.append(val_loss)
       model.restore()
```

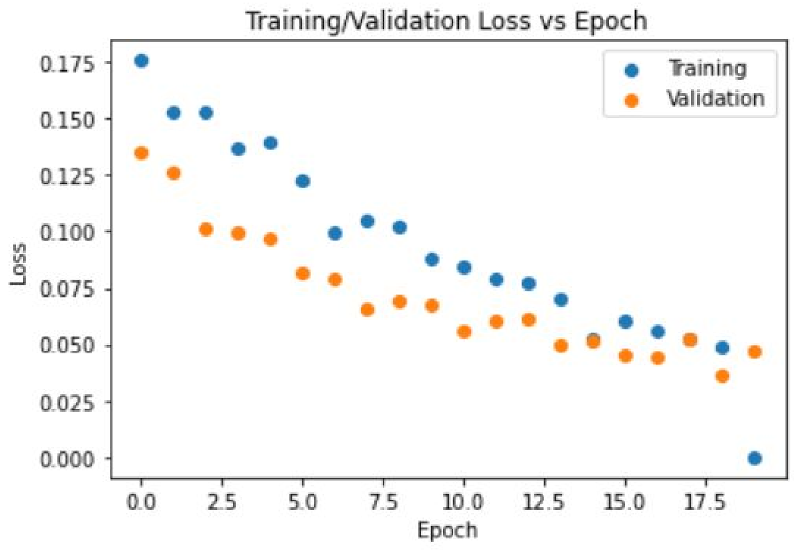
Tuning the hyperparameters for different values during the training, we found that the loss function converged at batch_size=50 and epoch=20.



```
/tensorflow-1.15.2/python3.6/tensorflow_core/python/framework/indexed_slices.py:424:
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
/tensorflow-1.15.2/python3.6/tensorflow_core/python/framework/indexed_slices.py:424:
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
/tensorflow-1.15.2/python3.6/tensorflow_core/python/framework/indexed_slices.py:424:
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
Epoch 0 loss: 0.176314
Epoch 1 loss: 0.147185
Epoch 2 loss: 0.154264
Epoch 3 loss: 0.142109
Epoch 4 loss: 0.120959
Epoch 5 loss: 0.125117
Epoch 6 loss: 0.094353
Epoch 7 loss: 0.114832
Epoch 8 loss: 0.107279
Epoch 9 loss: 0.095961
Epoch 10 loss: 0.091641
Epoch 11 loss: 0.086668
Epoch 12 loss: 0.078141
Epoch 13 loss: 0.077718
Epoch 14 loss: 0.060679
Epoch 15 loss: 0.070262
Epoch 16 loss: 0.064375
Epoch 17 loss: 0.063439
Epoch 18 loss: 0.060468
Epoch 19 loss: 0.000000
Epoch 20 loss: 0.050272
Epoch 21 loss: 0.048155
Epoch 22 loss: 0.051451
Epoch 23 loss: 0.042479
Epoch 24 loss: 0.039164
```

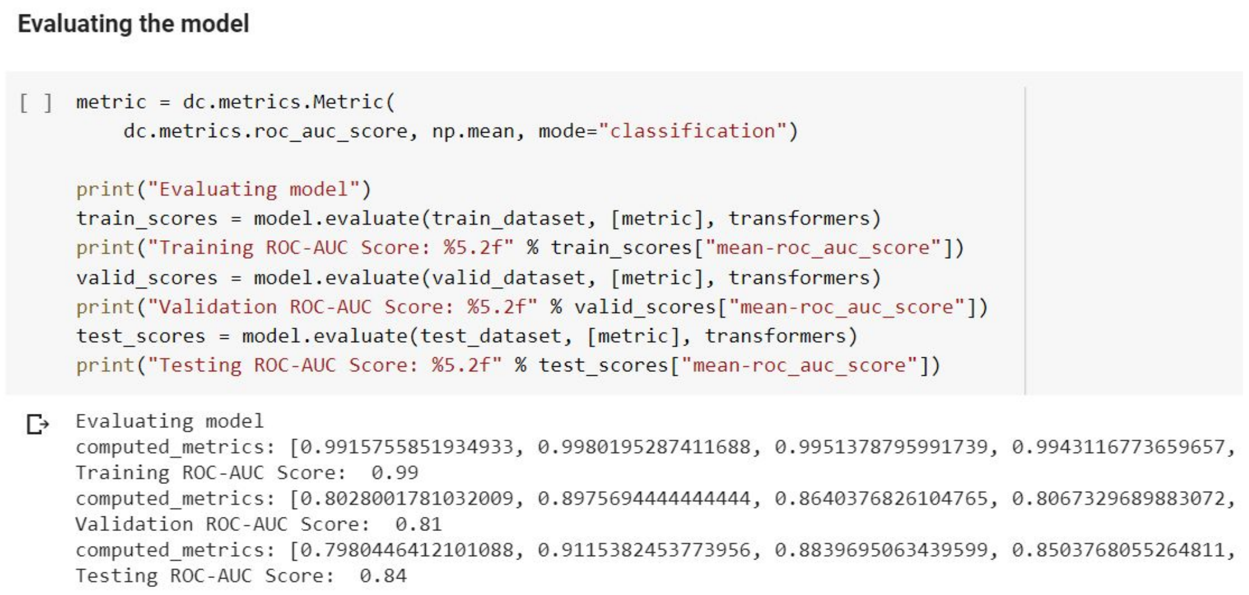
RESULTS AND OBSERVATIONS

The graph illustrating the training losses and validation losses against each epoch shows a stable training of the GCN model:



Evaluating the model against the training, validation and the testing data set, we got the following ROC-AUC scores:

DATASET	ROC-AUC SCORE
Training	0.99
Validation	0.81
Testing	0.84



Evaluating the model

```
[ ]  metric = dc.metrics.Metric(
         dc.metrics.roc_auc_score, np.mean, mode="classification")

     print("Evaluating model")
     train_scores = model.evaluate(train_dataset, [metric], transformers)
     print("Training ROC-AUC Score: %5.2f" % train_scores["mean-roc_auc_score"])
     valid_scores = model.evaluate(valid_dataset, [metric], transformers)
     print("Validation ROC-AUC Score: %5.2f" % valid_scores["mean-roc_auc_score"])
     test_scores = model.evaluate(test_dataset, [metric], transformers)
     print("Testing ROC-AUC Score: %5.2f" % test_scores["mean-roc_auc_score"])
```

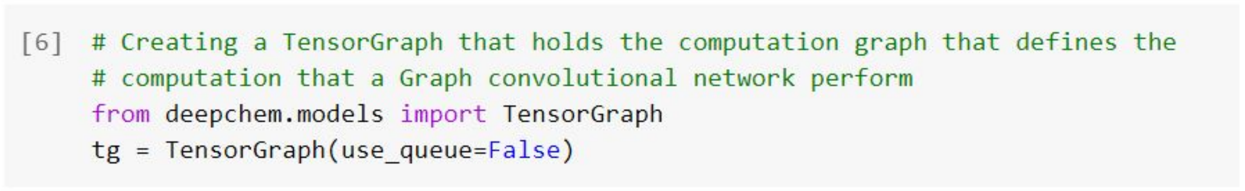
```
Evaluating model
computed_metrics: [0.9915755851934933, 0.9980195287411688, 0.9951378795991739, 0.9943116773659657,
Training ROC-AUC Score:  0.99
computed_metrics: [0.8028001781032009, 0.8975694444444444, 0.8640376826104765, 0.8067329689883072,
Validation ROC-AUC Score:  0.81
computed_metrics: [0.7980446412101088, 0.9115382453773956, 0.8839695063439599, 0.8503768055264811,
Testing ROC-AUC Score:  0.84
```

DEEP DIVE INTO GCNs: Building the GCN Model using TensorGraphs

To better understand how GCNs work, we tried to implement the GCN model ourselves from scratch. The first step was to create a TensorGraph object. This object would hold the “computational graph” that defines the computation that a graph convolutional network would perform. Next, we needed to define the inputs to our model. Conceptually, graph convolutions just require the structure of the molecule in question and a vector of features for every atom that describes the local chemical environment. However in practice, due to TensorFlow's limitations as a general programming environment, we also needed to have some preprocessed auxiliary information.

EXPERIMENTAL SETUP

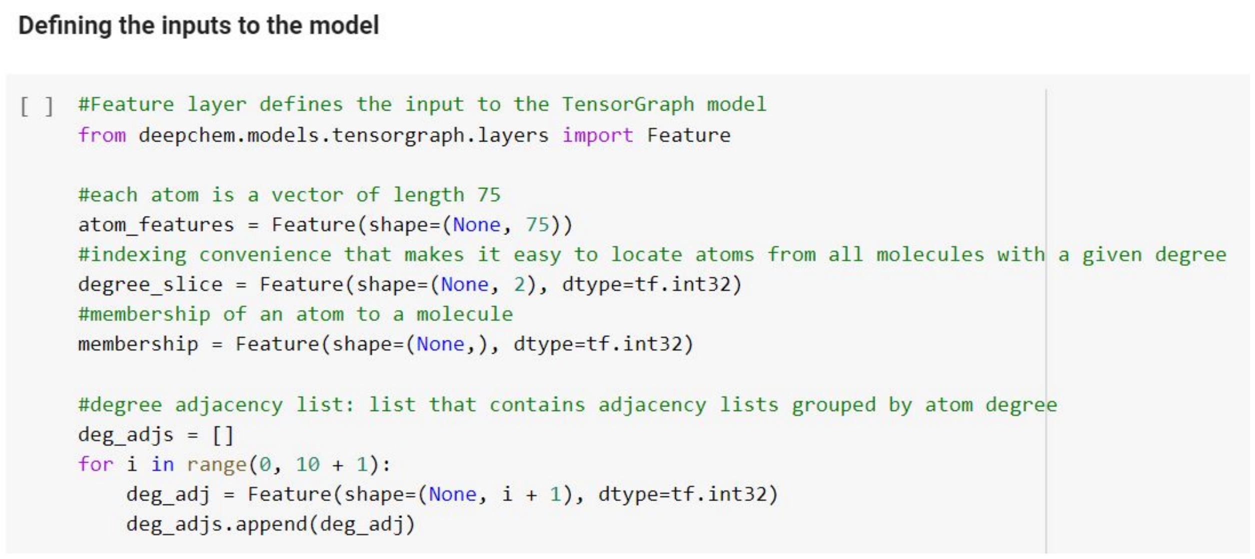
We defined the TensorGraph object as follows:



```
[6]  # Creating a TensorGraph that holds the computation graph that defines the
     # computation that a Graph convolutional network perform
     from deepchem.models import TensorGraph
     tg = TensorGraph(use_queue=False)
```

Next, we defined the model inputs. We defined a variable ‘atom_features’ to hold a feature vector of length 75 for each atom. The other inputs are required to support minibatching in TensorFlow. The variable ‘degree_slice’ is used for indexing convenience that makes it easy to locate atoms from all molecules with a given degree. ‘membership’ determines the membership of atoms in molecules (atom i belongs to molecule membership[i]). ‘deg_adjs’ is a list that contains adjacency lists grouped by the atom degree.

To define feature inputs with Keras, we use the Input layer. Conceptually, a model is a mathematical graph composed of layer objects. Input layers have to be the root nodes of the graph since they constitute the inputs.

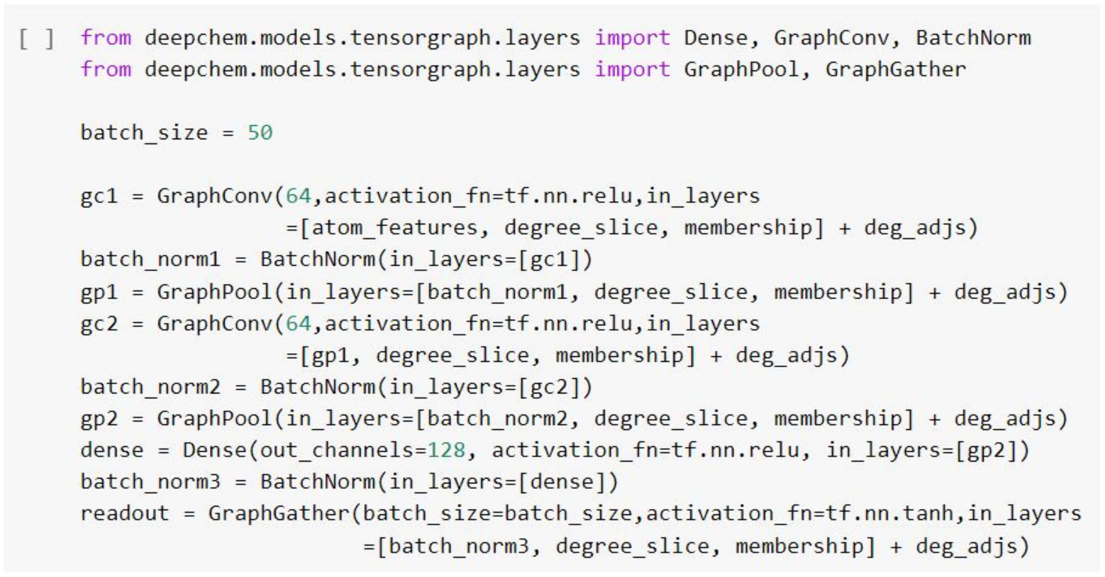


Defining the inputs to the model

```
[ ]  #Feature layer defines the input to the TensorGraph model
     from deepchem.models.tensorgraph.layers import Feature

     #each atom is a vector of length 75
     atom_features = Feature(shape=(None, 75))
     #indexing convenience that makes it easy to locate atoms from all molecules with a given degree
     degree_slice = Feature(shape=(None, 2), dtype=tf.int32)
     #membership of an atom to a molecule
     membership = Feature(shape=(None,), dtype=tf.int32)

     #degree adjacency list: list that contains adjacency lists grouped by atom degree
     deg_adjs = []
     for i in range(0, 10 + 1):
         deg_adj = Feature(shape=(None, i + 1), dtype=tf.int32)
         deg_adjs.append(deg_adj)
```

To implement the body of the graph convolutional network, we used the standard neural network layers such as ‘Dense’ and ‘BatchNormalization’. The layers that we add provide a "feature transformation" that will create one vector for each molecule.

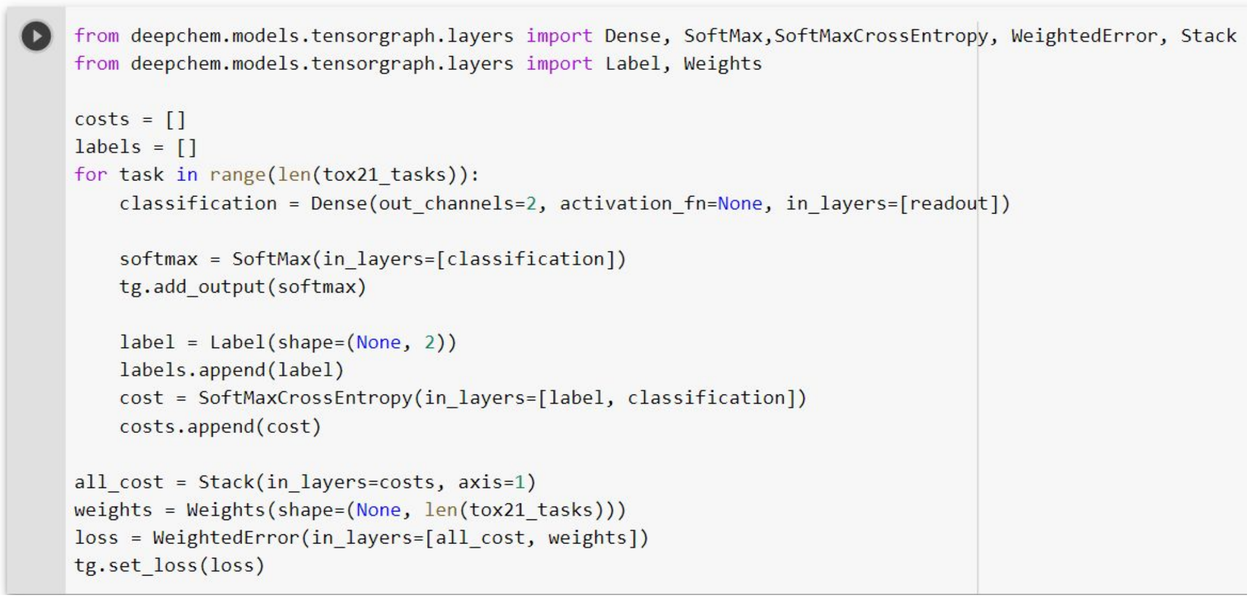
Implementing the body of the Graph Convolutional Network



```
[ ] from deepchem.models.tensorgraph.layers import Dense, GraphConv, BatchNorm
    from deepchem.models.tensorgraph.layers import GraphPool, GraphGather

    batch_size = 50

    gc1 = GraphConv(64,activation_fn=tf.nn.relu,in_layers
                    =[atom_features, degree_slice, membership] + deg_adjs)
    batch_norm1 = BatchNorm(in_layers=[gc1])
    gp1 = GraphPool(in_layers=[batch_norm1, degree_slice, membership] + deg_adjs)
    gc2 = GraphConv(64,activation_fn=tf.nn.relu,in_layers
                    =[gp1, degree_slice, membership] + deg_adjs)
    batch_norm2 = BatchNorm(in_layers=[gc2])
    gp2 = GraphPool(in_layers=[batch_norm2, degree_slice, membership] + deg_adjs)
    dense = Dense(out_channels=128, activation_fn=tf.nn.relu, in_layers=[gp2])
    batch_norm3 = BatchNorm(in_layers=[dense])
    readout = GraphGather(batch_size=batch_size,activation_fn=tf.nn.tanh,in_layers
                          =[batch_norm3, degree_slice, membership] + deg_adjs)
```

Next we make predictions from the Tensorgraph model by defining a loss for the model which tells the network the objective to minimize during training. The output layers are denoted with `TensorGraph.add_output(layer)`. Similarly, we tell the network its loss with `TensorGraph.set_loss(loss)`.

Predictions on TensorGraph model from Tox21 dataset



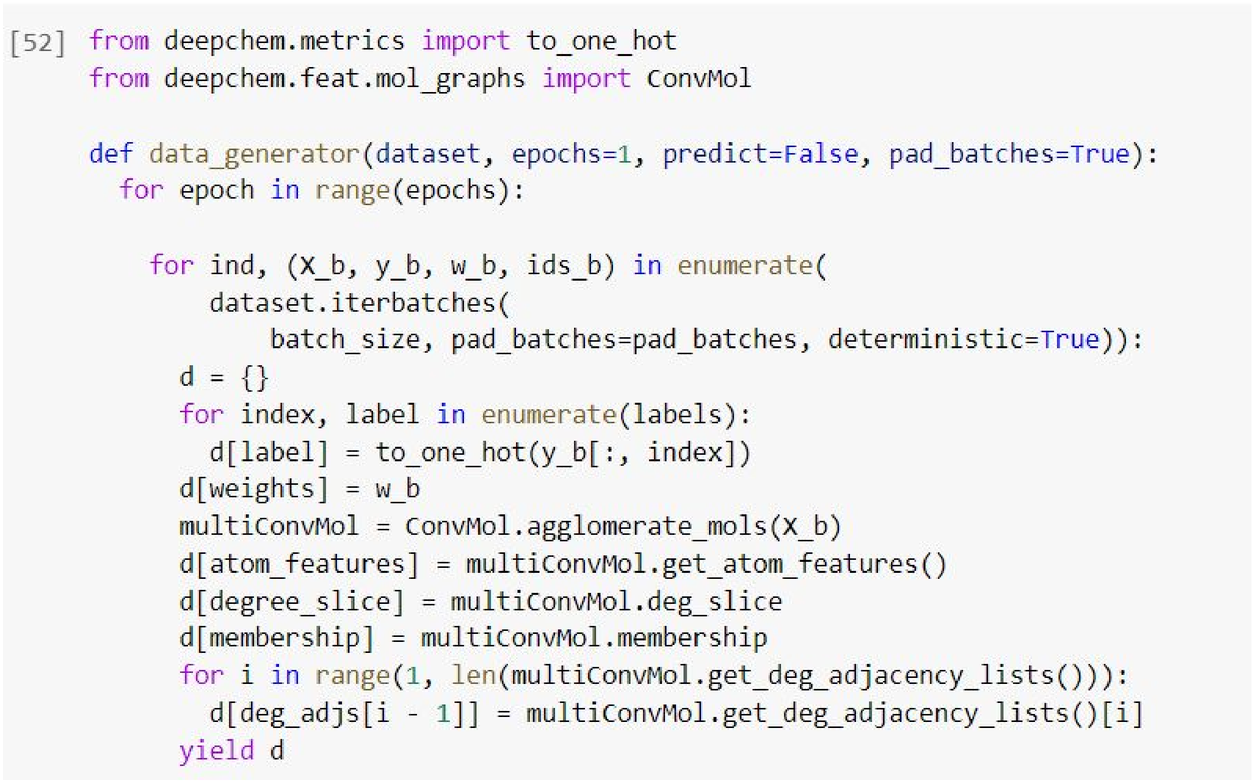
```
from deepchem.models.tensorgraph.layers import Dense, SoftMax,SoftMaxCrossEntropy, WeightedError, Stack
from deepchem.models.tensorgraph.layers import Label, Weights

costs = []
labels = []
for task in range(len(tox21_tasks)):
    classification = Dense(out_channels=2, activation_fn=None, in_layers=[readout])

    softmax = SoftMax(in_layers=[classification])
    tg.add_output(softmax)

    label = Label(shape=(None, 2))
    labels.append(label)
    cost = SoftMaxCrossEntropy(in_layers=[label, classification])
    costs.append(cost)

all_cost = Stack(in_layers=costs, axis=1)
weights = Weights(shape=(None, len(tox21_tasks)))
loss = WeightedError(in_layers=[all_cost, weights])
tg.set_loss(loss)
```

MODEL TRAINING

We trained the model by calling `fit()` and created a Python generator that, given a batch of data, generates a dictionary whose keys are the `Feature` layers and whose values are Numpy arrays we have used for this step of training.

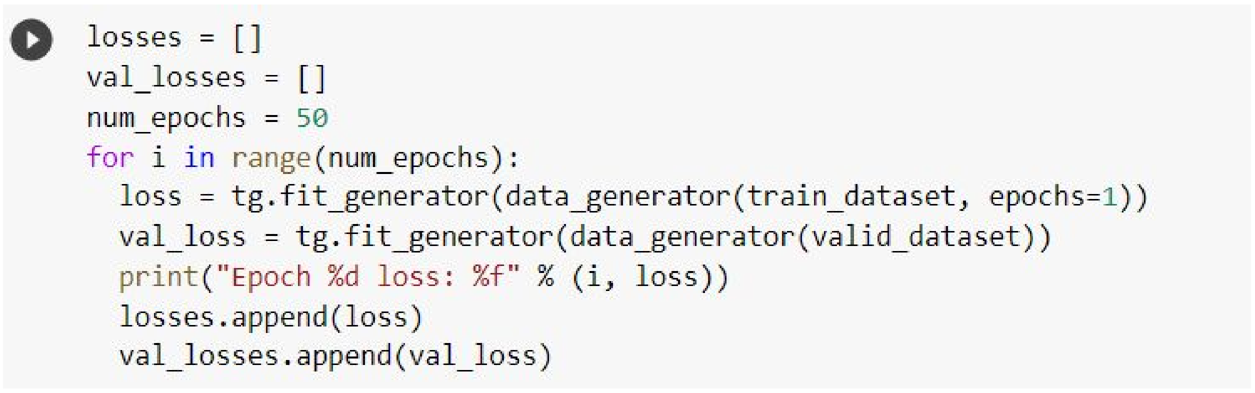
We have trained the model using `TensorGraph.fit_generator(generator)` which will use the generator that we have defined to train the model. We have taken the number of epochs as 50.

Training the model



```
[52] from deepchem.metrics import to_one_hot
     from deepchem.feat.mol_graphs import ConvMol

     def data_generator(dataset, epochs=1, predict=False, pad_batches=True):
       for epoch in range(epochs):

         for ind, (X_b, y_b, w_b, ids_b) in enumerate(
             dataset.iterbatches(
                 batch_size, pad_batches=pad_batches, deterministic=True)):
           d = {}
           for index, label in enumerate(labels):
             d[label] = to_one_hot(y_b[:, index])
           d[weights] = w_b
           multiConvMol = ConvMol.agglomerate_mols(X_b)
           d[atom_features] = multiConvMol.get_atom_features()
           d[degree_slice] = multiConvMol.deg_slice
           d[membership] = multiConvMol.membership
           for i in range(1, len(multiConvMol.get_deg_adjacency_lists())):
             d[deg_adjs[i - 1]] = multiConvMol.get_deg_adjacency_lists()[i]
           yield d
```

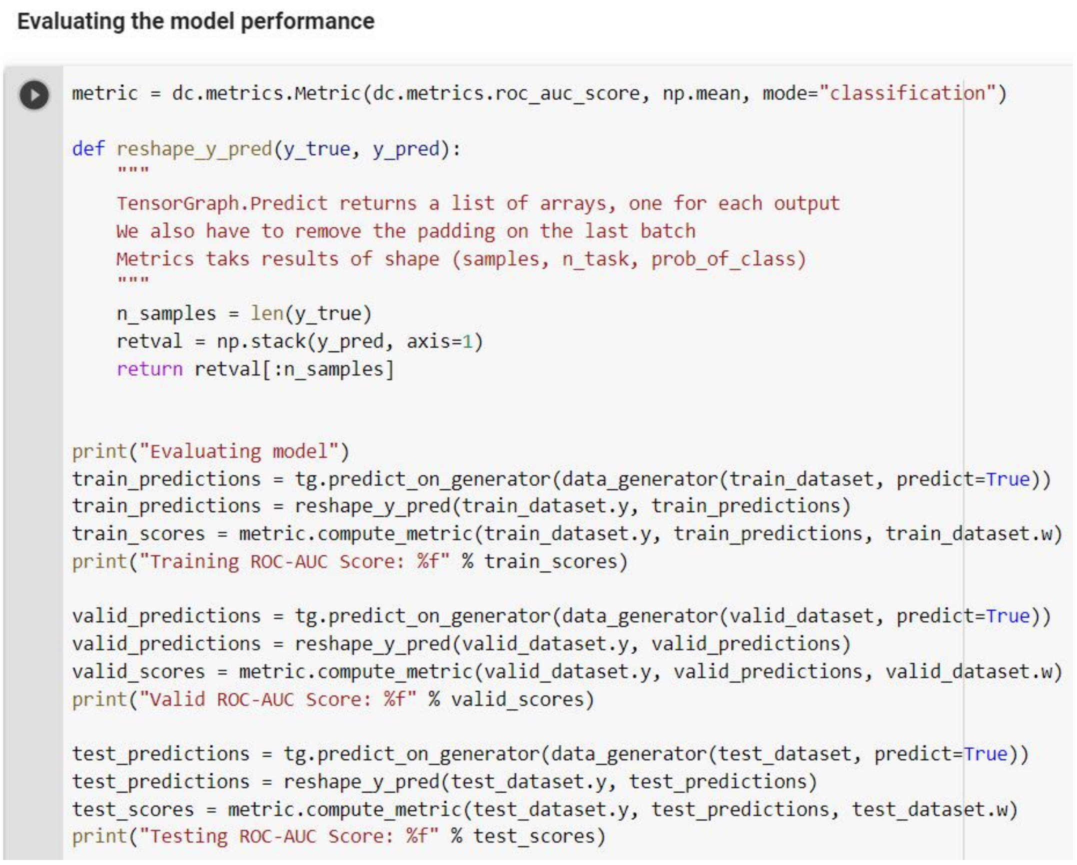
We also store the losses per epoch for each of the datasets for visualizing the model evaluation.



```
losses = []
val_losses = []
num_epochs = 50
for i in range(num_epochs):
  loss = tg.fit_generator(data_generator(train_dataset, epochs=1))
  val_loss = tg.fit_generator(data_generator(valid_dataset))
  print("Epoch %d loss: %f" % (i, loss))
  losses.append(loss)
  val_losses.append(val_loss)
```

RESULTS AND OBSERVATIONS

We have used the defined generator to evaluate the model performance.

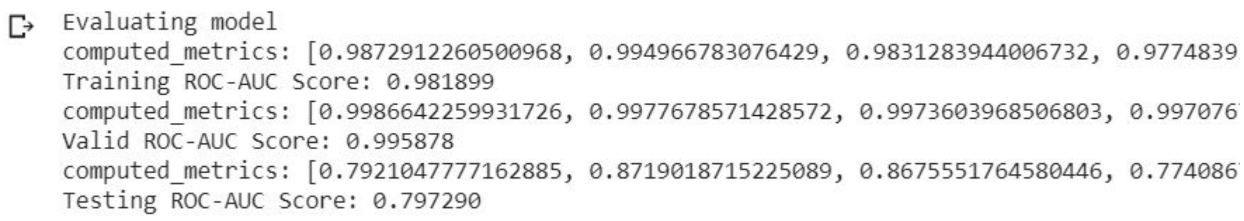


Evaluating the model performance

```
metric = dc.metrics.Metric(dc.metrics.roc_auc_score, np.mean, mode="classification")

def reshape_y_pred(y_true, y_pred):
    """
    TensorGraph.Predict returns a list of arrays, one for each output
    We also have to remove the padding on the last batch
    Metrics taks results of shape (samples, n_task, prob_of_class)
    """
    n_samples = len(y_true)
    retval = np.stack(y_pred, axis=1)
    return retval[:n_samples]


print("Evaluating model")
train_predictions = tg.predict_on_generator(data_generator(train_dataset, predict=True))
train_predictions = reshape_y_pred(train_dataset.y, train_predictions)
train_scores = metric.compute_metric(train_dataset.y, train_predictions, train_dataset.w)
print("Training ROC-AUC Score: %f" % train_scores)

valid_predictions = tg.predict_on_generator(data_generator(valid_dataset, predict=True))
valid_predictions = reshape_y_pred(valid_dataset.y, valid_predictions)
valid_scores = metric.compute_metric(valid_dataset.y, valid_predictions, valid_dataset.w)
print("Valid ROC-AUC Score: %f" % valid_scores)

test_predictions = tg.predict_on_generator(data_generator(test_dataset, predict=True))
test_predictions = reshape_y_pred(test_dataset.y, test_predictions)
test_scores = metric.compute_metric(test_dataset.y, test_predictions, test_dataset.w)
print("Testing ROC-AUC Score: %f" % test_scores)
```

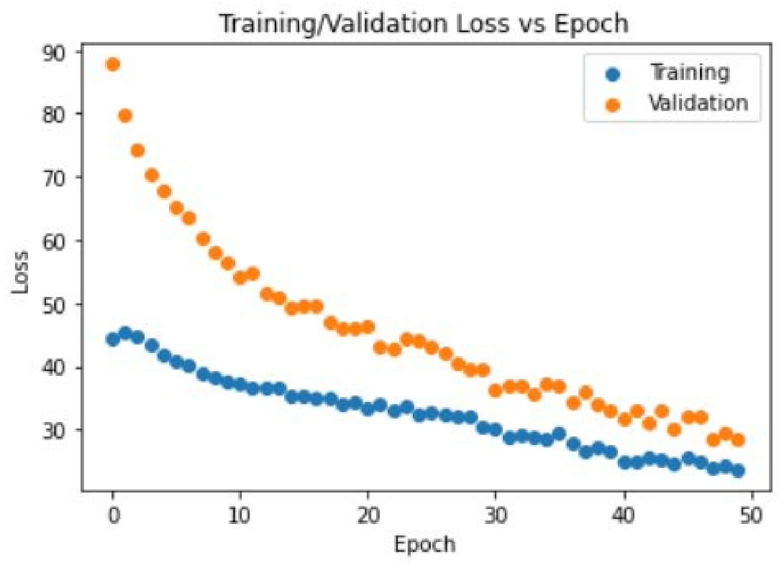
Evaluating the model against the training, validation and the testing data set, we got the following ROC-AUC scores:

DATASET	ROC-AUC SCORE
Training	0.98
Validation	0.99
Testing	0.80



```
Evaluating model
computed_metrics: [0.9872912260500968, 0.994966783076429, 0.9831283944006732, 0.9774839
Training ROC-AUC Score: 0.981899
computed_metrics: [0.9986642259931726, 0.9977678571428572, 0.9973603968506803, 0.997076
Valid ROC-AUC Score: 0.995878
computed_metrics: [0.7921047777162885, 0.8719018715225089, 0.8675551764580446, 0.774086
Testing ROC-AUC Score: 0.797290
```

The graph illustrating the training losses and validation losses against each epoch shows a stable training of the GCN model:



APPROACH 2: Using IBM RXN

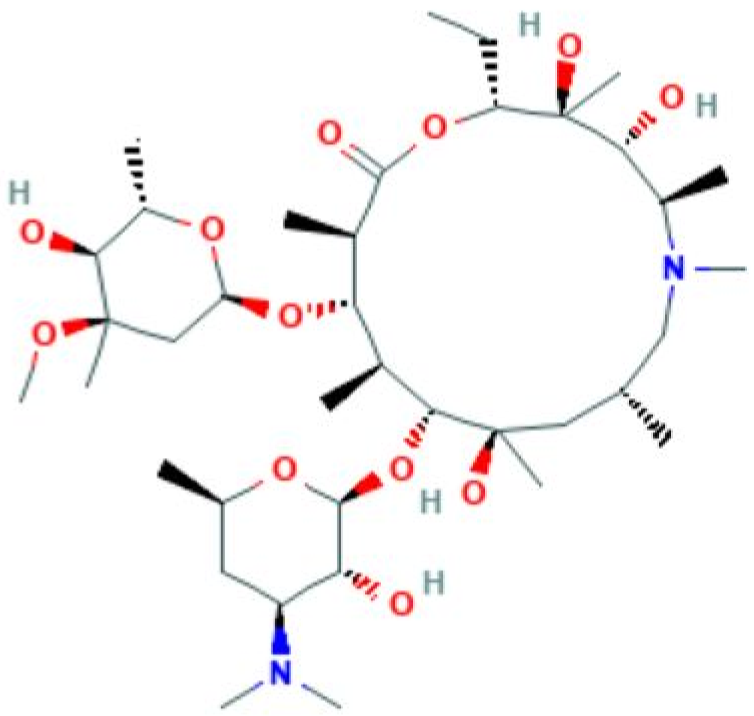
IBM provides an advanced Artificial Intelligence tool called IBM RXN which is useful in daily research activities and experiments. It can predict chemical reactions very quickly.

IBM RXN for Chemistry uses a system known as a simplified molecular-input line-entry system or SMILES. SMILES is used to represent a molecule as a sequence of characters. The model was trained using a combination of reaction datasets, equivalent to a total of 2 million reactions.

Ketcher is a web-based chemical structure editor that is designed for chemists, lab scientists, and technicians. It involves selecting, modifying, and erasing the connected, and unconnected atom bonds. Here we can also check the compatibility, toxicity, and confidence by combining two or more drugs. Being mindful of the current situation of Corona, we are trying to make a reaction between Azithromycin and Hydrochloroquine, potential essential compounds of corona medicine.

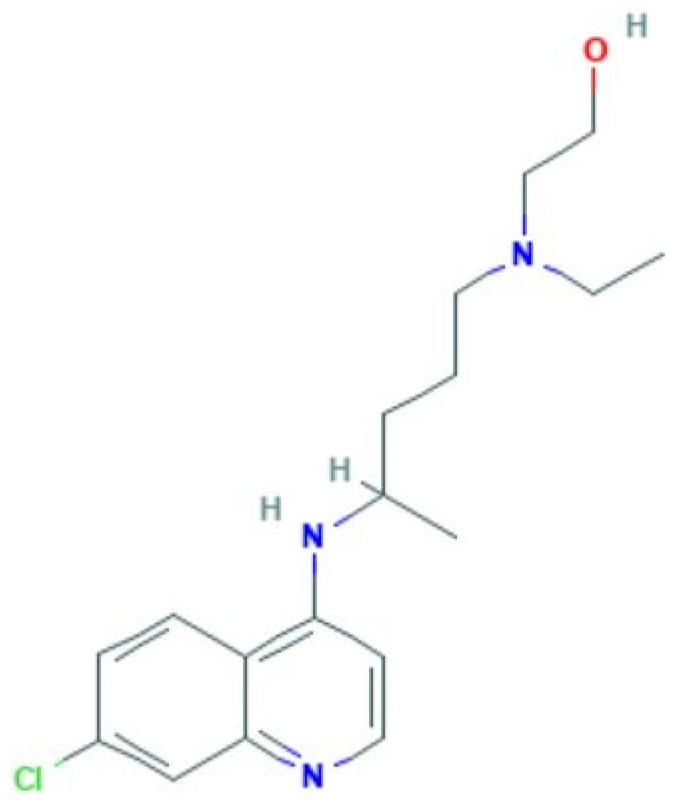
Molecular Formula for Azithromycin: $C_{38}H_{72}N_2O_{12}$

Chemical Structure:

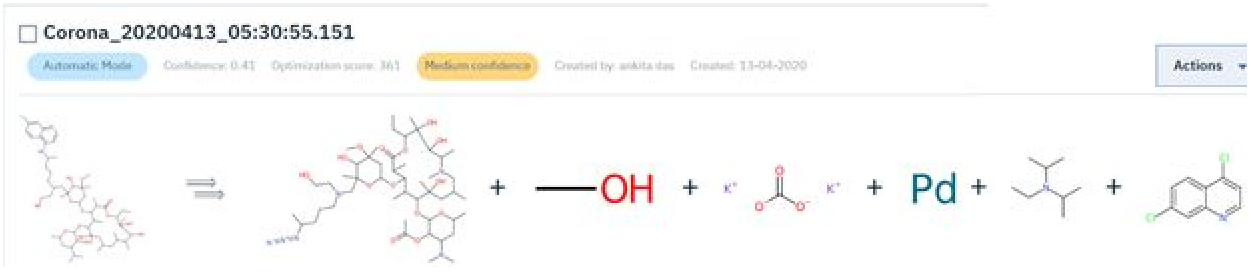


Molecular Formula for Hydroxychloroquine: $C_{18}H_{26}ClN_3O$

Chemical Structure:



Below is the snippet of the result of the reaction which we get after combining Azithromycin and Hydroxychloroquine. We receive a confidence of 41% here.


☐ Corona_20200413_05:30:55.151

Automatic Mode   Confidence: 0.41   Optimization score: 361   Medium confidence   Created by: ankita das   Created: 13-04-2020

Information about the retrosynthesis

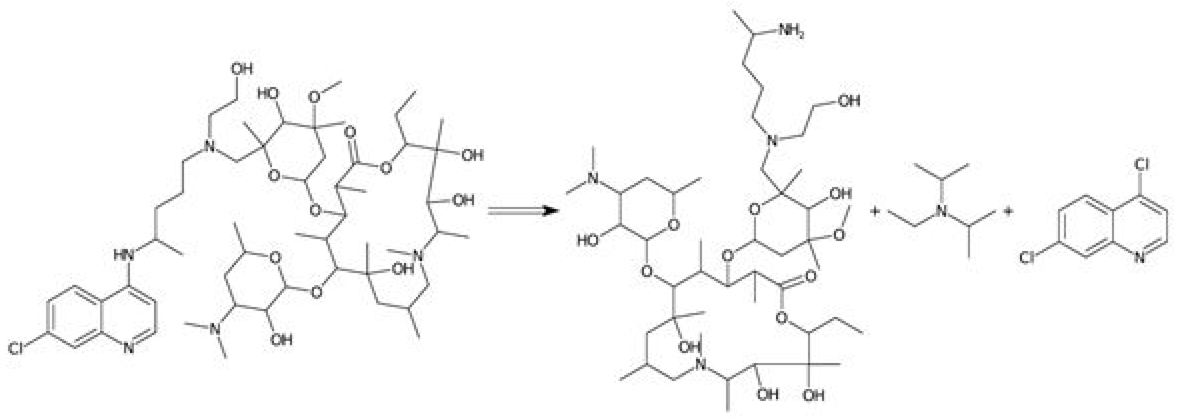
Created On: 2020-04-13T05:27:41.954000
Model: MolecularTransformer_v2.0_R-Inchi-MolecularTransformer_v2.0_F
Product: C(C)1(OC(CC(OC)(C)C1O)OC1C(C)C(OC(C(O)(C)C(O)C(C)N(C)CC(C)CC(O)(C)C(OC2OC(C)CC(N(C)C)C2O)C1C)CC)=O)CN(CCO)CCCC(NC1C=CN=C2C=1C=CC(Cl)=C2)C
MSSR: 3
FAP: 0.65
MRP: 10
SbP: 1
Available smiles:
Exclude smiles: CCC1C(C(C(N(CC(CC(C(C(C(C(C(=O)O1)C)OC2CC(C(C(O2)C)O)(C)OC)C)OC3C(C(CC(O3)C)N(C)C)O)(C)O)C)C)C)O)(C)O, C(C)1(OC(CC(OC)(C)C1O)OC1C(C)C(OC(C(O)(C)C(O)C(C)N(C)CC(C)CC(O)(C)C(OC2OC(C)CC(N(C)C)C2O)C1C)CC)=O)CN(CCO)CCCC(NC1C=CN=C2C=1C=CC(Cl)=C2)C
Exclude substructures: CCN(CCCC(C)NC1=C2C=CC(=CC2=NC=C1)Cl)CCO
Availability pricing threshold: 0

Sequence 0, Confidence: 0.41

Step 1

Type: Chloro N-arylation, Confidence: 0.504

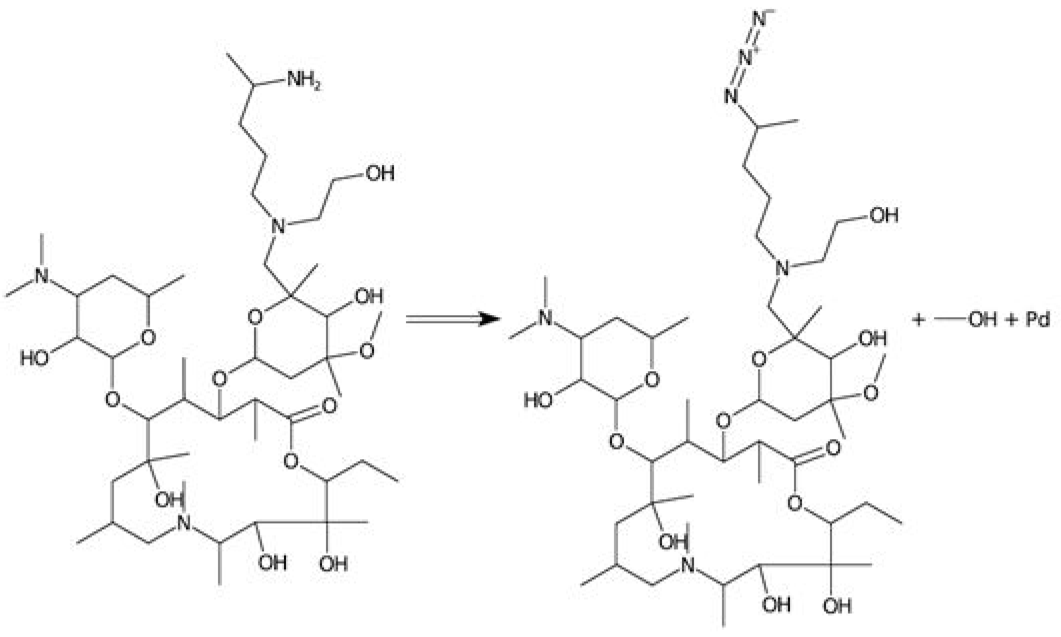
CCC1OC(=O)C(C)C(OC2CC(C)(OC)C(O)C(C)(CN(CCO)CCCC(C)N)O2)C(C)C(OC2OC(C)CC(N(C)C)C2O)C(C)(O)CC(C)CN(C)C(C)C(O)C1(C)O.CCN(C(C)C)C(C)C.Clc1ccc2c(Cl)ccnc2c1>>CCC1OC(=O)C(C)C(OC2CC(C)(OC)C(O)C(C)(CN(CCO)CCCC(C)Nc3ccnc4cc(Cl)ccc34)O2)C(C)C(OC2OC(C)CC(N(C)C)C2O)C(C)(O)CC(C)CN(C)C(C)C(O)C1(C)O



Step 2

Type: Azido to amino, Confidence: 0.913

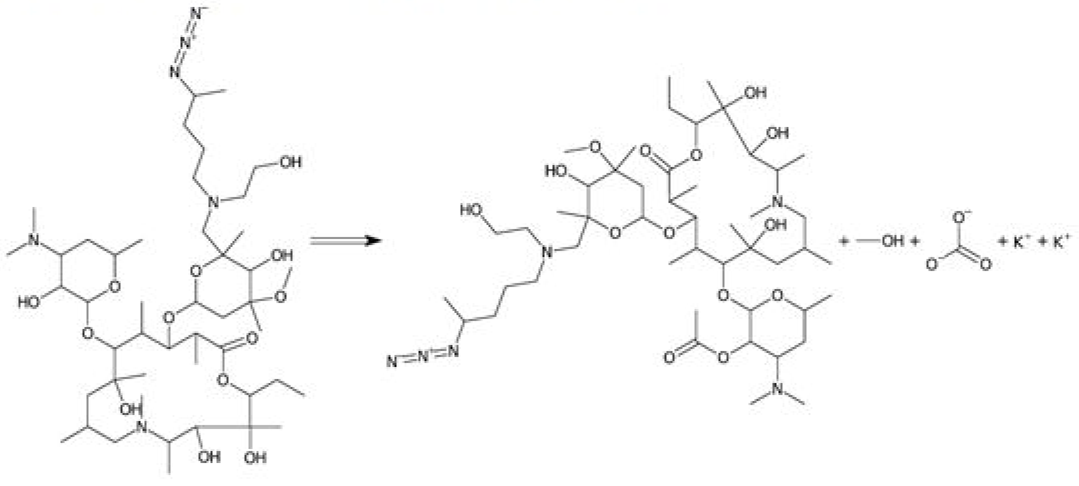
CCC1OC(=O)C(C)C(OC2CC(C)(OC)C(O)C(C)(CN(CCO)CCCC(C)N=[N+]=[N-])O2)C(C)C(OC2O C(C)CC(N(C)C)C2O)C(C)(O)CC(C)CN(C)C(C)C(O)C1(C)O.CO.[Pd]>>CCC1OC(=O)C(C)C(OC2C C(C)(OC)C(O)C(C)(CN(CCO)CCCC(C)N)O2)C(C)C(OC2OC(C)CC(N(C)C)C2O)C(C)(O)CC(C)CN(C)C(C)C(O)C1(C)O



Step 3

Type: O-Ac deprotection, Confidence: 0.89

CCC1OC(=O)C(C)C(OC2CC(C)(OC)C(O)C(C)(CN(CCO)CCCC(C)N=[N+]=[N-])O2)C(C)C(OC2O C(C)CC(N(C)C)C2OC(C)=O)C(C)(O)CC(C)CN(C)C(C)C(O)C1(C)O.CO.O=C([O-])[O-].[K+].[K+]>> CCC1OC(=O)C(C)C(OC2CC(C)(OC)C(O)C(C)(CN(CCO)CCCC(C)N=[N+]=[N-])O2)C(C)C(OC2O C(C)CC(N(C)C)C2O)C(C)(O)CC(C)CN(C)C(C)C(O)C1(C)O



APPROACH 3: Using Generative Adversarial Network on TOX21

A Generative Adversarial Network (GAN) is a type of generative model. It consists of two parts called the "generator" and the "discriminator". The generator takes random values as input and transforms them into an output that resembles the training data. The discriminator takes a set of samples as input and tries to distinguish the real training samples from the ones created by the generator. Both of them are trained together. The discriminator tries to get better and better at telling real from false data, while the generator tries to get better and better at fooling the discriminator.

DeepChem has an inbuilt GAN module which can be used to implement different types of GANs, one such is Conditional GAN (CGAN).

But, after going through a few papers[4][5] and articles[6][7][8], we figured that GANs work best on Image Datasets like MNIST dataset. So, implementing it on structured data like the TOX-21 dataset, would require building a custom GAN.

CONCLUSION

From our experiment, we observed the following:

1. The Deepchem GraphConv module performed better than the TensorGraph model approach.
2. By using IBM RXN, we made a reaction between two compounds Azithromycin and Hydroxychloroquine and received a confidence of 41% which is showing the non-toxicity nature of the resultant drug.
3. GANs work best on image datasets. Custom GANs are required to work successfully on structured data.

REFERENCES

1. www.towardsdatascience.com - How to do Deep Learning on Graphs with Graph Convolutional Networks by Tobias Skovgaard Jepsen
2. Deepchem Tutorials - Introduction to Graph Convolutions
3. Deepchem Notebook - Graph Convolutions For Tox21

4. Ian J. Goodfellow, Generative Adversarial Networks

5. Zhijie Deng, Structured Generative Adversarial Networks

6. DeepChem Tutorial - Conditional Generative Adversarial Network

7. DeepChem Tutorial - Training a Generative Adversarial Network on MNIST

8. Aviv Elbag's youtube video on Generative Adversarial Network (GANs) Full Coding Example Tutorial in Tensorflow 2.0

9. Article on Drug Repurposing https://www.nature.com/articles/nrd.2018.168

10. https://pubchem.ncbi.nlm.nih.gov/compound/Azithromycin#section=Vapor-Pressure

11. Chemical composition of Azithromycin https://hub.packtpub.com/say-hello-to-ibm-rxn-a-free-ai-tool-in-ibm-cloud-for-predicting-chemical-reactions/

12. Chemical composition of Hydroxychloroquine

13. https://pubchem.ncbi.nlm.nih.gov/compound/Hydroxychloroquine#section=2D-Structure