Practical Work: Few-Shot Learning in Drug Discovery (Frequent Hitters)

Khaled Awadallah - K11945506

December 2024

1 Introduction

In the field of medicine, it is essential to constantly work on discovering new drugs in order to combat diseases and enhance human health. However, drug discovery is an extremely complex and time-consuming process that involves identifying molecules that can interact with specific molecular targets in the body to treat diseases [1]. The usual time that is needed to bring a drug to the market is about 10-15 years. The time and cost of any drug discovery project makes it hard for companies and researchers to sustain their efforts.

Recently, Machine Learning-based approaches have proven to be beneficial in shortening the time and cost of developing new drugs. Nowadays, Machine Learning [8] is used in all stages of drug development, especially in the initial stages of finding candidate molecules for further testing. However, Machine Learning methods need large amounts of training data which can be expensive or even unavailable in many drug discovery projects. Therefore, it is still important to develop methods that can deal with the low-data problem.

Few-Shot Learning [3] [4] [5] is an area of Machine Learning that deals with the problem of few available data samples. It tries to achieve reasonable predicting performance with only few data samples to learn from. In my work, I will use the "Frequent Hitters" method [6] , which is a simple few-shot learning method and will test this method on the MUV dataset and compare its performance with standard machine learning techniques such as Random Forest [7].

2 Methods

In standard Supervised Machine Learning [8], we rely on a dataset to train a model, and evaluate its performance by testing the predictions on a separate test set. This approach has proven to be powerful and can achieve state-of-the-art performance if there exists large sets of training samples that are drawn from the same distribution as the test set. However, in the field of drug discovery, training data samples can be scarce and therefore it can be that no large training set exists.

Few-Shot Learning can effectively tackle the problem of few available training samples by leveraging information obtained from pretraining on different but related tasks and then adapting the training process on the few data samples of the target task, using the information obtained from the pretraining process. In few-shot learning, the objective is to develop an effective predictive model using a small collection of molecules $X = \{x_1, \ldots, x_N\}$ paired with their corresponding measurements $y = \{y_1, \ldots, y_N\}$. Typically, these measurements are binary ($y_n \in \{0, 1\}$), representing inactive and active molecules, respectively. This dataset, denoted as Z = {X, y} , is referred to as the support set.

$$\hat{y} = g_w(m, Z)$$

In my work, I will use the Frequent Hitters model g^{FH}, which uses the usual few-shot learning process. This involves evaluating a specific query molecule m, paired with its label y, by leveraging information from the support set Z. However, the frequent hitters model g^{FH} neglects the support set Z:

$$\hat{y} = g_w^{FH}(m)$$

Practically, the frequent hitters model tries to predict the average activity of a molecule across all trained tasks in order to minimize the cross-entropy loss, since the model might predict both $y = 1$ and $y = 0$ during the training for the same molecule m considering that the molecule can be active in one task and inactive in another task.

3 Dataset and Pre-Processing

3.1 Dataset

The dataset I use in my work is the MUV dataset [9], which consists of 93087 molecules and 19 different tasks. Each entry in this labels matrix is either 0 (inactive) or 1 (active) and thus the same molecule can be active in one task and inactive in another task. In the last two columns of the labels matrix we have a unique ID and a unique SMILES representation for each molecule. SMILES strings are textual representations of the molecular structure of each molecule.

Then, for each molecule in the dataset, the molecule's ECFP fingerprints and descriptors are computed [10]. Fingerprints and descriptors are a representation of certain structural features of a molecule. In total, 2248 features (fingerprints and descriptors) are computed for each molecule. Thus, we get a molecules features matrix of the shape (93087, 2284).

```python
molecules = list(muv["smiles"])

# create mol objects
mols = list()

for smiles in molecules:
    mol = Chem.MolFromSmiles(smiles)
    mols.append(mol)

# ECFP fingerprints
ecfps = list()

for mol in mols:
    fp_sparseVec = rdFingerprintGenerator.GetCountFPs(
                     [mol], fpType=rdFingerprintGenerator.MorganFP
                    )[0]
    fp = np.zeros((0,), np.int8)  # Generate target pointer to fill
    DataStructs.ConvertToNumpyArray(fp_sparseVec, fp)

    ecfps.append(fp)

ecfps = np.array(ecfps)
ecfps.shape

# Descriptors
rdkit_descriptors = list()

for mol in mols:
    descrs = list()
    for descr in Descriptors._descList:
        _, descr_calc_fn = descr
        descrs.append(descr_calc_fn(mol))

    descrs = np.array(descrs)
    descrs = descrs[real_200_descr]
    rdkit_descriptors.append(descrs)

rdkit_descriptors = np.array(rdkit_descriptors)
rdkit_descriptors.shape
```

Code Block 1: Python Implementation for computing the features (fingerprints and descriptors) for each molecule.

3.2 Pre-Processing

During preprocessing, all NaN values in the label matrix (rows refer to molecules and columns refer to tasks) are ignored and the label matrix format is restructured such that we have 3 vectors: Molecule ID, Target ID, Label, where a triplet (mol_id, target_id, label) is one entry in the matrix. This structure makes it easier for the dataloader to retrieve the precomputed features using mol_id. Moreover, all features are standardized by subtracting the mean and scaling to unit variance.

```python
muv_matrix = np.empty(labels.shape, dtype=object)

for r in range(len(muv_matrix)):
    for c in range(len(muv_matrix[0])):
        label = labels[r, c]
        if label != 0.0 and label != 1.0:
            label = -1
        muv_matrix[r, c] = (r, c, int(label))

def filter(matrix):
    matrix_copy = matrix.copy()
    flattened = matrix_copy.reshape(-1)
    filtered = []
    for triplet in flattened:
        if triplet[2] != -1:
            filtered.append(triplet)

    return np.array(filtered)
```

Code Block 2: Python implementation for restructuring the labels matrix.

4 Metrics for Evaluation

In my work, I will use two metrics for evaluating the performance of the models: ROC AUC (Area Under the ROC Curve) [11] and DAUPRC (Area Under the Precision-Recall Curve).

4.1 ROC AUC

The ROC AUC metric measures the area under the Receiver Operating Characteristic (ROC) curve, which plots the true positive rate (sensitivity) against the false positive rate (specificity) at various classification thresholds. This score provides an aggregate measure of performance across all possible threshold values. It ranges from 0.5 (random guessing) to 1 (perfect classification).
To calculate this metric, I use the roc_auc_score function from the scikit-learn library. The ROC AUC score is a robust metric that reflects the model's ability to discriminate between positive and negative classes, where a higher ROC AUC indicates that the model has a strong ability to differentiate between the two classes.

4.2 DAUPRC

The DAUPRC metric is derived from the AUPRC (Area Under the Precision-Recall Curve), which plots precision (the ratio of true positives to all positive predictions) against recall (the ratio of true positives to all actual positives). The DAUPRC score subtracts the baseline AUPRC (random classifier performance) from the actual AUPRC, giving a clear view of how much better the model performs relative to random guessing.

```
def compute_dauprc_score(predictions, labels, target_ids):
    """
    Computes the AUC-PR score for each target separately and the mean AUC-PR score.
    """
    dauprcs = list()
    target_id_list = list()

    for target_idx in torch.unique(target_ids):
        rows = torch.where(target_ids == target_idx)
        preds = predictions[rows].detach()
        y = labels[rows].int()

        if torch.unique(y).shape[0] == 2:
            number_actives = y[y == 1].shape[0]
            number_inactives = y[y == 0].shape[0]
            number_total = number_actives + number_inactives

            random_clf_auprc = number_actives / number_total
            auprc = average_precision_score(
                y.numpy().flatten(), preds.numpy().flatten()
            )

            dauprc = auprc - random_clf_auprc
            dauprcs.append(dauprc)
            target_id_list.append(target_idx.item())
        else:
            dauprcs.append(np.nan)
            target_id_list.append(target_idx.item())

    return np.nanmean(dauprcs), dauprcs, target_id_list
```

Code Block 3: Implementation of a function that calculates the DAUPRC score.

5 Experiments

5.1 Random Forest

Random Forest is a common and effective machine learning algorithm that combines the output of multiple decision trees to reach a single prediction. This approach is particularly effective in handling overfitting and provides strong performance across a variety of tasks. In my work, I train a Random Forest classifier on 3 different tasks independently, identified by indices 13, 14, and 15. The results across these tasks are then averaged to assess overall model performance. To ensure reproducibility and robustness, I use five random seeds to initialize the random number generator.

For each task and random seed, I create a training and testing split by randomly selecting five positive samples and five negative samples from the dataset for training, and the remaining samples are used for testing. This design emphasizes the classifier's capability to learn effectively even with a limited training set. Before training, all features are standardized using the StandardScaler from scikit-learn to ensure that the Random Forest classifier operated on a uniform scale, which is crucial for optimal performance. The classifier is then trained on the training set and tested on the test set.

The performance of the Random Forest Classifier is then evaluated using the ROC AUC Score and the DAUPRC Score. For computing the ROC AUC Score, I use the roc_auc_score function from the scikit-learn library, and for computing the DAUPRC Score, I use the custom compute_dauprc_score function described earlier (*see Code Block 3*). The results of the experiments are aggregated and reported for each seed in the Results section, and the mean and standard deviation are computed across all seeds (*See Table 2*).

```
tasks = [13, 14, 15]
seeds = [0, 1, 2, 3, 4]
auc_results = {seed: [] for seed in seeds}
dauprc_results = {seed: [] for seed in seeds}

for seed in seeds:
    np.random.seed(seed)
    for task in tasks:
        data = filtered_test[filtered_test[:, 1] == task]
        X = features[data[:, 0]]
        y = data[:, 2]
        pos_indices = np.where(y == 1)[0]
        neg_indices = np.where(y == 0)[0]
        pos_train_indices = np.random.choice(pos_indices, size=5, replace=False)
        neg_train_indices = np.random.choice(neg_indices, size=5, replace=False)
        train_indices = np.concatenate([pos_train_indices, neg_train_indices])
        test_indices = [i for i in range(len(y)) if i not in train_indices]
        X_train, y_train = X[train_indices], y[train_indices]
        X_test, y_test = X[test_indices], y[test_indices]

        scaler = StandardScaler()
        scaler.fit(X_train)
        X_train = scaler.transform(X_train)
        X_test = scaler.transform(X_test)

        clf = RandomForestClassifier(random_state=seed)
        clf.fit(X_train, y_train)
        y_pred_proba = clf.predict_proba(X_test)[:, 1]
        predictions_tensor = torch.tensor(y_pred_proba)
        labels_tensor = torch.tensor(y_test)
        target_ids_tensor = torch.zeros_like(labels_tensor)

        auc_score = roc_auc_score(y_test, y_pred_proba)
        mean_dauprc, dauprcs, target_id_list = compute_dauprc_score(
            predictions_tensor,
            labels_tensor,
            target_ids_tensor)

        auc_results[seed].append(auc_score)
        dauprc_results[seed].append(mean_dauprc)

mean_auc_seeds, std_auc_seeds = [], []
mean_dauprc_seeds, std_dauprc_seeds = [], []
for seed in seeds:
    auc_scores = auc_results[seed]
    mean_auc = np.mean(auc_scores)
    mean_auc_seeds.append(mean_auc)
    print(f"Seed {seed}:")
    print(f"Mean roc_auc_score = {mean_auc:.4f}")

    dauprcs = dauprc_results[seed]
    mean_dauprc = np.nanmean(dauprcs)
    mean_dauprc_seeds.append(mean_dauprc)
    print(f"Mean dauprc_score = {mean_dauprc:.4f}\n")

print(f"\nMean AUC Score over all seeds: {np.mean(mean_auc_seeds):.4f}")
print(f"Standard Deviation over all seeds: {np.std(mean_auc_seeds):.4f}")
print(f"Mean DAUPRC Score over all seeds: {np.nanmean(mean_dauprc_seeds):.4f}")
print(f"Standard Deviation over all seeds: {np.nanstd(mean_dauprc_seeds):.4f}")
```

Code Block 4: Training procedure of a Random Forest Classifier

5.2 Frequent Hitters

To implement the frequent hitters method, the dataset is prepared by splitting it into training, validation, and test sets. 10 tasks are used for training, 3 tasks for validation, and 3 tasks for testing. All features are standardized using StandardScaler to ensure consistent scaling.

DataLoader: For the dataset handling, I define a MoleculeDataset class that extends the Dataset class from PyTorch. The __init__ method initializes an instance of the class with two inputs: features and labels. The __len__ method returns the number of samples in the dataset. The __getitem__ method retrieves a single feature-label pair from the dataset, given an index *idx*. This structure allows to prepare the data for the training procedure using PyTorch.

```
class MoleculeDataset(Dataset):
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        feature = self.features[idx]
        label = self.labels[idx, 2]
        return torch.tensor(feature, dtype=torch.float32),
        torch.tensor(label, dtype=torch.float32)
```

Code Block 5: MoleculeDataset class which extend the Dataset class from PyTorch.

After defining the MoleculeDataset class, I create a single dataloader for the training dataset, treating each molecule-task pair as an independent training sample. This means that the same molecule can appear multiple times in the dataset, associated with different tasks, and each instance is treated independently. Using this approach, we do not differentiate between which task a molecule is associated with, but instead, the model learns patterns from the entire dataset.

For validation and testing, the dataset is separated by the specific tasks, creating individual datasets and loaders for each task. This allows the model's performance to be evaluated on different tasks separately while keeping the training phase generalized across all molecule-task pairs. The task-specific separation ensures that testing and validation are done in a targeted manner, helping identify how well the model generalizes across different tasks.

```
batch_size = 8192

train_dataset = MoleculeDataset(features_train, labels_train)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True,
generator=torch.Generator().manual_seed(seed))

val_tasks = [10, 11, 12]
val_datasets = {}
val_loaders = {}
for task in val_tasks:
    val_datasets[task] = MoleculeDataset(features_val[labels_val[:,1] == task],
    labels_val[labels_val[:, 1] == task])
for task, dataset in val_datasets.items():
    val_loaders[task] = DataLoader(dataset=dataset, batch_size=batch_size, shuffle=False,
    generator=torch.Generator().manual_seed(seed))

test_tasks = [13, 14, 15]
test_datasets = {}
test_loaders = {}
for task in test_tasks:
    test_datasets[task] = MoleculeDataset(features_test[labels_test[:, 1] == task],
    labels_test[labels_test[:, 1] == task])
for task, dataset in test_datasets.items():
    test_loaders[task] = DataLoader(dataset=dataset, batch_size=batch_size, shuffle=False,
    generator=torch.Generator().manual_seed(seed))
```

Code Block 6: Create training, validation, testing datasets and loaders.

Neural Network's Architecture: In this part, I define a feed-forward neural network architecture, consisting of 4 fully connected layers with ReLU activation functions and dropout [12] for regularization. The dropout is set to 0.25 for the input and then it is set to 0.5. The input layer size is determined by the number of features (2248), and the output layer size is set to 1 for binary classification. The sizes of the three hidden layers are set to 64, 32, 16 respectively.

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hs1, hs2, hs3, output_size):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hs1)
        self.fc2 = nn.Linear(hs1, hs2)
        self.fc3 = nn.Linear(hs2, hs3)
        self.fc4 = nn.Linear(hs3, output_size)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
        self.input_dropout = nn.Dropout(0.25)
        self.dropout = nn.Dropout(0.5)
    def forward(self, x):
        x = self.input_dropout(x)
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.dropout(self.relu(self.fc2(x)))
        x = self.dropout(self.relu(self.fc3(x)))
        x = self.sigmoid(self.fc4(x))
        return x
```

Code Block 7: Feed-Forward Neural Network Architecture.

The model is then created, optimizing the parameters using the Adam optimizer and using the Binary Cross-Entropy Loss as the minimization criterion. For training, 5 random seeds are set to ensure reproducibility and robustness, and the training process involves iterating over 5 epochs, adjusting the model's weights based on the calculated gradients. Moreover, different combinations of hyperparameters and model architectures (learning rate, batch size, hidden layers, etc.) are tried to get the best possible results (*see Table 1*).

```
input_size = features_train.shape[1]
hs1 = 64
hs2 = 32
hs3 = 16
output_size = 1
num_epochs = 5
num_steps = len(train_loader)
learning_rate = 0.01
model = NeuralNet(input_size, hs1, hs2, hs3, output_size)
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

code Block 8: Model architecture and training parameters.

Hyperparameter	**Explored Values**
Number of hidden layers	1, 2, **3**, 4
Number of units per hidden layer	128-64-32, **64-32-16**, 32-16-8
Learning rate	0.0001, 0.001, **0.01**, 0.1
Optimizer	**Adam**
Batch size	512, 1024, 4096, **8192**
Activation function	**ReLU**, SELU
Input dropout	0.1, **0.25**
Dropout	0.3, **0.5**, 0.7

Table 1: Hyperparameters considered for model selection. The bold values represent the best configuration.

Training and validation: During the training phase, the model is set to the training mode and processes batches of features and labels from train_loader. For each batch, the model makes predictions, calculates the loss using binary cross-entropy, and updates the model parameters through back-propagation and gradient descent. After each training epoch, the model is switched to evaluation mode, where it is tested on validation datasets corresponding to different tasks. For each task, predictions and labels are collected, and performance is evaluated using ROC AUC Score and Mean DAUPRC Score.

```
for epoch in range(num_epochs):
    # Training
    model.train()
    for i, (features, labels) in enumerate(train_loader):
        labels = labels.unsqueeze(1)
        outputs = model(features)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if (i+1) % 10 == 0:
            print(f"Epoch {epoch+1}/{num_epochs}, Step {i+1}/{num_steps}, Loss = {loss}")
    print()

    # Validation
    model.eval()
    with torch.no_grad():
        for task, val_loader in val_loaders.items():
            all_labels = []
            all_preds = []
            for features, labels in val_loader:
                labels = labels.unsqueeze(1)
                outputs = model(features)
                all_preds.extend(outputs.numpy().flatten())
                all_labels.extend(labels.numpy().flatten())

            all_labels = np.array(all_labels, dtype=int)
            all_preds = np.array(all_preds)

            # roc_auc score
            roc_auc = roc_auc_score(all_labels, all_preds)

            # dauprc score
            all_labels_tensor = torch.tensor(all_labels)
            all_preds_tensor = torch.tensor(all_preds)
            target_ids_tensor = torch.zeros_like(all_labels_tensor)
            mean_dauprc, dauprcs, target_id_list = compute_dauprc_score(all_preds_tensor,
            all_labels_tensor, target_ids_tensor)

            print(f"Validation Task {task}, ROC AUC Score: {roc_auc:.4f})
            print(f"Mean DAUPRC Score: {mean_dauprc:.4f}")

    print()
```

Code Block 9: Training procedure for a single seed

Testing: During the testing phase, the trained model is evaluated on multiple test tasks. For each task, predictions and true labels are gathered from the test dataset. The model's performance is then evaluated using the ROC AUC score and the Mean DAUPRC score. Finally, the average ROC AUC and DAUPRC scores across all test tasks are computed, providing an overall measure of model performance on the test data, and the mean and standard deviation are computed across all seeds.

```
# Testing
mean_roc_auc_list = []
mean_dauprc_list = []

model.eval()
with torch.no_grad():
    for task, test_loader in test_loaders.items():
        all_preds = []
        all_labels = []
        for features, labels in test_loader:
            labels = labels.unsqueeze(1)
            outputs = model(features)
            all_preds.extend(outputs.numpy().flatten())
            all_labels.extend(labels.numpy().flatten())

        all_labels = np.array(all_labels, dtype=int)
        all_preds = np.array(all_preds)

        # ROC AUC Score
        roc_auc = roc_auc_score(all_labels, all_preds)

        # DAUPRC Score
        all_preds_tensor = torch.tensor(all_preds)
        all_labels_tensor = torch.tensor(all_labels)
        target_ids_tensor = torch.zeros_like(all_labels_tensor)
        mean_dauprc, dauprcs, target_id_list = compute_dauprc_score(all_preds_tensor,
        all_labels_tensor, target_ids_tensor)

        mean_roc_auc_list.append(roc_auc)
        mean_dauprc_list.append(mean_dauprc)

        print(f"Test Task {task}, ROC AUC Score: {roc_auc:.4f})
        print(f"Mean DAUPRC Score: {mean_dauprc:.4f}")

print(f"\n Mean ROC AUC Score across all test tasks: {np.mean(mean_roc_auc_list)}")
print(f"Mean DAUPRC Score across all test tasks: {np.mean(mean_dauprc_list)}")
```

Code Block 10: Testing the trained model for a single seed

6 Results

Seed	**Random Forest**		**Frequent Hitters**	
	ROC AUC	**DAUPRC**	**ROC AUC**	**DAUPRC**
0	0.7430	0.0472	0.6757	0.0018
1	0.6724	0.0063	0.6270	0.0008
2	0.7611	0.0241	0.6912	0.0018
3	0.5241	0.0157	0.6608	0.0014
4	0.7215	0.0156	0.6881	0.0020
Mean	**0.6844**	**0.0218**	**0.6686**	**0.0015**
Standard Deviation	**0.0855**	**0.0139**	**0.0234**	**0.0004**

Table 2: Comparison of performance results between Random Forest and Frequent Hitters models across 5 random seeds.

The performance of the two models, Random Forest and Frequent Hitters, is evaluated using the metrics ROC AUC and DAUPRC, and the results clearly indicate that the Random Forest model outperformed the Frequent Hitters model across all tasks and seeds (*See Table 2*). For the Random Forest model, the mean ROC AUC score across all seeds was 0.6844, with a standard deviation of 0.0855, while the mean DAUPRC score was 0.0218, with a standard deviation of 0.0139. On the other hand, the Frequent Hitters model showed a mean ROC AUC score of 0.6686, with a standard deviation of 0.0234, and a mean DAUPRC score of 0.0015, with a standard deviation of 0.0004. Despite the simplicity of the Frequent Hitters model, its performance was consistently inferior to that of the Random Forest model. This outcome falls short of expectations , particularly because the Frequent Hitters model was initially hypothesized to serve as a competitive baseline for low-data scenarios [6]. Although both models achieved a low DAUPRC score, the Random Forest model demonstrated better performance, even when trained on limited data. In contrast, the Frequent Hitters model's design, which does not utilize the support set, likely restricted its performance, particularly in scenarios requiring more nuanced decision boundaries. These findings highlight the limitations of the Frequent Hitters model.

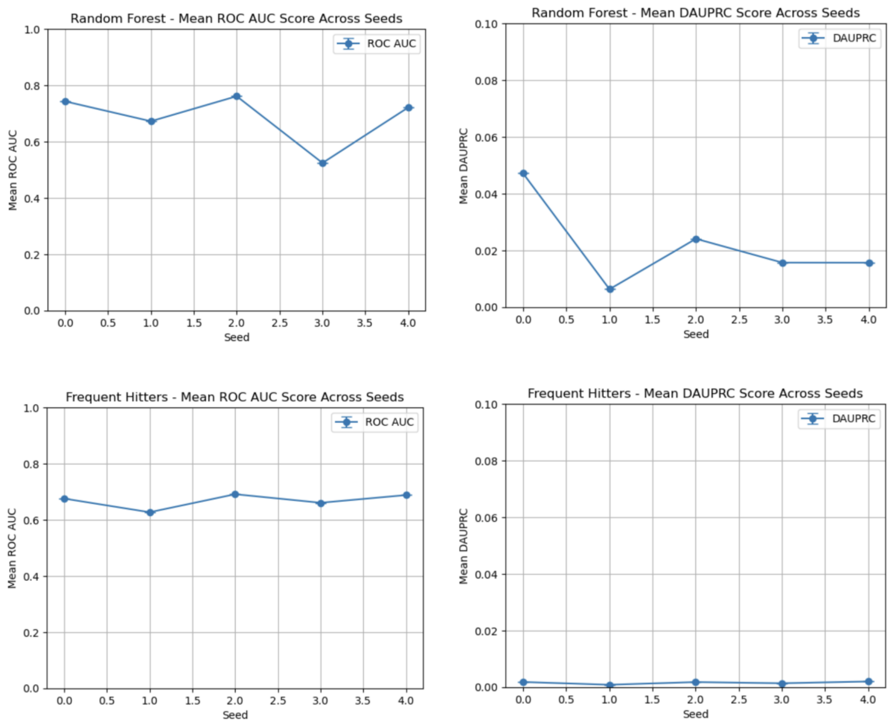

Figure 1: Performance comparison of Random Forest and Frequent Hitters models across all seeds

The performance plots (*see Fig. 1*) provide insight into the behavior of both the Random Forest and Frequent Hitters models across multiple seeds. In the case of Random Forest, the ROC AUC scores range between 0.6 and 0.8. This indicates that the model consistently achieves reasonable discriminative power when distinguishing active molecules from inactive ones. However, the DAUPRC scores for Random Forest are significantly low, with values ranging from 0.0005 to approximately 0.05, revealing a considerable weakness in the model's ability to predict the classes accurately.

In contrast, the Frequent Hitters model shows consistently lower ROC AUC scores compared to Random Forest, with values between 0.6 and 0.7 across seeds. These scores suggest that, while the model performs reasonably well, its ability to differentiate between active and inactive molecules is more limited. Like Random Forest, the Frequent Hitters model exhibits very low DAUPRC scores, indicating that the model fails to address the challenge of accurately predicting the correct class. When comparing the two models, it becomes evident that Random Forest outperforms Frequent Hitters on nearly every metric and task. The Random Forest model's higher ROC AUC scores demonstrate its better ability to capture complex patterns and relationships within the dataset, even in low-data scenarios. This advantage allows it to generalize better across seeds. By contrast, the Frequent Hitters model, while simpler and more stable, lacks the flexibility and power to adapt to the intricacies of the data. Its performance across seeds remains consistent, but it is consistently weaker than Random Forest, suggesting that its inability to leverage the support set limits its usefulness in this context.

The DAUPRC scores for both models, however, reveal a shared limitation: neither is particularly effective at addressing the imbalanced nature of the dataset, which is crucial in drug discovery applications. While Random Forest achieves marginally better DAUPRC scores than Frequent Hitters, its performance remains inadequate, highlighting the need for more sophisticated techniques. Frequent Hitters, by design, is not equipped to handle the complexity of minority class detection. In contrast, Random Forest's slightly better performance suggests that it could benefit from enhancements like class weighting, oversampling, or more advanced imbalanced learning techniques to boost its minority class sensitivity.

7 Conclusion

In my work, I explored the application of Few-Shot Learning methods, particularly the Frequent Hitters model, in the context of drug discovery using the MUV dataset. The performance of the Frequent Hitters model was benchmarked against a Random Forest classifier using two evaluation metrics: ROC AUC and DAUPRC. The findings revealed that the Random Forest model achieved better results, demonstrating its capability to handle low-data scenarios more effectively than the Frequent Hitters model. In contrast, the Frequent Hitters model, while conceptually simple, was outperformed due to its design limitation of neglecting the support set, which is critical for leveraging additional information in Few-Shot Learning. Future work should focus on enhancing the Frequent Hitters model by integrating support set information and exploring alternative Few-Shot Learning strategies that balance simplicity with performance. Furthermore, investigating advanced hyperparameter optimization techniques and incorporating domain-specific knowledge could potentially improve the effectiveness of both models in addressing the challenges of drug discovery.

References

[1] Arrowsmith, J. (2011). Phase ii failures: 2008-2010. Nature reviews drug discovery, 10(5).

[2] Wikipedia, Machine Learning. https://en.wikipedia.org/wiki/Machine_learning

[3] Bendre, N., Marín, H. T., and Najafirad, P. (2020). Learning from few samples: A survey.

[4] Wang, Y., Yao, Q., Kwok, J. T., and Ni, L. M. (2020). Generalizing from a few examples: A survey on few-shot learning.

[5] Schimunek, J., Friedrich, L., Kuhn, D., Rippmann, F., Hochreiter, S., and Klambauer, G. A generalized framework for embedding-based few-shot learning methods in drug discovery.

[6] Schimunek, J., Seidl, P., Friedrich, L., Kuhn, D., Rippmann, F., Hochreiter, S., and Klambauer, G. (2023). Context-enriched molecule representations improve few-shot druf discovery.

[7] Breiman, L. (2001). Random Forests. *Machine Learning* **45**, 5–32.

[8] Singh, A., Thakur, N., and Sharma, A. (2016). A review of supervised machine learning algorithms.

[9] Sebastian G. Rohrer and Knut Baumann. (2009). Maximum Unbiased Validation (MUV) data sets for virtual screening based on PubChem bioactivity data.

[10] Rogers, D. and Hahn, M. (2010). Extended-connectivity fingerprints. Journal of chemical information and modeling.

[11] Andrew P. Bradley. (1997). The use of the area under the ROC curve in the evaluation of machine learning algorithms.

[12] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting.