Long Nguyen
Ming-Shih Wang

Project 3

Introduction

This project aims to fine-tune and optimize a method to generate accurate predictions for the toxicity of molecules in the Tox21 dataset. Although we spent the majority of our time using data manipulation for data imbalances, fine-tuning, and trying different methods and models, due to time constraints, we were still looking for a way to overcome the hurdle of a 0.7855% validation score.

Contributions

Long Nguyen: data preprocessing, model fine-tuning, different-model searching, report, training
Ming-Shih Wang: data preprocessing, model fine-tuning, different-model searching, report, training

Preprocess The Data

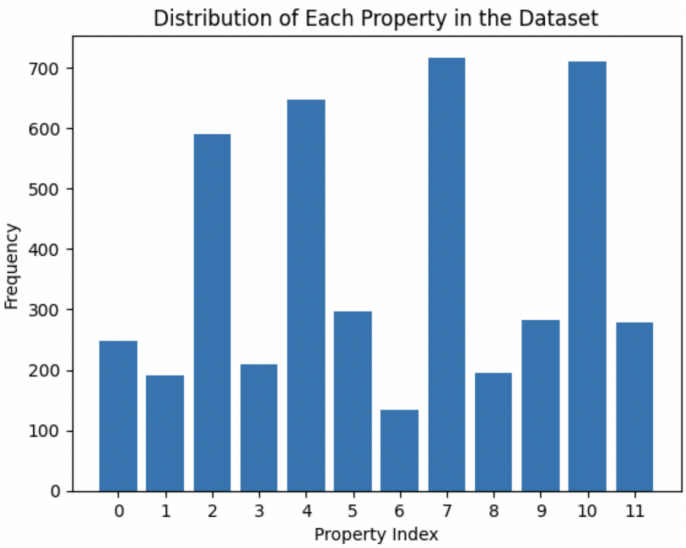
Balancing Each Property Individually:

The data was extremely imbalanced — the 0:1 proportion was close to 0.95% for some, and so we focused much of our time on accounting for this, as this was a likely contributor to the overfitting of our model. The key is to resample in a way that improves the representation of underrepresented labels without significantly distorting the distribution of other labels. Here is our approach.

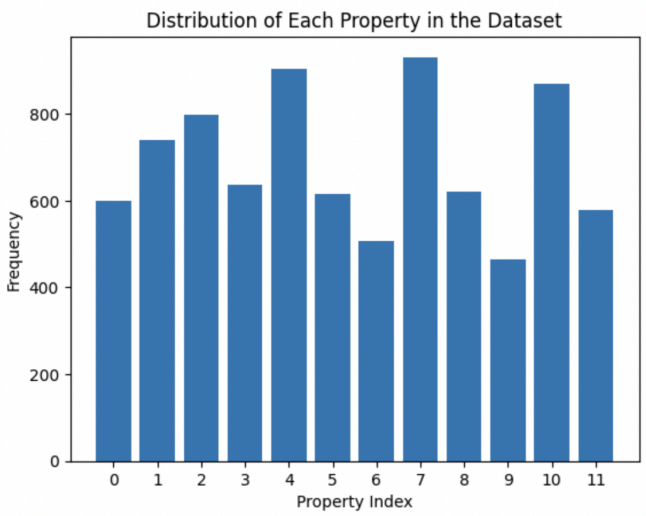
```
# Adding miniority class samples
new_train_dataset = torch.load("train_data.pt")
graph_to_add = []
new_property_counts = np.zeros(12)
count = 0

for graph in train_dataset:
    labels = graph.y
    # if labels[0, 6] == 1 and labels[0, 7] != 1 and labels[0, 4] != 1 and labels[0, 2] != 1:
    if labels[0, 6] == 1 and labels[0, 7] != 1:
        graph_to_add.extend([graph])
    if labels[0, 6] == 1 and labels[0, 7] != 1 and labels[0, 10] != 1:
        graph_to_add.extend([graph]*2)
    if labels[0, 3] == 1 and labels[0, 7] != 1:
        graph_to_add.extend([graph]*3)
    if labels[0, 1] == 1 and labels[0, 10] != 1:
        graph_to_add.extend([graph]*3)
    if labels[0, 8] == 1 and labels[0, 10] != 1 and labels[0, 4] != 1:
        graph_to_add.extend([graph]*4)


new_train_dataset.extend(graph_to_add)
```

Long Nguyen
Ming-Shih Wang

Given this is a multi-label classification problem, the distribution of the 12 labels in each data entry is dependent on each other; Which means that simply replicating the minority class would not produce a balanced distribution. Therefore, through trial and error, we found the best combination of data to replicate, which produced fairly balanced data as shown in the graphs below.



Left: Before oversampling



Right: After oversampling

Integrating SMILES and Molecular Data

Another issue we thought was the cause of overfitting and the "12-task" imbalance was the lack of data. To account for this, we tried adding more features using the *rdkit* to utilize SMILES and Molecular data.

```
💡 Click here to ask Blackbox to help you code faster
import molvs as mv
from rdkit import Chem
from rdkit.Chem import AllChem, rdMolDescriptors

def parent(smiles):
    st = mv.Standardizer()
    try:
        mols = st.charge_parent(Chem.MolFromSmiles(smiles))
        return Chem.MolToSmiles(mols)
    except:
        return "NaN"

def morgan_fp(smiles, radius=3, n_bits=8192):
    standardized_smiles = parent(smiles)
    if standardized_smiles != "NaN":
        mol = Chem.MolFromSmiles(standardized_smiles)
        fp = AllChem.GetMorganFingerprintAsBitVect(mol, radius, nBits=n_bits)
        return np.array(list(fp.ToBitString())).astype('int')
    else:
        return np.zeros(n_bits, dtype='int')

# Add fingerprint data to the datasets
for dataset in [train_dataset, valid_dataset]:
    for data in dataset:
        smiles = data.smiles
        fp_array = morgan_fp(smiles)
        fp_array = torch.tensor(fp_array, dtype=torch.float)
        # Expand fp_array to match the number of nodes in data.x
        num_nodes = data.x.size(0)
        fp_array_expanded = fp_array.repeat(num_nodes, 1)  # Repeat for each node
        data.x = torch.cat((data.x, fp_array_expanded), dim=1)
        print(data.x.shape)
✓ 0.0s
```

Long Nguyen
Ming-Shih Wang

```python
def smiles_to_fingerprints(smiles_list, radius=2, n_bits=1024):
    fingerprints = []
    for smiles in smiles_list:
        mol = Chem.MolFromSmiles(smiles)
        fp = AllChem.GetMorganFingerprintAsBitVect(mol, radius, nBits=n_bits)
        fingerprints.append(np.array(fp))
    return np.array(fingerprints)

# Apply to the augmented dataset
new_fingerprints = smiles_to_fingerprints([data.smiles for data in new_train_dataset])
for data, fp in zip(new_train_dataset, new_fingerprints):
    data.fp = torch.tensor(fp, dtype=torch.float)

valid_fingerprints = smiles_to_fingerprints([data.smiles for data in valid_dataset])
for data, fp in zip(valid_dataset, valid_fingerprints):
    data.fp = torch.tensor(fp, dtype=torch.float)

class MyDataLoader(DataLoader):
    def collate_fn(self, batch):
        batch = super(MyDataLoader, self).collate(batch)
        batch.fp = torch.stack([data.fp for data in batch.to_data_list()])
        return batch
```

By using RDKit and MolVS libraries, the code standardizes molecules from their SMILES representation and extracts their parent structure. Then, it generates Morgan fingerprints, a form of circular fingerprint that encapsulates the molecular structure. These fingerprints are integrated into our dataset by expanding them to match the graph representation of each molecule, thereby augmenting the original node features. This enrichment with detailed chemical information aims to provide a more comprehensive dataset, potentially enhancing the predictive accuracy of our models in tasks such as molecular property prediction. However, this caused more overfitting, quickly going from *68%* to *84%*, to *93%* within 3 epochs for the training ROC-AUC score — and leaving the validation score still to be around *75%*.

Weighted Loss Function

We also calculated the weights in each label class to account for data imbalance. The original loss calculation simply converts the two-dimensional toxicity label dataset into a huge single-dimension array while excluding all the nan values. This would not work if class weights were applied, we were experiencing dimension mismatches

```python
import numpy as np
# Assuming there are 12 properties, initialize an array to store counts for each property
property_counts = np.zeros(12)
# Iterate over the dataset to count each property
for graph in train_dataset:
    labels = graph.y
    for i in range(12):
        if not torch.isnan(labels[0, i]):
            # print(labels[0, i])
            property_counts[i] += labels[0, i]

# The weight for each class is inversely proportional to its frequency
class_weights = [1.0 / count if count > 0 else max(class_weights) for count in property_counts]
class_weights = torch.tensor(class_weights, dtype=torch.float)

# Handle the case where a class is not present in the dataset (to avoid division by zero)
class_weights[class_weights == float('inf')] = 0

print("Class weights:", class_weights)
```

✓ 0.5s

```
Class weights: tensor([0.0040, 0.0053, 0.0017, 0.0048, 0.0015, 0.0034, 0.0075, 0.0014, 0.0051,
        0.0035, 0.0014, 0.0036])
```

because of the missing values. To account for the dimension difference, because the labels are in a binary format that represents the absence of the property associated with the label, I replaced nan values with 0.

```
import torch.nn.functional as F

def weighted_binary_cross_entropy(output, target, weights=None):
    if weights is not None:
        # Apply weights to the loss
        loss = torch.nn.BCEWithLogitsLoss(pos_weight=weights)(output, target)
        return loss.mean()

    else:
        loss = F.binary_cross_entropy(output, target, reduction='none')

    return torch.mean(torch.sum(loss, dim=1))
✓ 0.0s
```

After class weights calculation, I applied them to a custom loss calculation function using the BCEWithLogitsLoss()function that works better with binary values.

Model Optimization and Fine-Tuning:

Our optimizations did not stop in preprocessing, as we implemented a random search for tuning combinations of layers and hyperparameters.

- Models: GraphConv, GATConv, Regular Neural Network, GNN, GCNConv, BatchNorm, ReLu
- Hyperparameters: dropout_rate, fingerprint_dim, num_classes, hidden_classes, num_rdkit_features

However, we continued to hit a plateau of around *78.98%* with these models and combinations of layers and hyperparameter tuning. In the end, we used the model shown above, with a combination of Graph Convolutional Layers, Batch Normalizations, Linears, and layers that utilize the fingerprint data. The most optimal hyperparameters were found to be:

hidden_channels = 32, dropout_rate = 0.6, fingerprint_dim = 8192, num_node_features = 9

```python
# A simple graph neural network model
import torch
from torch_geometric.nn import GraphConv, global_mean_pool as gap

import torch.nn.functional as F
from torch.nn import Linear, BatchNorm1d
from torch_geometric.nn import global_mean_pool

class GNN(torch.nn.Module):
    def __init__(self, hidden_channels, num_classes, dropout_rate, num_node_features=9, fingerprint_dim=8192):
        super(GNN, self).__init__()
        torch.manual_seed(12345)
        self.emb = AtomEncoder(hidden_channels=hidden_channels)
        self.conv1 = GraphConv(hidden_channels, hidden_channels)
        self.bn1 = BatchNorm1d(hidden_channels)
        self.conv2 = GraphConv(hidden_channels, hidden_channels)
        self.bn2 = BatchNorm1d(hidden_channels)
        self.lin = Linear(hidden_channels, num_classes)
        self.fp_lin = Linear(fingerprint_dim, hidden_channels)
        self.dropout_rate = dropout_rate

    def forward(self, batch):
        x, edge_index, batch_size = batch.x, batch.edge_index, batch.batch

        x = self.emb(x)  # Initial embedding

        x = self.conv1(x, edge_index)
        x = self.bn1(x)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout_rate, training=self.training)

        x = self.conv2(x, edge_index)
        x = self.bn2(x)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout_rate, training=self.training)


        x = gap(x, batch_size)  # Global Average Pooling
        x = self.lin(x)  # Linear layer for class prediction

        return x
```

Conclusion and Takeaways

All in all, this project was much harder than expected. Admittedly, if there was more time, we would spend more of it understanding the fundamentals of the complex data, as all the efforts of preprocessing with the imbalances, data concatenation, smiles, and molecular data, and all the model and hyperparameter optimization only resulted in a negligible increase in AUC-ROC score. Ultimately, we learned that the APIs of *Pytorch, Keras,* and *scikit-learn* make it accessible and quite easy to implement machine learning models and techniques, truly understanding the data, and the intricacies of how models work is crucial in achieving a high success rate.