DRUG CHARACTER PREDICTION USING BLOOD BRAIN BARRIER PENETRATION DATASET

A project Report
submitted in the fulfilment of the requirements for the course
CSCI B-565 Data Mining

By

Durga Sai Sailesh Chodabattula

Sai Jagan Reddy Lakku

Raja Simha Reddy Allampati

Dhanusha Duraiyan

Under the supervision of

Dr. Yuzhen Ye

Associate Professor
Luddy School of Informatics, Computing and Engineering, Indiana University Bloomington

ABSTRACT

The blood-brain barrier (BBB) has proven to be a significant obstacle to drug delivery to the brain. The BBB in a healthy brain serves as a diffusion barrier that prevents most substances from passing from the bloodstream to the brain, allowing only tiny molecules to pass through. The BBB is interrupted in specific pathological states of diseases like stroke, diabetes, seizures, multiple sclerosis, Parkinson's disease, and Alzheimer's disease. As a result, breaking through the barrier has long been a problem in the development of medications that target the central nervous system. Using blood brain barrier penetration (BBBP) dataset which includes binary labels for over 2000 compounds on their permeability properties, the permeability of the drug for the test set were checked. This project proposes deep learning methods and neural networks to predict the BBB permeability based on the clinical phenotypes data. Different models like linear, lasso, Elastic Net, SVM, and Naive Bayes were used to train the data and to predict the performance of the test data. The results indicated that neural network approaches may considerably increase drug BBB penetration prediction accuracy, allowing researchers to minimize clinical trials and identify novel CNS medicines.

Keywords: BBB, BBBP, Deep learning methods, Neural networks, Gradient boosting, correlation

TABLE OF CONTENTS

1. INTRODUCTION

Brain disorders, such as central nervous system (CNS) issues and brain tumors, are among the most frequent, fatal, and undertreated ailments. Because the number of seniors and patients with CNS issues is increasing, global drug development for brain diseases will need to increase significantly during the next 20 years. However, as compared to other therapeutic sectors, the discovery of medications for brain ailments has the lowest success rates. The time it takes to develop CNS drugs is frequently significantly longer than the time it takes to develop non-CNS treatments. Clinical trials of CNS drugs are challenging due to the brain's complexity, adverse effects, and the blood-brain barrier's (BBB) impermeability.

CNS drug development is limited by a lack of adequate technologies for transporting drugs over the BBB, in addition to the complications of brain disorders. Small and macromolecules are being investigated as possible therapeutic agents for a wide range of brain diseases. The BBB can only be crossed by minuscule lipid-soluble molecules with a molecular weight of less than 400 Da; most macromolecules cannot pass through the brain endothelium. The BBB is a physiological barrier that stops 95% of chemicals from becoming medicines. The BBB filters the blood, including the solutes found in the blood going to the brain.

To overcome all these complications in the process of drug discovery and obstacles in caused by the BBB to the drugs, we use the Blood Brain Barrier Penetration Dataset (BBBP) which gives the information of whether a drug can get through the BBB’s impermeability.

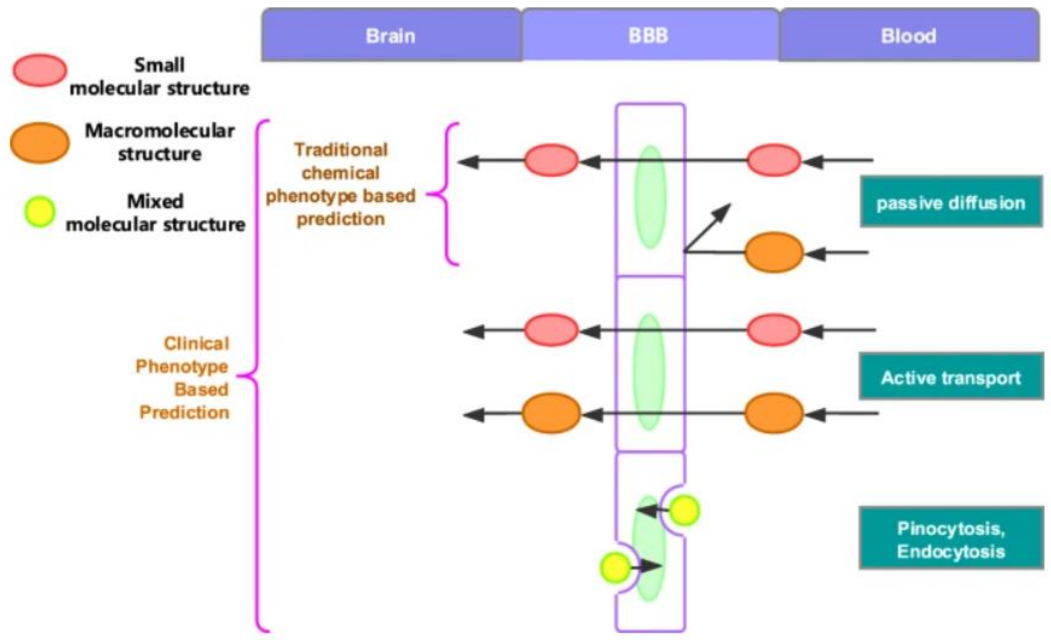

Brain
BBB
Blood
Small molecular structure
Macromolecular structure
Mixed molecular structure
Traditional chemical phenotype based prediction
Clinical Phenotype Based Prediction
passive diffusion
Active transport
Pinocytosis, Endocytosis

Figure 1. Mechanisms of drug transport through the BBB and the applicability of prediction methods (Miao et al., 2019).

2. EXPERIMENTAL INVESTIGATIONS:

2.1 Dataset

DeepChem provides a high-quality open-source toolchain that democratizes the application of deep-learning in drug discovery, materials science, quantum chemistry, and biology.

The BBBP dataset from the deepchem module was used. BBBP dataset contains 2053 items with four attributes :

- the index number from 1 to 2053 (“num”)
- the name of the compound (“name”)
- the penetrating or nonpenetrating properties (“p_np”)
- the SMILES string of the compound (“smiles”)

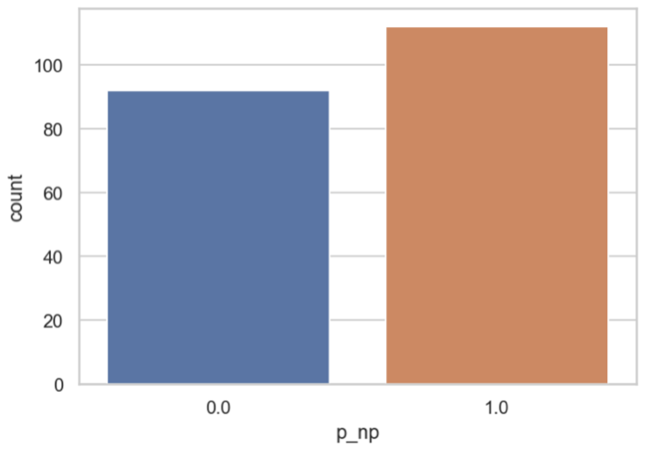


Figure 2. Class balance of p_np

2.1.1 Class balance

As the count of p_np for values of 0 and 1 are close to each other, we could say that the classes are balanced.

2.1.2 Data splitting and cross validation

The deepchem module primarily uses the scaffold spilt by default to perform the function. Scaffold split algorithm in MoleculeNet splits the given dataset into training, validation, and test data. It creates an unbalanced split which makes the prediction harder.

Initially, the molecular compounds are grouped into scaffold sets on the basis of the skeletal ring structures termed scaffolds then the compounds and scaffold sets are sorted in reverse order (largest to the smallest). Finally the dataset are split into training, validation, and test data in an 8:1:1 ratio from the top.

In scaffold splitting, the sets are sorted from the largest to the smallest. The test data will consist of compounds that are less related to others. This leads to the overfitting of the data which ends up in poor accuracy in both qualitative and quantitative prediction as the split is biased.

So to overcome this problem we used k – fold cross validation technique along with shuffling is done to increase the randomness in data for the train and test data. In k-fold, the training set is subdivided into k smaller subsets (other approaches are described below, but generally follow the same principles). For each of the k "folds," the following approach is used. A model is trained using the k-1 folds as training data, and the resulting model is verified using the remaining data (i.e., it is used as a test set to compute a performance measure such as accuracy). The values obtained are then averaged to give the performance metric of the different models we use in our project.

2.1.3 Conversion of the data to SMILES

The simplified molecular-input line-entry system (SMILES) is a specification in form of a line notation for describing the structure of chemical species using short ASCII strings. There are almost 1800 characteristics of each smile which needs to examined to determine which characteristics influence the penetration of a drug.

When using the raw .csv file, the raw data has to be converted to SMILES but while using the deepchem module this is simplified as the dataset already has the SMILES in it.

2.1.4 Conversion of SMILES to fingerprints

Deep learning models nearly usually require numerical arrays as inputs. We need to represent each molecule as one or more arrays of numbers if we wish to process molecules with them. Many (but not all) types of models need fixed-size inputs. This can be difficult for molecules because the amount of atoms in each molecule varies. If we wish to employ these models, we must somehow represent variable-sized molecules with fixed-sized arrays.

Fingerprints are intended to address these issues. A fingerprint is a fixed-length array in which distinct entries represent the existence of various properties in the molecule. If two molecules have similar fingerprints, it means they have many of the same traits and, as a result, will most likely have comparable chemistry.

DeepChem recognizes a type of fingerprint known as a "Extended Connectivity Fingerprint," or "ECFP" for short. They are also known as "circular fingerprints" at times. The ECFP method starts by categorizing atoms only based on their direct characteristics and bonds. Each distinct pattern is a distinguishing trait. For example, "carbon atom coupled to two hydrogens and two heavy atoms" is a feature, and for each molecule that includes that feature, a specific element of the fingerprint is set to 1. It then looks at bigger circular regions to identify new traits repeatedly. A higher level feature is formed when one specific feature is bound to two additional specific features, and the associated element is set for every molecule that includes it.

2.2 Models

The following models were used in the project:

- Linear Regression
- Lasso classifier
- SVC Model
- Naïve Bayes Classifier
- Neural Network
- Gradient Boosting on Linear regression and SVC model

2.2.1 Linear Regression

Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data. One variable is considered to be an explanatory variable, and the other is considered to be a dependent variable.

2.2.2 Lasso Classifier

Lasso (least absolute shrinkage and selection operator) is a regression analysis method that performs variable selection as well as regularization to improve the prediction accuracy and interpretability of the resulting statistical model. Lasso is a linear model with L1 prior as regularizer.

2.2.3 Support Vector Classifier (SVC)

The Linear Support Vector Classifier (SVC) method applies a linear kernel function to perform classification and it performs well with a large number of samples. If we compare it with the SVC model, the Linear SVC has additional parameters such as penalty normalization which applies `L1' or 'L2' and loss function. The kernel method cannot be changed in linear SVC, because it is based on the kernel linear method.

2.2.4 Naïve Bayes Classifier

It is a classification technique based on Bayes Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

2.2.5 Neural network

Artificial neural network models are behind many of the most complex applications of machine learning. Classification, regression problems, and sentiment analysis are some of the ways artificial neural networks are being leveraged today. As an emerging field, there are many different types of artificial neural networks. They differ for a variety of reasons, including: B. Complexity, network architecture, density, and data flow. However, different types share a common goal of modelling neuronal behaviour and replicating neuronal behaviour to improve machine learning.

2.2.6 Gradient Boosting on Linear regression and SVC model

Gradient boosting is a machine learning technique used especially for regression and classification tasks. This produces a predictive model in the form of an ensemble of weak predictive models, which is usually a decision tree. If the decision tree is a weak learner, the resulting algorithm is called a gradient boosting tree. Usually better than Random Forest. The gradient boosting tree model is built in stages like any other boosting method but generalizes the other methods by allowing optimization of the differentiable loss function.

3. RESULTS

3.1 Linear Regression

```
              precision    recall  f1-score   support

         0.0       0.77      1.00      0.87        71
         1.0       1.00      0.84      0.91       133

    accuracy                           0.90       204
   macro avg       0.89      0.92      0.89       204
weighted avg       0.92      0.90      0.90       204
```

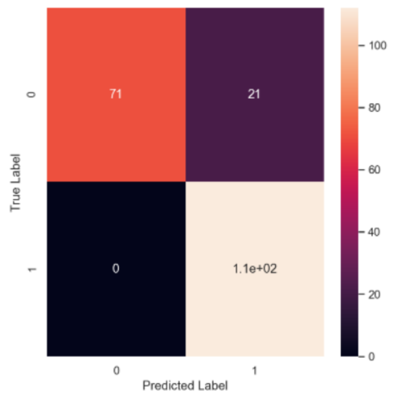


Figure 3

Overall, the accuracy for the model is 0.90. From the confusion matrix we can see that false negatives are not found.

3.2 Lasso Classifier

```
              precision    recall  f1-score   support

         0.0       0.76      0.96      0.85        73
         1.0       0.97      0.83      0.90       131

    accuracy                           0.88       204
   macro avg       0.87      0.90      0.87       204
weighted avg       0.90      0.88      0.88       204
```

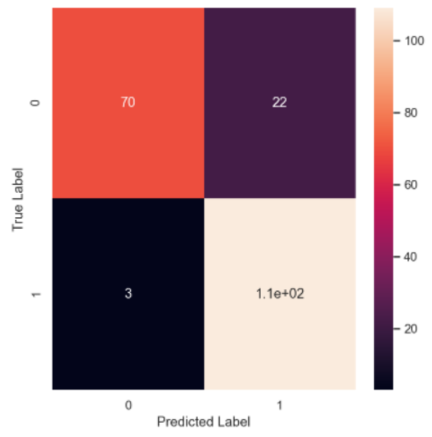


Figure 4

From the scores, we can see that precision for class 1.0 is high and yet the recall for it lower than class 0 but the F1 score is higher. Overall, the accuracy for the model is 0.88. From the confusion matrix we can see that false negatives are low as 3.

3.3 Support Vector Classifier (SVC)

```
              precision    recall  f1-score   support

         0.0       0.72      1.00      0.84        66
         1.0       1.00      0.81      0.90       138

    accuracy                           0.87       204
   macro avg       0.86      0.91      0.87       204
weighted avg       0.91      0.87      0.88       204
```

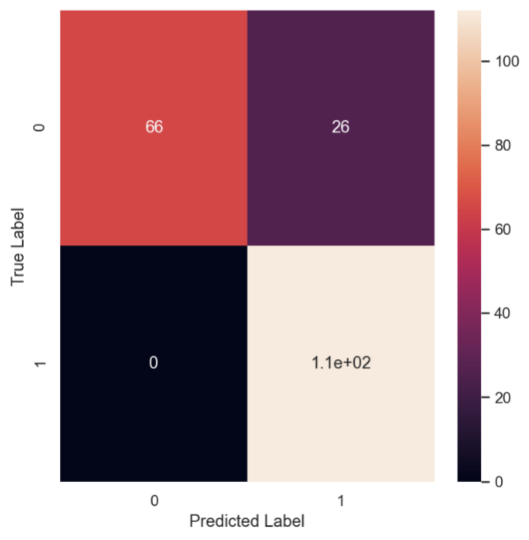


Figure 5

The confusion matrix shows that the model had no false negative results and the overall accuracy for the model is 0.87.

3.4 Naïve Bayes Classifier

```
              precision    recall  f1-score   support

         0.0       0.39      0.44      0.41        82
         1.0       0.59      0.54      0.56       122

    accuracy                           0.50       204
   macro avg       0.49      0.49      0.49       204
weighted avg       0.51      0.50      0.50       204
```

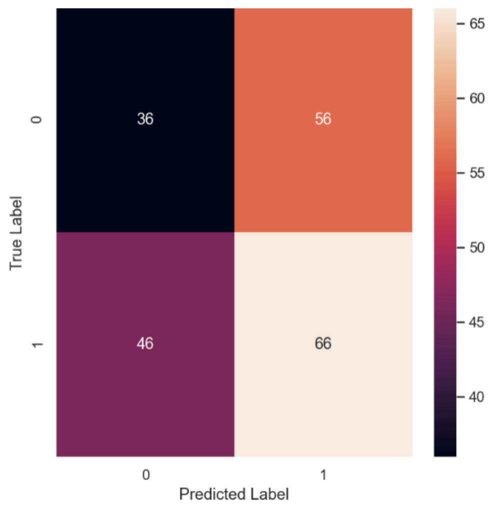
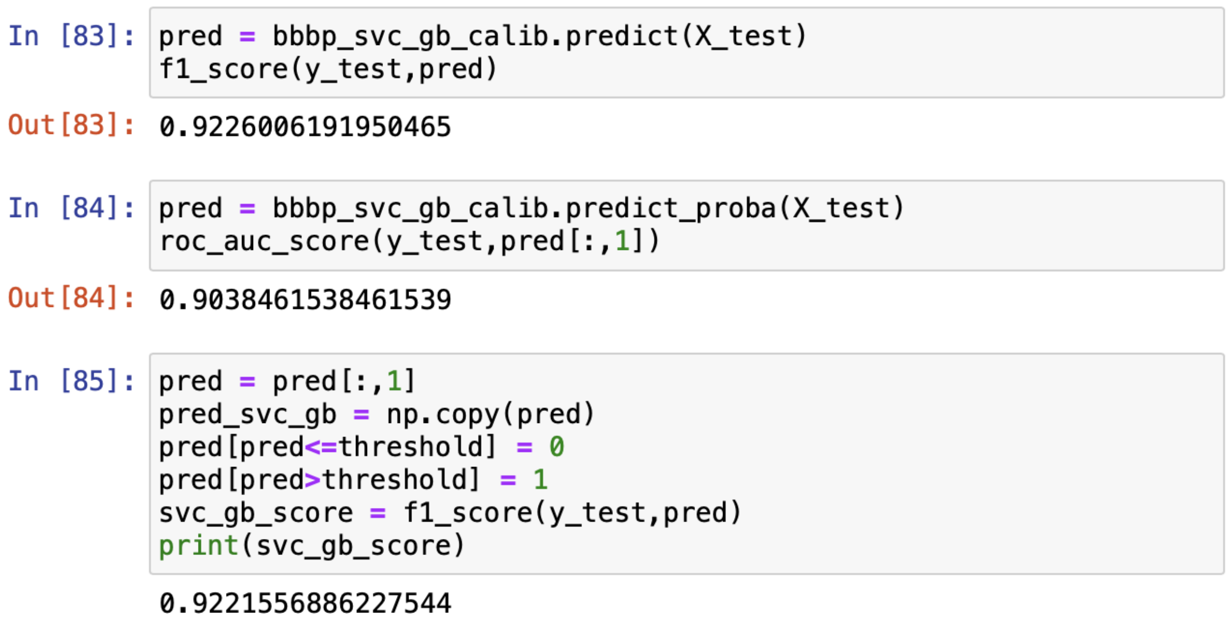


Figure 6

The naïve bayes classifier doesn’t have a good performance as the accuracy comes only upto 50% and it has high counts of both false positives and false negatives which is also reflected in the poor F1 scores.

3.5 Neural network



```
In [83]: pred = bbbp_svc_gb_calib.predict(X_test)
         f1_score(y_test,pred)

Out[83]: 0.9226006191950465

In [84]: pred = bbbp_svc_gb_calib.predict_proba(X_test)
         roc_auc_score(y_test,pred[:,1])

Out[84]: 0.9038461538461539

In [85]: pred = pred[:,1]
         pred_svc_gb = np.copy(pred)
         pred[pred<=threshold] = 0
         pred[pred>threshold] = 1
         svc_gb_score = f1_score(y_test,pred)
         print(svc_gb_score)

         0.9221556886227544
```

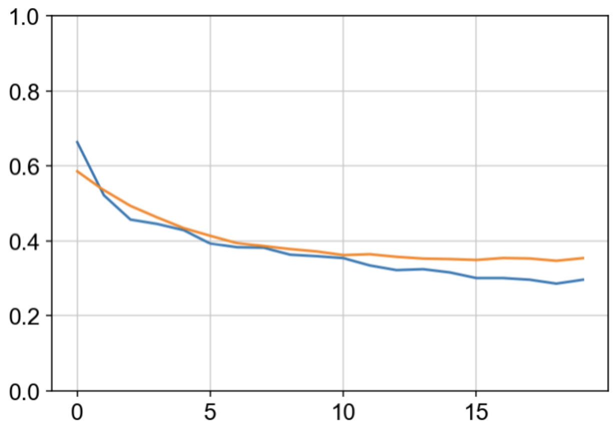
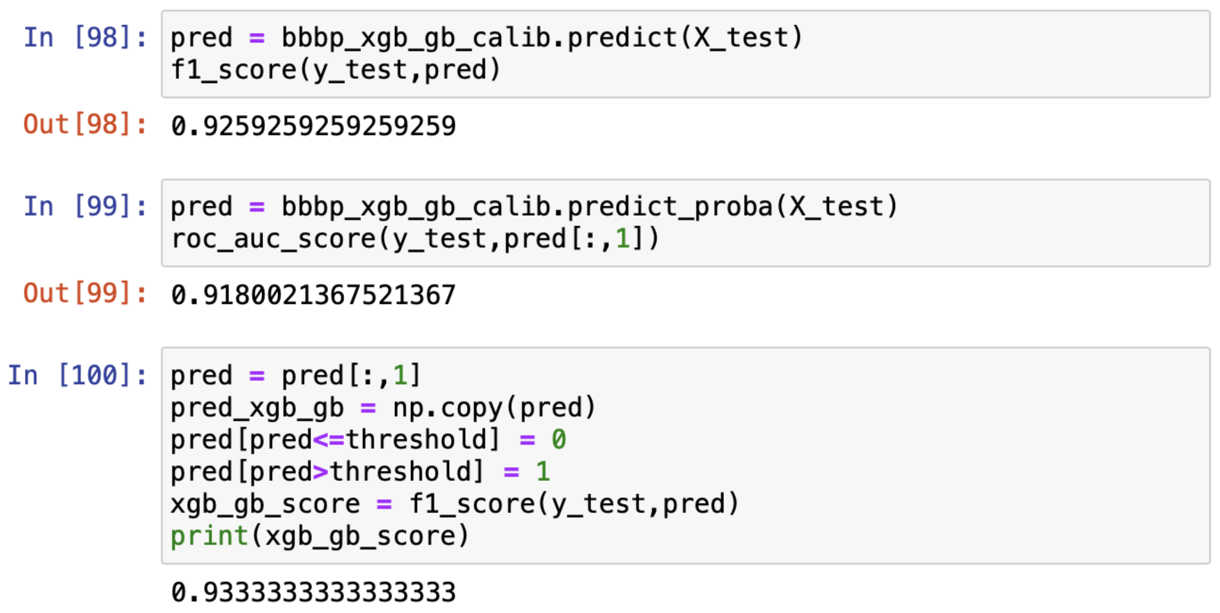


Figure 7

The F1 score for the neural networks is 0.92 which is pretty good. The test and train loss are depicted in the graph.

3.6 Gradient Boosting on Linear regression and SVC model

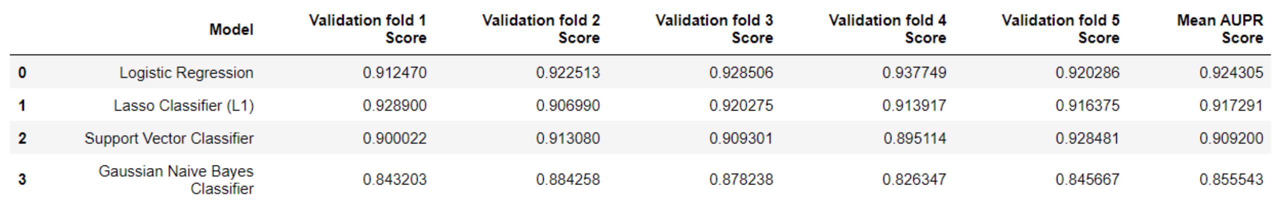


```
In [98]:  pred = bbbp_xgb_gb_calib.predict(X_test)
           f1_score(y_test,pred)

Out[98]:  0.9259259259259259

In [99]:  pred = bbbp_xgb_gb_calib.predict_proba(X_test)
           roc_auc_score(y_test,pred[:,1])

Out[99]:  0.9180021367521367

In [100]: pred = pred[:,1]
           pred_xgb_gb = np.copy(pred)
           pred[pred<=threshold] = 0
           pred[pred>threshold] = 1
           xgb_gb_score = f1_score(y_test,pred)
           print(xgb_gb_score)

           0.9333333333333333
```

Figure 8

Gradient boosting on linear regression and SVC models have elevated F1 score and accuracy. Also they have high roc_auc score which is better.

3.7 Overall performance of all the models



	Model	Validation fold 1 Score	Validation fold 2 Score	Validation fold 3 Score	Validation fold 4 Score	Validation fold 5 Score	Mean AUPR Score
0	Logistic Regression	0.912470	0.922513	0.928506	0.937749	0.920286	0.924305
1	Lasso Classifier (L1)	0.928900	0.906990	0.920275	0.913917	0.916375	0.917291
2	Support Vector Classifier	0.900022	0.913080	0.909301	0.895114	0.928481	0.909200
3	Gaussian Naive Bayes Classifier	0.843203	0.884258	0.878238	0.826347	0.845667	0.855543

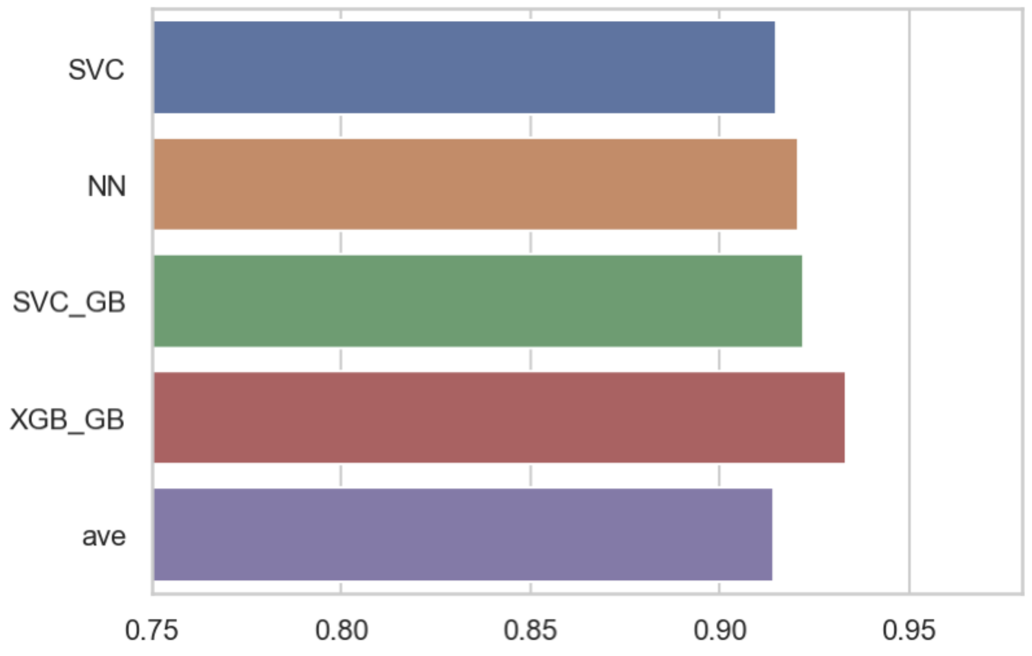
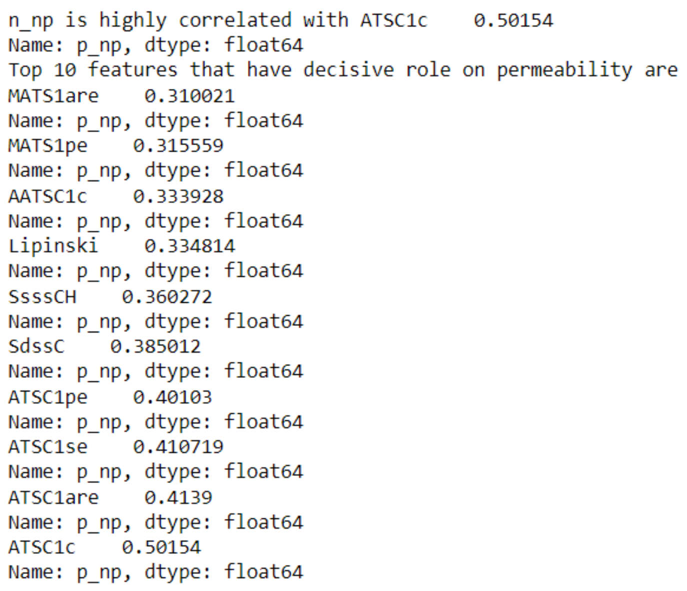


Figure 9

The overall performance of the all the models show that the gradient boosting performed better than all the other models tested followed by neural networks.

3.8 Correlation



```
n_np is highly correlated with ATSC1c    0.50154
Name: p_np, dtype: float64
Top 10 features that have decisive role on permeability are
MATS1are    0.310021
Name: p_np, dtype: float64
MATS1pe    0.315559
Name: p_np, dtype: float64
AATSC1c    0.333928
Name: p_np, dtype: float64
Lipinski    0.334814
Name: p_np, dtype: float64
SsssCH    0.360272
Name: p_np, dtype: float64
SdssC    0.385012
Name: p_np, dtype: float64
ATSC1pe    0.40103
Name: p_np, dtype: float64
ATSC1se    0.410719
Name: p_np, dtype: float64
ATSC1are    0.4139
Name: p_np, dtype: float64
ATSC1c    0.50154
Name: p_np, dtype: float64
```

Figure 10

4. DISCUSSION

4.1 Comparison of different models and neural networks

We used the train dataset to train the 5 ML models and incorporated a 5-fold cross validation (5-CV) in the process. We conducted the model predictions on the unseen Test data and displayed the results after training models and evaluated performance with 5-CV.
All models except Naïve bayes have very low False Negative score, but almost all models have a little high False Positive score. The classification reports show that even though accuracy of logistic regression are very high, these models have trouble classifying non-penetratable targets. This could be because there is more penetratable class data than non-p class data.

One of the reasons why Logistic Regression outperforms other models is because we did not include any regularization values - C=1 for logistic. So, by doing so, the model considers all features as part of its learning process and does not completely shrink/remove any feature weights, which is important in this biological problem statement where each feature has a meaningful contribution to finding the target values and thus including all of them helps the model get better results.

Furthermore, the main reason Gaussian Naive Bayes performs poorly (low accuracy, very low recall, precision, and high FN, FP) is due to its main idea of considering features to be independent of each other - this assumption will not work in many cases, particularly in biological problems where each molecular description and chemical feature is important to understanding the pattern in data.
Usually, the neural networks have better results than gradient boosted models. Here from figure 9, we can see that gradient boosted models work better for the dataset than the neural networks. This could be because of the number of datapoints available. The datapoints might not be sufficient for the neural network to work at its best.

4.2 Significance of the study

It is estimated that more than 98 percent of small organic molecules (drugs) will not penetrate the BBB. The time taken to develop drugs for CNS related diseases can be reduced significantly, as we can eliminate the possibility of a chemical being useful as a drug which can cross the blood brain barrier. This minimizes the number of clinical trials to be conducted which saves resources, time and is cost effective.

4.3 Cost of false positives and false negatives

When a drug is predicted that it can penetrate the barrier but actually it doesn’t, it is a false positive. This costs us a lot of time and clinical trials because when a chemical is considered as predictable, further research and work is done in developing the drug and when we figure out that it is actually not penetrable, efforts are gone for no use.

Similarly when a drug is predicted negative and is actually positive, it is false negative. False negatives doesn’t cost more because as a result we just lose one chemical which could have potentially become a drug.

Efforts and investment wise, false positives are costly. At the expense of lives being lost due to a disease, false negatives are costly.

4.4 Correlation

When correlation was calculated between the variables it was found that n_np is highly correlated with ATSC1c with the value of 0.50154. Also, the top 10 features which are decisive on permeability was found as seen in figure 10. This could infer that these features could influence a chemical being able to penetrate or not penetrate BBB.

5. REFERENCES

- Abbott, N. Joan, N. Joan Abbott, Adjanie A. K. Patabendige, Diana E. M. Dolman, Siti R. Yusof, and David J. Begley. 2010. “Structure and Function of the Blood–brain Barrier.” Neurobiology of Disease. https://doi.org/10.1016/j.nbd.2009.07.030.

- Banks, William A. 2009. “Characteristics of Compounds That Cross the Blood-Brain Barrier.” BMC Neurology 9 Suppl 1 (June): S3.

- Geldenhuys, Werner J., Afroz S. Mohammad, Chris E. Adkins, and Paul R. Lockman. 2015. “Molecular Determinants of Blood–brain Barrier Permeation.” Therapeutic Delivery 6 (8): 961–71.

- Guerra, Angela, Juan A. Páez, and Nuria E. Campillo. 2008. “Artificial Neural Networks in ADMET Modeling: Prediction of Blood-Brain Barrier Permeation.” QSAR & Combinatorial Science 27 (5): 586–94.

- Miao, Rui, Liang-Yong Xia, Hao-Heng Chen, Hai-Hui Huang, and Yong Liang. 2019. “Improved Classification of Blood-Brain-Barrier Drugs Using Deep Learning.” Scientific Reports 9 (1): 8802.

- MoleculeNet. Available online: https://moleculenet.org/ (accessed on 29 March 2022).

- Sakiyama, Hiroshi, Motohisa Fukuda, and Takashi Okuno. 2021. “Prediction of Blood-Brain Barrier Penetration (BBBP) Based on Molecular Descriptors of the Free-Form and In-Blood-Form Datasets.” Molecules 26 (24). https://doi.org/10.3390/molecules26247428.

- Weininger, David. 1988. “SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules.” Journal of Chemical Information and Computer Sciences 28 (1): 31–36.

- Wu, Zhenqin, Bharath Ramsundar, Evan N. Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S. Pappu, Karl Leswing, and Vijay Pande. 2018. “MoleculeNet: A Benchmark for Molecular Machine Learning.” Chemical Science 9 (2): 513–30.

- Along with these, other sites were referred for missing information and definitions that was required to carry on with the work.

In [153…
```
import pandas as pd

bbbp_df = pd.read_csv('/Users/jagan/Downloads/BBBP_Descriptors/BBBP_df_revi
```

In [154…
```
bbbp_df.head()
```

Out[154…

	num	name	p_np	smiles
0	1	Propanolol	1	[Cl].CC(C)NCC(O)COc1cccc2ccccc12
1	2	Terbutylchlorambucil	1	C(=O)(OC(C)(C)C)CCCc1ccc(cc1)N(CCCl)CCCl
2	3	40730	1	c12c3c(N4CCN(C)CC4)c(F)cc1c(c(C(O)=O)cn2C(C)CO...
3	4	24	1	C1CCN(CC1)Cc1cccc(c1)OCCCNC(=O)C
4	5	cloxacillin	1	Cc1onc(c2ccccc2Cl)c1C(=O)N[C@H]3[C@H]4SC(C)(C)...

In [155…
```
for col in bbbp_df.columns:
    print(col)
```

```
num
name
p_np
smiles
```

In [144…
```
bbbp_df.shape
```

Out[144…
```
(2039, 4)
```

In [156…
```
column_to_move = bbbp_df.pop('p_np')


bbbp_df.insert(3, 'p_np', column_to_move)
```

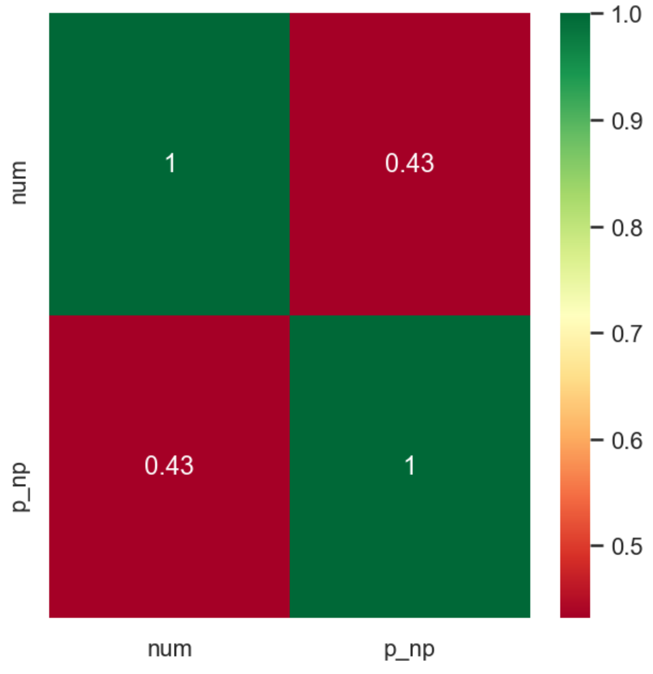
In [158…
```
bbbp_df.head()
```

Out[158…

	num	name	smiles	p_np
0	1	Propanolol	[Cl].CC(C)NCC(O)COc1cccc2ccccc12	1
1	2	Terbutylchlorambucil	C(=O)(OC(C)(C)C)CCCc1ccc(cc1)N(CCCl)CCCl	1
2	3	40730	c12c3c(N4CCN(C)CC4)c(F)cc1c(c(C(O)=O)cn2C(C)CO...	1
3	4	24	C1CCN(CC1)Cc1cccc(c1)OCCCNC(=O)C	1
4	5	cloxacillin	Cc1onc(c2ccccc2Cl)c1C(=O)N[C@H]3[C@H]4SC(C)(C)...	1

In [166…

```
X = bbbp_df.iloc[:,0:2]  #independent columns
y = bbbp_df.iloc[:,-1]    #target column i.e price range
#get correlations of each features in dataset
corrmat = bbbp_df.corr()
top_corr_features = corrmat.index
plt.figure(figsize=(5,5))
#plot heat map
g=sns.heatmap(bbbp_df[top_corr_features].corr(),annot=True,cmap="RdYlGn")
```



In [107…

```
import deepchem as dc
from deepchem.molnet.load_function.molnet_loader import TransformerGenerat
from deepchem.data import Dataset
```

In [108…

```
bbbp_data = dc.molnet.load_bbbp(splitter='scaffold')
```

```
[00:24:23] Explicit valence for atom # 1 N, 4, is greater than permitted
Failed to featurize datapoint 59, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:23] WARNING: not removing hydrogen atom without neighbors
[00:24:23] Explicit valence for atom # 6 N, 4, is greater than permitted
Failed to featurize datapoint 61, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:23] WARNING: not removing hydrogen atom without neighbors
[00:24:24] WARNING: not removing hydrogen atom without neighbors
[00:24:24] WARNING: not removing hydrogen atom without neighbors
[00:24:24] WARNING: not removing hydrogen atom without neighbors
[00:24:24] WARNING: not removing hydrogen atom without neighbors
[00:24:24] WARNING: not removing hydrogen atom without neighbors
```

```
[00:24:25] Explicit valence for atom # 6 N, 4, is greater than permitted
Failed to featurize datapoint 391, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:25] WARNING: not removing hydrogen atom without neighbors
[00:24:25] WARNING: not removing hydrogen atom without neighbors
[00:24:26] WARNING: not removing hydrogen atom without neighbors
[00:24:27] WARNING: not removing hydrogen atom without neighbors
[00:24:27] Explicit valence for atom # 11 N, 4, is greater than permitted
Failed to featurize datapoint 614, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:27] Explicit valence for atom # 12 N, 4, is greater than permitted
Failed to featurize datapoint 642, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:27] Explicit valence for atom # 5 N, 4, is greater than permitted
Failed to featurize datapoint 645, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:27] Explicit valence for atom # 5 N, 4, is greater than permitted
Failed to featurize datapoint 646, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:27] Explicit valence for atom # 5 N, 4, is greater than permitted
Failed to featurize datapoint 647, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:27] Explicit valence for atom # 5 N, 4, is greater than permitted
Failed to featurize datapoint 648, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:27] Explicit valence for atom # 5 N, 4, is greater than permitted
Failed to featurize datapoint 649, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:27] WARNING: not removing hydrogen atom without neighbors
```

```
[00:24:27] WARNING: not removing hydrogen atom without neighbors
[00:24:27] Explicit valence for atom # 5 N, 4, is greater than permitted
Failed to featurize datapoint 685, None. Appending empty array
Exception message: Python argument types in
    rdkit.Chem.rdmolfiles.CanonicalRankAtoms(NoneType)
did not match C++ signature:
    CanonicalRankAtoms(RDKit::ROMol mol, bool breakTies=True, bool includeC
hirality=True, bool includeIsotopes=True)
[00:24:28] WARNING: not removing hydrogen atom without neighbors
[00:24:29] WARNING: not removing hydrogen atom without neighbors
[00:24:30] WARNING: not removing hydrogen atom without neighbors
[00:24:30] WARNING: not removing hydrogen atom without neighbors
[00:24:30] WARNING: not removing hydrogen atom without neighbors
[00:24:31] WARNING: not removing hydrogen atom without neighbors
[00:24:31] WARNING: not removing hydrogen atom without neighbors
[00:24:32] WARNING: not removing hydrogen atom without neighbors
[00:24:32] WARNING: not removing hydrogen atom without neighbors
[00:24:32] WARNING: not removing hydrogen atom without neighbors
[00:24:32] WARNING: not removing hydrogen atom without neighbors
[00:24:32] WARNING: not removing hydrogen atom without neighbors
[00:24:32] WARNING: not removing hydrogen atom without neighbors
[00:24:32] WARNING: not removing hydrogen atom without neighbors
[00:24:33] WARNING: not removing hydrogen atom without neighbors
[00:24:33] WARNING: not removing hydrogen atom without neighbors
[00:24:33] WARNING: not removing hydrogen atom without neighbors
[00:24:34] WARNING: not removing hydrogen atom without neighbors
[00:24:35] WARNING: not removing hydrogen atom without neighbors
[00:24:35] WARNING: not removing hydrogen atom without neighbors
[00:24:35] WARNING: not removing hydrogen atom without neighbors
[00:24:35] WARNING: not removing hydrogen atom without neighbors
[00:24:36] WARNING: not removing hydrogen atom without neighbors
[00:24:36] WARNING: not removing hydrogen atom without neighbors
[00:24:36] WARNING: not removing hydrogen atom without neighbors
[00:24:37] WARNING: not removing hydrogen atom without neighbors
[00:24:38] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:42] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
```

```
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
[00:24:43] WARNING: not removing hydrogen atom without neighbors
```

In [109…
```
len(bbbp_data[1][1].ids)
```

Out[109…
```
204
```

In [110…
```
bbbp_data[1][1].ids
```

Out[110…
```
array(['[H+].C2=C1C(OC(=NC1=CC=C2Cl)NCC)(C3=CC=CC=C3)C.[Cl-]',
        'C1=CC=CC2=C1C(C3=C(N(C)C2=O)C=CC=C3)OCC',
        'C1=C(Cl)C=CC3=C1C(C2=CC=CC=C2)SC(=N3)NCC',
        'C1=CC=CC3=C1C4=C(C2=C(C=CC=C2)N3C)CCN(CC4)C',
        'C3=C(OC2C1(NC(=O)C(O1)C)CCCC2)C=CC=C3Cl',
        '[C@@H]1([C@@H]([C@H]2C=C[C@@H]1C2)C(N(CC)CC)=O)C(N(CC)CC)=O',
        'C1=CC=CC=C1CN(C2CC2)C(OCC)=O',
        'C3=C(N2CCN(CC(COC1=CC(=C(OC)C(=C1)OC)OC)O)CC2)C(=CC=C3)OC',
        'C1=C(C2=C(C=C1)OCCO2)N3CCNCC3',
        'C1=C(C2=C(C=C1)C=CO2)N5CCN(CC3=CC=C([NH]3)C4=CC=C(C=C4)F)CC5',
        'C1=CC(=CC3=C1\\C(C2=CC=CC=C2N3)=C/CN(CC)CC)Cl',
        'C2=C(N1N=C(C)CC1=O)C=CC=C2',
        '[C@H]2(N(C1=CC=C(C(=C1)Cl)Cl)C(CC)=O)[C@@H](CCC2)N(C)C',
        'C(N1C(CCC1)=O)C(NNC(CN2C(CCC2)=O)=O)=O',
        'C1=C(C(F)(F)F)C=CC3=C1N(C2=C(C=CC=C2)S3)CCCN5CCC(C(C4=CC=C(F)C=C4)=
O)CC5',
        'C1=CC=C3C(=C1OC(C2=CC=CS2)CCNC)C=CC=C3',
        'C1=CC=CC=C1C2(C(CN(C2=O)CC)CCN3CCOCC3)C4=CC=CC=C4',
        'C4=C(CCNC(C12CC3CC(C1)CC(C2)C3)=O)C=CC(=C4O)O',
        '[H+].C1=C(OC)C(=CC2=C1C(=O)C(C2)CC3CCN(CC3)CC4=CC=CC=C4)OC.[Cl-]',
        '[C@]4([C@@]3([C@H]([C@H]2[C@@H]([C@@]1(C(=CC(=O)C=C1)CC2)C)[C@H](C3
)O)CCC4)C)(OC(CCC)=O)C(C)=O',
        'C1=CC=CC2=C1C3(C4=C(CC2N3)C=CC=C4)C',
        'C1=C(N=C2[N]1C(=C(CC)C(=N2)OC)C)C(C3=CC=CC=C3)=O',
        'O=C2N1C(NC(=O)C1)CC2',
        'C1=CC=CC3=C1N(C2=C(C=CC=C2)S3)CC4(CN(C)CC4)C',
        'C4=C(C(C1=CC=C(C=C1)F)CCCN3CCN(CCNC2=CC=CC=C2)CC3)C=CC(=C4)F',
        '[C@]23([C@@H]1[C@@H](C(=CO[C@H]1OC(=O)CC(C)C)COC(=O)CC(C)C)C[C@@H]2
OC(=O)C)OC3',
        '[C@@H]1(NCCCC1)[C@@H]2OC(OC2)(C3=CC=CC=C3)C4=CC=CC=C4',
        'C3=C(C1N2C(=NC1)NCC2)C=CC=C3', '[C@@H]3(C1=CC=CC=C1)CN2CCSC2=N3',
        '[C@]4([C@@]3([C@H]([C@H]2[C@]([C@@]1(C(=CC(=O)C=C1)CC2)C)(F)[C@H](C
3)O)C[C@H]4C)C)(OC(C5=CC=CO5)=O)C(COC(C)=O)=O',
        'C2=C(CC1=CN=C[NH]1)C(=C(C=C2)C)C',
        'C3=C(C1=C2C(=NC(=O)CN1O)C=CC(=C2)Cl)C=CC=C3',
        'C1=CC(=CC=C1C(C2=NCCN2)(C3=NC=CC=C3)O)Cl',
        'C1=CC(=CC=C1C23C(C(=O)NC2=O)C3)Cl',
        'C4=C(C(C2C(C1=CC=NC=C1)C2)(C3=CC=CC=C3)O)C=CC=C4',
        'C1=NC(=NC(=C1CN(C(/C)=C2\\CCOC(=O)S2)C=O)N)C',
        '[C@]23(C1(C(C1)CC2)[C@H](O)C[C@@H]4[C@@H]3CC[C@]5([C@H]4CC[C@@H]5C(
=O)C)C)C',
        'C1=CC=CC=C1NC(OCC3(COC(NC2=CC=CC=C2)=O)CCCC3)=O',
        'CC2(CC(OC(C1NC(=O)CC1)=O)CC(C2)C)C', 'C1=CC=NC=C1C2N(C(=O)CC2)C',
        '[C@H]26[C@H]1[C@@]([C@](C(COC(C)=O)=O)(O)[C@@H](C1)C)(C[C@@H]([C@@H
]2[C@@]3(C(=CC4=C(C3)C=N[N]4C5=CC=CC=C5)C(=C6)C)C)O)C',
        '[C@H]37[C@H]2[C@@]([C@](C(COC(C1=CC(=CC=C1)[S](O)(=O)=O)=O)=O)(O)[C
@@H](C2)C)(C[C@@H]([C@@H]3[C@@]4(C(=CC5=C(C4)C=N[N]5C6=CC=CC=C6)C(=C7)C)C)O
)C',
        'C1=C(O)C=CC4=C1C3(C(C(N(CC2CCC2)CC3)C4)(C)C)CC',
        'C1=C(CC)SC2=C1C(=NCC(N2C)=O)C3=CC=CC=C3Cl',
        '[C@H]24[C@H]1[C@@]([C@](C(CO)=O)(O)CC1)(C[C@@H]([C@@H]2[C@@]3(C(=CC
(=O)C=C3)C(=C4)Cl)C)O)C',
        'C1=CC=CC4=C1C(N(CCCN3CCN(C2=CC(=CC=C2)Cl)CC3)C(N4)=O)=O',
        'C1=C(Cl)C=CC3=C1N(C2=CC=CC=C2)C(=O)N3CCCN(C)C',
        'C1=C(C(=C(C=C1)Cl)NC2=NCCO2)Cl',
        'C1=C(C(=CC(=C1Cl)N)OC)C(NC3CCN(CC2=CC=CC=C2)CC3)=O',
        'C1=C(C#N)C=CC3=C1C4=C(C2=C(C=CC=C2)S3)CCN(CC4)C',
        '[C@]34([C@H](C2[C@@](F)([C@@]1(C(=CC(=O)C=C1)[C@@H](F)C2)C)[C@@H](O
)C3)C[C@H]5OC(O[C@@]45C(=O)COC(=O)C6CC6)(C)C)C',
        'C1=C(Cl)C=CC(=C1C(C2=C(C=CC=C2)Cl)=O)N(C(CNC3CC3)=O)C',
        'C1=C(C(=C(C=C1\\C=C\\C(N2CCCCCC2)=O)OC)OC)OC',
        '[C@@]1([C@H](C2CCC1CC2)NC(C)C)(C3=CC(=C(C=C3)Cl)Cl)O',
        'C1=C(SC2=C1C(=NCC3=NN=C([N]23)C4CCCCC4)C5=CC=CC=C5Cl)Br',
        '[C@@]45([C@@]3([C@H]([C@H]2[C@@H]([C@@]1(C(=CC(=O)C=C1)CC2)C)[C@H](
C3)O)C[C@H]4O[C@H](O5)C6CCCCC6)C)C(COC(C(C)C)=O)=O',
        'C1=C2C(=CC=C1)C(C=C(O2)C(O)=O)=O',
        '[C@@H]2(OC1O[C@@H](O[C@@H]1[C@H]2O)C(Cl)(Cl)Cl)[C@H](O)CO',
        'CC1=CC(=NNC1=O)C',
        '[C@@H](/C=C/C1=C(N=C(C(=C1C2=CC=C(C=C2)F)COC)C(C)C)C(C)C)(C[C@H](CC
(O)=O)O)O',
        'OC1OC(COCCOC(O)C(Cl)(Cl)Cl)C(OC2OC(COCCOCCOC(O)C(Cl)(Cl)Cl)CC(OC(O)
C(Cl)(Cl)Cl)C2O)C(OCCOCCOC(O)C(Cl)(Cl)Cl)C1O',
        'C1=CC(=CC=C1C3CN(CC(N2CC(NCC2)=O)=O)C(C3)=O)Cl',
        'C1=CC=CC2=C1CN(CC(N)=O)C(O2)=O',
        'O=C(C4N(C12CC3CC(C1)CC(C2)C3)CC4)O',
        'C1=C(Cl)C=CC2=C1C(N(CC(=O)N2C)C(N)=O)C3=CC=CC=C3',
        'CC12C(C(CC1=O)CC2)(C)C',
        '[C@]235[C@]([C@H](N(CC1CCC1)CC2)CC4=C3C=C(O)C=C4)(CCCC5)O',
        'C1=CC(=CC2=C1NC(=O)OC2(C)C)Br',
        'C1=C(C3=C(N=C1N2CCN(CC)CC2)CCCCCC3)C4=CC=C(F)C=C4',
        'C1=NC(=NC(=C1CN(C(=C(SSC(=C(\\C)N(C=O)CC2=C(N)N=C(C)N=C2)/CCOC(=O)C
3=CC=CC=C3)/CCOC(=O)C4=CC=CC=C4)\\C)C=O)N)C',
        'C1=CC=CC2=C1C4=C([NH]2)CN3CCN(CC3C4)CCCC(C5=CC=C(C=C5)F)=O',
        'S1[C@H]4N(CC1(C2CN3[C@H](S2)CC3=O)O)C(C4)=O',
        'C(C2C1C(NC(N1)=O)CS2)CCCC(O)=O',
        'O=C2CC1(CCCC1)CC(=O)N2CCNCC4COc3ccccc3O4',
        '[C@]4([C@@]3([C@H]([C@H]2[C@]([C@@]1(C(=CC(=O)C=C1)CC2)C)(F)[C@H](C
3)O)C[C@@H]4C)C)(OC(C5=CC=CC=C5)=O)C(CO)=O',
        '[C@H](O)(/C=C/C1=C(C3=C(OC12CCCC2)C=CC=C3)C4=CC=C(F)C=C4)C[C@H](O)C
C(OCC)=O',
        '[Cl].C1=CC=CC2=C1[N](C=C2CCC3=CC=NC=C3)CC4=CC=CC=C4',
        'C3=C(C(SC(/CCOC(C1=CC=CC=C1)=O)=C(N(CC2=C(N=C(N=C2)C)N)C=O)/C)=O)C=
CC=C3',
        'C4=C(C2=C1C3=C(SC1=NC(=O)CN2)CCCC3)C=CC=C4',
        'C2=C(CN(C(=C(SC(=O)C1=CC=CC=C1)/CCO[P](=O)(O)O)/C)C=O)C(=NC(=N2)C)N
',
        'C(C1(CC(NC(C1)=O)=O)C)C',
        'C1=CC=CC3=C1N=C([N]2C(=NC(=N2)C)C3)N4CCN(C)CC4',
        'C1=CC=CC=C1CC2=NC3=C([N]2CCN(CCO)CC)C(N(C)C(N3C)=O)=O',
        '[C@H]13[C@H](C[C@@H]1C2=CC=C(F)C=C2)CN(C3)CCN4C(=O)C5=C(NC4=O)C=CC=
C5',
        'CN(CCc2cc3cccc4CCc1ccccc1n2c34)Cc5ccccc5',
        'C1=CC=CC2=C1[N]3C(=C2)CNCCC3', 'C1=CC=CC2=C1C3N(CC2)CCNC3',
        'O=C1NCCN1C4CCN(CCC3COc2ccccc2O3)CC4',
        'C3=C(C(O)(C1=CC=CC=C1)C2CCNCC2)C=CC=C3',
        'C1=CC(=CC=C1C(CCCN2CC3N(CC2)CCC3)=O)F',
        'C1=CC(=CC=C1[S](N2CC3CCC(C2)CC3)(=O)=O)N',
        'C1=CC(=CC5=C1N(C4CCN(CC3OC2=C(C=CC=C2)OC3)CC4)C(N5)=O)Cl',
        'C(O)C(O)C1OC(=C(O)C1=O)O',
        'C1=C4C(=CC=C1OCCCCN3CCN(C2=CC=CC(=C2Cl)Cl)CC3)CCC(N4)=O',
        'COc2cc1oc(=O)c(C)c(C)c1cc2OCCCN3CCN(CC3)c4ccccc4OC',
        'C1=CC(=CC=C1C(N2C(CCC2)=O)=O)OC',
        'C3=C(CCN1CCC(CC1)(C2=CC=CC=C2)C(OCC)=O)C=CC(=C3)N',
        'C1=C(C=CC=C1N2C(C=CC=C2)=O)N',
        'C3=C(C(C1=CC=C(C=C1)F)CCCN2CCN(C(NCC)=O)CC2)C=CC(=C3)F',
        '[C@H]4(CC3C2=C1C(=C[N](C1=CC=C2)C(C)C)CC3N(C4)C)C(NC5CCCCC5)=O',
        '[C@@]45([C@@]3([C@H]([C@H]2[C@]([C@@]1(C(=CC(=O)C=C1)CC2)C)(F)[C@H]
(C3)O)C[C@H]4OC6(O5)CCCC6)C)C(COC(C)=O)=O',
        '[C@H]24C([C@@]1(O[C@@](O[C@@H]1C2)(C3=CC=CC=C3)C)C(=O)CO)(C[C@H](O)
[C@@]5(F)C4CCC6=CC(=O)C=CC56C)C',
        'C1=CC=C3C2=C1C(N(CC(O)=O)C(C2=CC=C3)=O)=O',
        'C1=C([N](C(=C1C=O)C)CCNC(C)=O)C',
        'C1=CC=CC3=C1C2(C(NC(=O)CC2)=O)CCC3=O',
        '[H+].C4=C3OCC(N(CCCCN1C(=O)CC2(CC1=O)CCCC2)CCC)CC3=C(OC)C=C4.[Cl-]'
,
        'C1=C3C2=C(C=C1O)C=CC=C2CC(C3)N(CCC)CCC',
        'C1=NC(=NC=C1)N5CCN(CCNC(C23CC4CC(C2)CC(C3)C4)=O)CC5',
        'C1=CC(=CC=C1OCC(OCCNC23CC4CC(C2)CC(C3)C4)=O)Cl',
        'C2=C(CC(C(NCC(OCC1=CC=CC=C1)=O)=O)CSC(C)=O)C=CC=C2',
        'C2=C(CN(C(=C(SC(=O)C1=CC=CO1)\\CCOC(=O)COC(=O)C)/C)C=O)C(=NC(=N2)C)
N',
        '[C@@]3(C1=CC=CS1)(N2CCCCC2)[C@H](CCCC3)C',
        '[Cl-].COc1cccc2C(=O)c3c(O)c4C[C@](O)(C[C@H](O[C@H]5C[C@H](N)[C@H](O
)[C@H](C)O5)c4c(O)c3C(=O)c12)/C(C)=N/NC(=O)c6ccccc6.[H+]',
        'Oc1ccc(cc1)/C=C([N+]#[C-])/C(=C/c2ccc(O)cc2)[N+]#[C-]',
        'CC(C)C1OC(=O)C2=CCCN2C(=O)c3coc(CC(=O)CC(O)\\C=C(C)/C=C\\CNC(=O)\\C
=C/C1C)n3',
        'CCCC[C@@H]1CC(=O)[C@]2(O)O[C@@H]3[C@@H](NC)[C@@H](O)[C@@H](NC)[C@H]
(O)[C@H]3O[C@@H]2O1',
        'CC(C)C(=O)OCc1cccc(OC(=O)[C@@H]2N3[C@H](SC2(C)C)[C@H](NC(=O)Cc4cccc
c4)C3=O)c1',
        'CC1(C)SC2C(NC(=O)CSc3ccccc3)C(=O)N2C1C(O)=O',
        'CCCCCCCCCC(=O)N[C@@H]1[C@@H](O)[C@H](O)[C@@H](CO)O[C@H]1Oc2c3Oc4ccc
(C[C@H]5NC(=O)[C@H](N)c6ccc(O)c(Oc7cc(O)cc(c7)[C@H](NC5=O)C(=O)N[C@H]8C(=O)
N[C@H]9C(=O)N[C@@H]([C@H](O[C@@H]%10O[C@H](CO)[C@@H](O)[C@H](O)[C@H]%10NC(C
)=O)c%11ccc(Oc2cc8c3)c(Cl)c%11)C(=O)N[C@@H](C(O)=O)c%12cc(O)cc(O[C@H]%13O[C
@H](CO)[C@@H](O)[C@H](O)[C@@H]%13O)c%12c%14cc9ccc%14O)c6)cc4Cl',
        'C[C@]1(Cn2ccnn2)[C@@H](N3[C@@H](CC3=O)[S]1(=O)=O)C(O)=O',
        'CC1(C)S[C@@H]2[C@H](NC(=O)[C@H](N)c3ccccc3)C(=O)N2[C@H]1C(=O)OC4OC(
=O)c5ccccc45',
        'CO[C@@H]1C[C@@H](CC[C@H]1O)/C=C(C)/[C@H]2OC(=O)[C@@H]3CCCCN3C(=O)C(
=O)[C@]4(O)O[C@H]([C@H](C[C@@H](C)CC(=C/[C@@H](CC=C)C(=O)C[C@H](O)[C@H]2C)/
C)OC)[C@H](C[C@H]4C)OC',
        'CC1(C)S[C@@H]2[C@H](NC(=O)[C@H](N)c3ccccc3)C(=O)N2[C@H]1C(O)=O.CC4(
C)[C@@H](N5[C@@H](CC5=O)[S]4(=O)=O)C(O)=O',
        'C[C@@H](O)[C@@H]1[C@H]2SC(=C(N2C1=O)C(O)=O)SC3CC[S](=O)C3',
        'COC1C(O)CC(=O)OC(C)C\\C=C\\C=C\\C(OC2CCC(C(C)O2)N(C)C)C(C)CC(CC=O)C
1OC3OC(C)C(OC4CC(C)(O)C(O)C(C)O4)C(C3O)N(C)C',
        'CC1C2CC(OC(C)=O)\\C=C\\C(=C\\CC(O)/C=C/C(=C/C(NC(=O)C(C)=O)C(C)(C(=
O)O2)C1=O)C)C',
        'COC1CCCC2C3C(C(C)O)C(=O)N3C(=C12)C([O-])=O',
        'CC1(C)S[C@@H]2[C@H](NC(=O)[C@H](NC(=O)Cc3ccc(cc3)C4=NCCCN4)c5ccccc5
)C(=O)N2[C@H]1C(O)=O',
        'CC[C@@]1(O)C[C@H](O[C@H]2C[C@@H]([C@H](O[C@H]3C[C@@H]4O[C@H]5CC(=O)
[C@H](C)O[C@H]5O[C@@H]4[C@H](C)O3)[C@H](C)O2)N(C)C)c6c(O)c7C(=O)c8c(O)cccc8
C(=O)c7c(O)c6[C@H]1O[C@H]9C[C@@H]([C@H](O)[C@H](C)O9)N(C)C',
        '[Na+].C[C@@H](O)[C@@H]1[C@H]2SC(=C(N2C1=O)C([O-])=O)COC(N)=O',
        'CO[C@H]1/C=C/O[C@@]2(C)Oc3c(C)c(O)c4c(O)c(NC(=O)C(=C\\C=C\\[C@H](C)
[C@H](O)[C@@H](C)[C@@H](O)[C@@H](C)[C@H](OC(C)=O)[C@@H]1C)/C)c5n6ccc(C)cc6n
c5c4c3C2=O',
        'CO[C@H]1\\C=C\\O[C@@]2(C)Oc3c(C)c(O)c4C(=C(NC(=O)C(=C/C=C/[C@H](C)[
C@H](O)[C@@H](C)[C@@H](O)[C@@H](C)[C@H](OC(C)=O)[C@@H]1C)\\C)C(=C/NN5CCN(CC
5)C6CCCC6)\\C(=O)c4c3C2=O)O',
        'COC1C=COC5(C)Oc4c(C)c(O)c3c(O)c(NC(=O)C(=CC=CC(C)C(O)C(C)C(O)C(C)C(
OC(C)=O)C1C)C)c(C=NN2CCN(C)CC2)c(O)c3c4C5=O',
        'CCN(CC)c1sc2c3NC(=O)C(=C\\C=C\\[C@H](C)[C@H](O)[C@@H](C)[C@@H](O)[C
@@H](C)[C@H](OC(C)=O)[C@H](C)[C@@H](OC)/C=C/O[C@@]4(C)Oc5c(C)c(O)c(c3O)c(c2
n1)c5C4=O)/C',
        'CO[C@H]1/C=C/O[C@@]2(C)Oc3c(C)c(O)c4C(=O)C(=C5NC6(CCN(CC6)CC(C)C)N=
C5c4c3C2=O)NC(=O)C(=C\\C=C\\[C@H](C)[C@H](O)[C@@H](C)[C@@H](O)[C@@H](C)[C@H
](OC(C)=O)[C@@H]1C)/C',
        'CC1(C)S[C@@H]2[C@H](NC(=O)c3nc4ccccc4nc3C(O)=O)C(=O)N2[C@H]1C(O)=O'
,
        'CCCNCC(O)COc1ccccc1C(=O)CCc2ccccc2',
        'Cc1cnn(c1C(=O)N[C@H]2[C@H]3SC(C)(C)[C@@H](N3C2=O)C(O)=O)c4c(Cl)cccc
4Cl',
        'CC1(C)S[C@@H]2[C@H](NC(=O)[C@H](NC(=O)NC3=CN=C(NC3=O)Nc4ccc(cc4)[S]
(N)(=O)=O)c5ccc(O)cc5)C(=O)N2[C@H]1C(O)=O',
        '[Na+].CC1(C)S[C@@H]2[C@H](NC(=O)[C@H](NC(=O)C3=CC=C(NC3=O)c4ccc(cc4
)[S](=O)(=O)N(CCO)CCO)c5ccc(O)cc5)C(=O)N2[C@H]1C([O-])=O',
        'CC1(C)SC2C(N([C@H](C(=O)NC(=O)CN)c3ccccc3)C(=N)c4ccncc4)C(=O)N2C1C(
O)=O',
        'COc1cccc2C(=O)c3c(O)c4C[C@](O)(C[C@H](O[C@H]5C[C@H](N)[C@H](O[C@H]6
CCCCO6)[C@H](C)O5)c4c(O)c3C(=O)c12)C(=O)CO',
        'CN(C)[C@H]1[C@@H]2C[C@H]3C(=C(O)c4c(O)cccc4[C@@]3(C)O)C(=O)[C@]2(O)
C(=O)\\C(=C(/O)NCNC(C(=O)NC5C6SC(C)(C)C(N6C5=O)C(O)=O)c7ccccc7)C1=O',
        'CN(C)[C@H]1[C@@H]2C[C@H]3C(=C(O)c4c(O)cccc4[C@@]3(C)O)C(=O)[C@]2(O)
C(=O)\\C(=C(/O)NCN5CCCC(C5)C(O)=O)C1=O',
        'CC(=O)O[C@H]1C(=O)[C@]2(C)[C@@H](O)C[C@H]3OC[C@@]3(OC(C)=O)C2[C@H](
OC(=O)c4ccccc4)[C@]5(O)C[C@H](OC(=O)[C@H](O)[C@@H](NC(=O)c6ccccc6)c7ccccc7)
C(=C1C5(C)C)C',
        'CC1(C)NC(C(=O)[NH+]1[C@H]2[C@H]3SC(C)(C)[C@@H](N3C2=O)C(O)=O)c4ccc(
O)cc4',
        'CO[C@@H]1[C@@H](OC(N)=O)[C@@H](O)[C@H](Oc2ccc3C(=O)C(=C(O)Oc3c2C)NC
(=O)c4ccc(O)c(CC=C(C)C)c4)OC1(C)C',
        'COC(=O)C1=C(C)NC(=C(C1c2cccc(c2)[N+]([O-])=O)C(=O)OCCN(C)Cc3ccccc3)
C',
        'CO[C@]1(NC(=O)C(C(O)=O)c2ccc(O)cc2)[C@H]3OCC(=C(N3C1=O)C(O)=O)CSc4n
nnn4C',
        'CO[C@@]12[C@H](COC(N)=O)C3=C(N1C[C@@H]4N[C@H]24)C(=O)C(=C(N)C3=O)C'
,
        'COCC(=O)O[C@]1(CCN(C)CCCc2[nH]c3ccccc3n2)CCc4cc(F)ccc4[C@@H]1C(C)C'
,
        'CC1(C)S[C@@H]2[C@H](NC(=O)[C@H](NC(=O)N3CCN(C3=O)[S](C)(=O)=O)c4ccc
cc4)C(=O)N2[C@H]1C(O)=O',
        'COc1cccc(OC)c1C(=O)N[C@H]2[C@H]3SC(C)(C)[C@@H](N3C2=O)C(O)=O',
        'CCCCc1nc(Cl)c(CO)n1Cc2ccc(cc2)c3ccccc3c4n[nH]nn4',
        'O.N[C@@H](C(=O)NC1C2CCC(=C(N2C1=O)C(O)=O)Cl)c3ccccc3',
        'CC1=C(COC(=O)[C@@H]2N3[C@H](SC2(C)C)[C@H](NC(=O)[C@H](N)c4ccccc4)C3
=O)OC(=O)O1',
        'O.C[C@H](O)[C@@H]1[C@H]2CC(=C(N2C1=O)C(O)=O)SCCN=CN',
        'CC(C)CC1C(=O)NC(C(=O)N2CCCC2C(=O)NC(C(=O)NC(C(=O)NC(C(=O)NC(C(=O)N3
CCCC3C(=O)NC(C(=O)NC(C(=O)N1)CCCN)C(C)C)CC4=CC=CC=C4)CC(C)C)CCCN)C(C)C)CC5=
CC=CC=C5',
        'C[C@@H]1[C@H](O)CC[C@@]2(C)[C@H]1CC[C@@]3(C)[C@H]2[C@H](O)C[C@H]4\\
C([C@H](C[C@]34C)OC(C)=O)=C(/CCC=C(C)C)C(O)=O',
        'CC(C)CC(OC(=O)c1occc1)C(=O)N[C@H]2[C@H]3SC(C)(C)[C@@H](N3C2=O)C(O)=
O',
        'CC1(C)S[C@@H]2[C@H](NC(=O)[C@@H](N=Cc3occc3)c4ccc(O)cc4)C(=O)N2[C@H
]1C(O)=O',
        'C[C@@H](O)[C@@H]1[C@H]2SC(=C(N2C1=O)C(O)=O)[C@H]3CCCO3',
        '[Na].CO[C@]1(NC(=O)CSC(F)(F)F)[C@H]2OCC(=C(N2C1=O)C(O)=O)CSc3nnnn3C
CO',
        'CC1(C)S[C@@H]2[C@H](NC(=O)[C@H](NC(=O)C(C)(C)Oc3ccc(Cl)cc3)c4ccccc4
)C(=O)N2[C@H]1C(O)=O',
        'CC1(C)S[C@@H]2[C@H](NC(=O)C(Oc3ccccc3)c4ccccc4)C(=O)N2[C@H]1C(O)=O'
,
        'CN(C)[C@H]1[C@@H]2C[C@H]3C(=C(O)c4c(O)cccc4[C@@]3(C)O)C(=O)[C@]2(O)
C(=O)\\C(=C(/O)NCN(C)CCN(C)CNC(/O)=C/5C(=O)[C@H]([C@@H]6C[C@H]7C(=C(O)c8c(O
)cccc8[C@@]7(C)O)C(=O)[C@]6(O)C5=O)N(C)C)C1=O',
        'CCC(C)[C@H]1O[C@]2(CC[C@@H]1C)CC3C[C@@H](C\\C=C(C)\\[C@@H](O[C@H]4C
[C@H](OC)[C@@H](OC5C[C@H](OC)[C@@H](O)[C@H](C)O5)[C@H](C)O4)[C@@H](C)/C=C/C
=C/6CO[C@@H]7[C@H](O)C(=C[C@@H](C(=O)O3)[C@]67O)C)O2.CO[C@H]8CC(O[C@@H](C)[
C@@H]8O)O[C@H]9[C@H](C)O[C@H](C[C@@H]9OC)O[C@H]/%10[C@@H](C)/C=C/C=C/%11CO[
C@@H]%12[C@H](O)C(=C[C@@H](C(=O)OC%13C[C@@H](C\\C=C%10C)O[C@@]%14(CC[C@H](C
)[C@H](O%14)C(C)C)C%13)[C@]%11%12O)C',
        'CC1(C)S[C@@H]2[C@H](NC(=O)[C@H](N)C3=CCC=CC3)C(=O)N2[C@H]1C(O)=O',
        '[Cl].CC[C@H]1CN2CCc3cc(OC)c(OC)cc3[C@@H]2C[C@@H]1C[C@H]4NCCc5cc(OC)
c(OC)cc45',
        'CCC(C(=O)NC\\C=C\\C=C(/C)C(OC)C(C)C1OC(\\C=C\\C=C\\C=C(/C)C(=O)C2=C
(O)N(C)C=CC2=O)C(O)C1O)C3(O)OC(\\C=C\\C=C\\C)C(C)(C)C(OC4OC(C)C(OC5OC(C)C(O
C)C(O)C5OC)C(OC)C4O)C3O',
        'CC(=O)O[C@@]12CO[C@@H]1C[C@H](O)[C@]3(C)[C@@H]2[C@H](OC(=O)c4ccccc4
)[C@]5(O)C[C@H](OC(=O)[C@H](O)[C@@H](NC(=O)OC(C)(C)C)c6ccccc6)C(=C([C@@H](O
)C3=O)C5(C)C)C',
        'CC[C@H]1OC(=O)[C@H](C)[C@@H](O[C@H]2C[C@@](C)(OC)[C@@H](O)[C@H](C)O
2)[C@H](C)[C@@H](O[C@@H]3O[C@H](C)C[C@@H]([C@H]3O)N(C)C)[C@](C)(O)C[C@@H](C
)[C@@H]4N[C@@H](COCCOC)O[C@H]([C@H]4C)[C@]1(C)O',
        'OCCN(CCO)c1nc(N2CCCCC2)c3nc(nc(N4CCCCC4)c3n1)N(CCO)CCO',
        'COc1ccc(cc1O)C2=CC(=O)c3c(O)cc(O[C@@H]4O[C@H](CO[C@@H]5O[C@@H](C)[C
@H](O)[C@@H](O)[C@H]5O)[C@@H](O)[C@H](O)[C@H]4O)cc3O2',
        'CC(C)C1NC(=O)C(NC(=O)c2ccc(C)c3OC4=C(C)C(=O)C(=C(C(=O)NC5C(C)OC(=O)
C(C(C)C)N(C)C(=O)CN(C)C(=O)C6CCCN6C(=O)C(NC5=O)C(C)C)C4=Nc23)N)C(C)OC(=O)C(
C(C)C)N(C)C(=O)CN(C)C(=O)C7CCCN7C1=O',
        'CCC1C(=O)N(CC(=O)N(C(C(=O)NC(C(=O)N(C(C(=O)NC(C(=O)NC(C(=O)N(C(C(=O
)N(C(C(=O)N(C(C(=O)N(C(C(=O)N1)C(C(C)CC=CC)O)C)C(C)C)C)CC(C)C)C)CC(C)C)C)C)
C)CC(C)C)C)C(C)C)CC(C)C)C)C',
        'CC1(C)S[C@@H]2[C@H](NC(=O)C3(N)CCCCC3)C(=O)N2[C@H]1C(O)=O',
        'CC(=O)C1=C(O)[C@]2(O)[C@@H](Cc3c(C)c4ccc(C)c(O)c4c(O)c3C2=O)[C@@H](
N)C1=O',
        'CO/N=C(C(=O)N[C@H]1[C@H]2SCC(=C(N2C1=O)C(O)=O)CSc3snnc3)/c4csc(N)n4
',
        'CO/N=C(C(=O)N[C@H]1[C@@H]2N(C1=O)C(=C(COC(C)=O)C[S]2=O)C(O)=O)/c3cs
c(N)n3',
        'CO/N=C(C(=O)N[C@H]1[C@H]2SCC(=C(N2C1=O)C(O)=O)\\C=C\\SC3=NNC(=O)C(=
O)N3CC=O)/c4csc(N)n4',
        'CO/N=C(C(=O)N[C@H]1[C@H]2SCC(=C(N2C1=O)C(O)=O)CSC(=O)c3occc3)/c4csc
(N)n4',
        'CO/N=C(C(=O)N[C@H]1[C@H]2SCC(=C(N2C1=O)C(O)=O)Cn3nnc(C)n3)/c4csc(N)
n4',
        'COC1=C(N3C(SC1)C(NC(=O)C(N)C2C=CCC=C2)C3=O)C(O)=O',
        'CC1=C(N2[C@H](SC1)[C@H](NC(=O)Cc3ccc(cc3)C4=NCCCN4)C2=O)C(O)=O',
        'Cn1nnnc1SCC2=C(N3[C@H](SC2)C(NC(=O)[C@H](NC(=O)C4=CNC(=CC4=O)C)c5cc
c(O)cc5)C3=O)C(O)=O',
        'CC(=O)OCC1=C(N2[C@H](SC1)[C@H](NC(=O)c3c(C)onc3c4ccccc4Cl)C2=O)C(O)
=O',
        'CN(C)CCn1nnnc1SCC2=C(N3[C@H](SC2)[C@H](NC(=O)Cc4csc(N)n4)C3=O)C(O)=
O',
        'CO[C@]1(NC(=O)C2S\\C(S2)=C(\\C(N)=O)C(O)=O)[C@H]3SCC(=C(N3C1=O)C(O)
=O)CSc4nnnn4C',
        'CO\\N=C(C(=O)N[C@H]1[C@H]2SCC(=C(N2C1=O)C(O)=O)CSc3sc(CC(O)=O)c(C)n
3)/c4csc(N)n4',
        'CO/N=C(C(=O)N[C@H]1[C@H]2SCC(=C(N2C1=O)C(O)=O)CSc3nnnn3C)/c4csc(N)n
4',
        'OC(=O)C1=C(CS[C@@H]2[C@H](NC(=O)Cc3sccc3)C(=O)N12)CSc4[nH]ncn4',
        'CO\\N=C(C(=O)NC1[C@H]2SCC(=C(N2C1=O)C(O)=O)\\C=C/c3scnc3C)\\c4csc(N
)n4',
        'CC(N)C(=O)OC(C(=O)NC1C2SCC(=C(N2C1=O)C(=O)OCC3=C(C)OC(=O)O3)CSc4sc(
C)nn4)c5ccccc5',
        'Cc1sc(SCC2=C(N3[C@H](SC2)[C@H](NC(=O)CN4C=C(Cl)C(=O)C(=C4)Cl)C3=O)C
(O)=O)nn1',
        'NC(C(=O)N[C@H]1[C@H]2SCC(=C(N2C1=O)C(O)=O)CSc3cn[nH]n3)c4ccc(O)cc4'
,
        'CC1(C)S[C@@H]2[C@H](NC(=O)C(C(=O)Oc3ccc4CCCc4c3)c5ccccc5)C(=O)N2[C@
H]1C(O)=O',
        'CC1(C)S[C@@H]2[C@H](NC(=O)C(C(=O)Oc3ccccc3)c4ccccc4)C(=O)N2[C@H]1C(
O)=O',
        'CC(C)COCC(CN(Cc1ccccc1)c2ccccc2)N3CCCC3',
        'CN[C@H]1[C@@H](O)[C@H]2O[C@@H](O[C@@H]3[C@@H](N)C[C@@H](N)[C@H](O)[
C@H]3O)[C@H](N)C[C@@H]2O[C@@H]1O[C@H]4O[C@H](CO)[C@@H](N)[C@H](O)[C@H]4O',
        'CC1(C)S[C@@H]2C(NC(=O)[C@H](NC(=O)C3=CNc4cccnc4C3=O)c5ccccc5)C(=O)N
2[C@H]1C(O)=O',
        'CC1(C)S[C@@H]2[C@H](NC(=O)C34C[C@H]5C[C@H](CC(N)(C5)C3)C4)C(=O)N2[C
@H]1C(O)=O',
        'COc1cccc2C(=O)c3c(O)c4CC(O)(CC(O)c4c(O)c3C(=O)c12)C(=O)CO',
        'CC[C@@]1(O)C[C@H](OC2CC(C(OC3CC(O)C(OC4CCC(=O)C(C)O4)C(C)O3)C(C)O2)
N(C)C)c5c(O)c6C(=O)c7c(O)cccc7C(=O)c6cc5[C@H]1C(=O)OC'],
      dtype=object)
```

In [225…

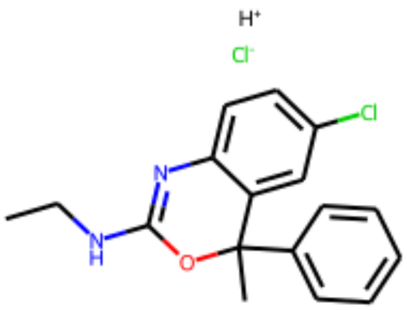
```python
smiles = 'CN1C=NC2=C1C(=O)N(C(=O)N2C)C'
mol = Chem.MolFromSmiles(smiles)
fingerPrint = RDKFingerprint(mol)
np.array(fingerPrint)
```

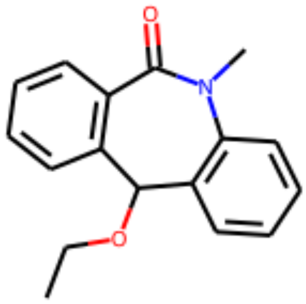
Out[225…

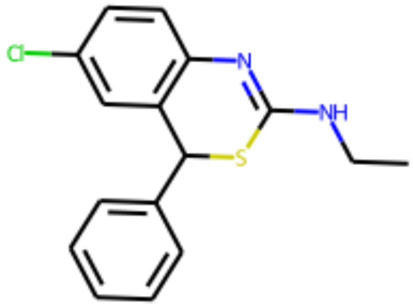
```
array([1, 1, 0, ..., 0, 0, 1])
```

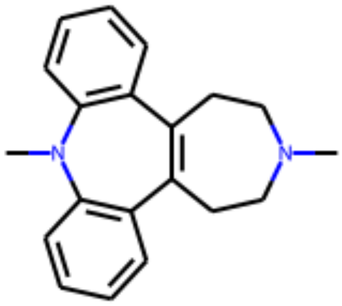
In [226…

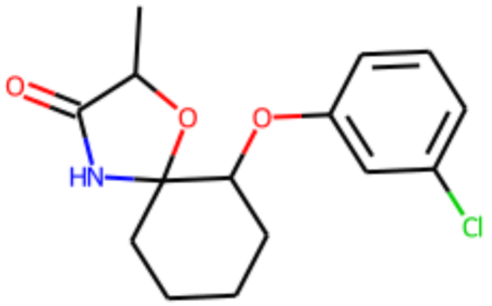
```python
smile = bbbp_data[1][1].ids
fig_prints = []
for i in range(len(smile)):
    fig_prints.append(Chem.MolFromSmiles(smile[i]))
for i in range(10):
    display(fig_prints[i])
```

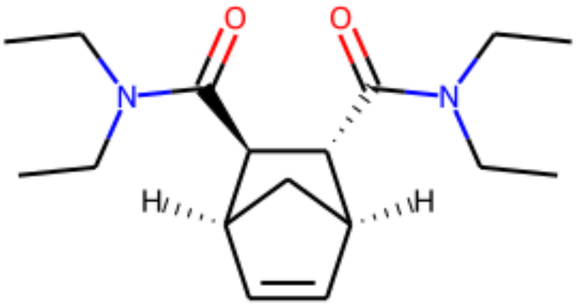


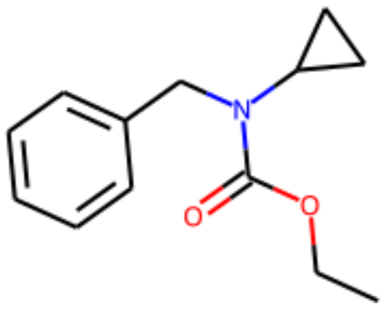

O
N
O

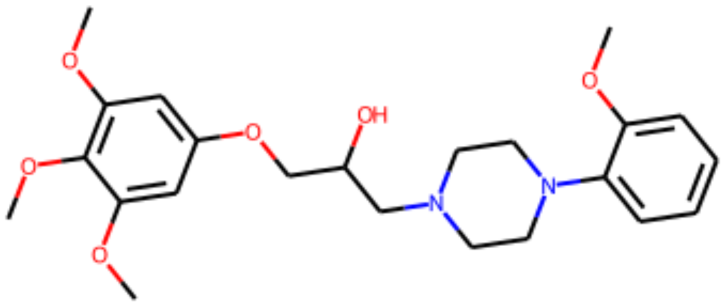

Cl
N
NH
S

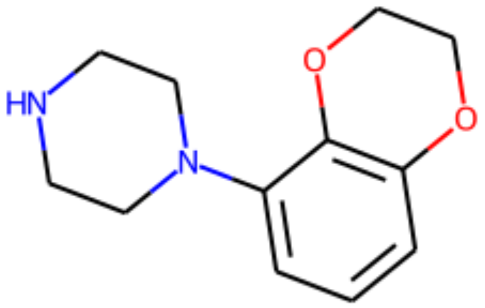

N
N

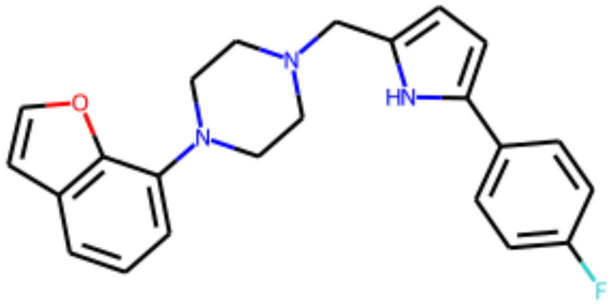

O
O
HN
O
Cl


O
O
N
N
H
H









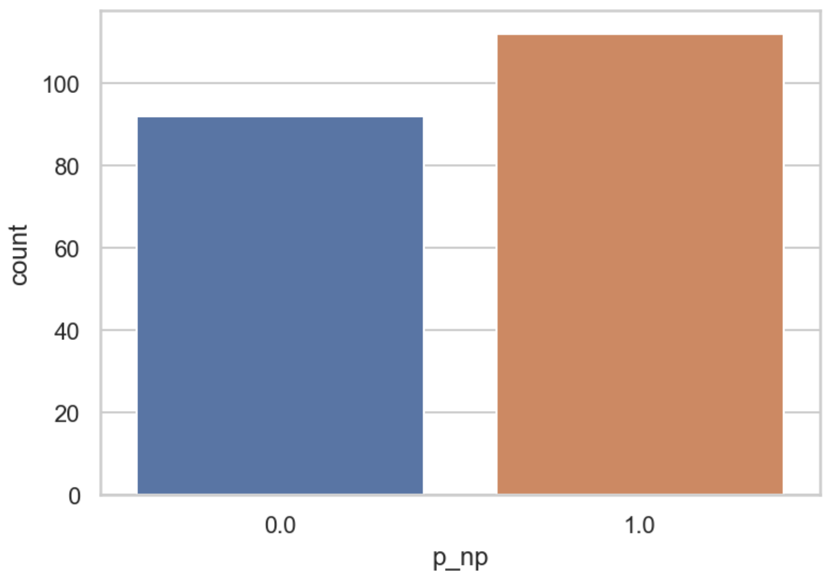
In [112…
```
fp_array = bbbp_data[1][0].X
y = bbbp_data[1][0].y
print(len(fp_array), len(y))
```

```
1631 1631
```

In [113…
```
fp_test_array = bbbp_data[1][1].X
y_test = bbbp_data[1][1].y
print(len(fp_test_array), len(y_test))
```

```
204 204
```

In [114…
```
import seaborn as sns
sns.countplot(pd.DataFrame(y_test, columns = ['p_np'])['p_np'])
plt.show()
```



We can see from above that the count of p_np for values of 0 and 1 are close to each other, so the classes are balanced

Lets start splitting the data by k-fold cross validation

In [120…
```
# Use this shuffled data for k-fold cv
from sklearn.metrics import average_precision_score

def k_fold_generator(X, y, n_folds, model, model_name): # n_folds can be 5

  # Shuffle data first
    X, y =  shuffle(X, y)

    fold_inx_start = 0
    fold_inx_end = int(len(X)/n_folds)

    train_scores = []
    valid_scores = []

    for i in range(n_folds):
        train_x_fold =  np.concatenate( (X[:fold_inx_start, :], X[fold_inx_
        train_y_fold = np.concatenate( (y[:fold_inx_start, :], y[fold_inx_e

        valid_x_fold = X[fold_inx_start:fold_inx_end, :]
        valid_y_fold = y[fold_inx_start:fold_inx_end, :]

        model.fit(train_x_fold, train_y_fold)

        train_preds = model.predict(train_x_fold)
        valid_preds = model.predict(valid_x_fold)


        train_scores.append( average_precision_score(train_y_fold, train_pr
        valid_scores.append( average_precision_score(valid_y_fold, valid_pr


        fold_inx_start = fold_inx_end
        fold_inx_end = fold_inx_start + int(len(X)/n_folds)


    return train_scores, valid_scores, model
```

LogisticRegression Model

In [121…
```
from sklearn.utils import shuffle
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(penalty = "l2", C = 0.5)
train_scores, valid_scores, lr = k_fold_generator(fp_array, y, 5, lr, 'lr'
model_scores = pd.DataFrame(columns = ['Model', 'Validation fold 1 Score',

print('Mean training scores', np.mean(train_scores), ',  Mean validation sc
```

```
Mean training scores 0.9711439596395174 ,  Mean validation scores 0.9243050
261161452
```

In [122…
```
# Add validation fold values to table
model_scores = model_scores.append({'Model': 'Logistic Regression', 'Valida
```

In [123…
```
print(classification_report(lr.predict(fp_test_array), y_test))
```

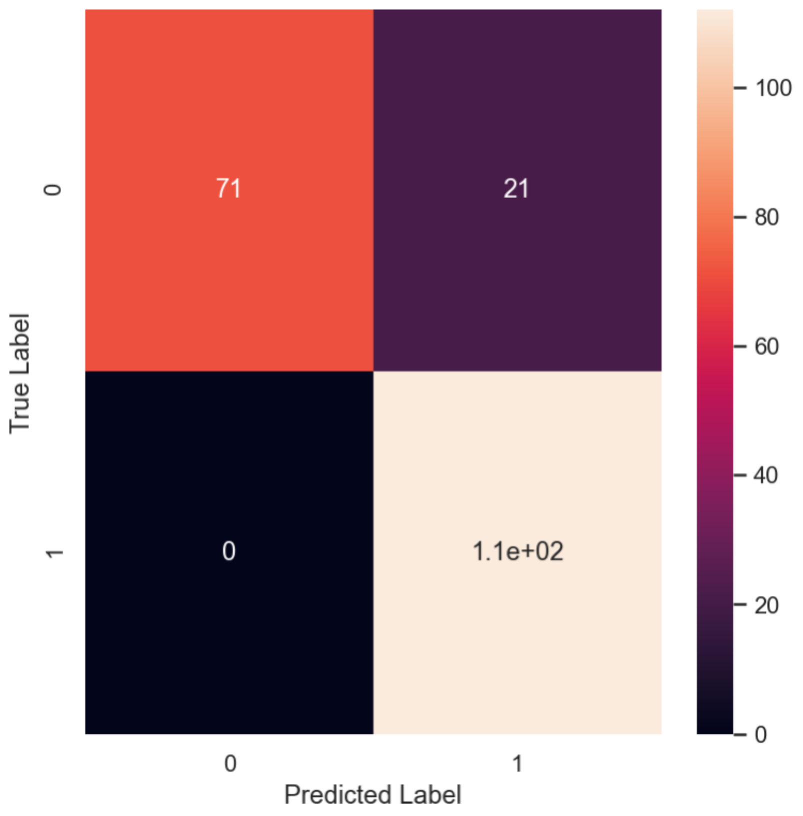
```
              precision    recall  f1-score   support

         0.0       0.77      1.00      0.87        71
         1.0       1.00      0.84      0.91       133

    accuracy                           0.90       204
   macro avg       0.89      0.92      0.89       204
weighted avg       0.92      0.90      0.90       204
```

In [124…

```
# Confusion matrix
cm = confusion_matrix(y_test, lr.predict(fp_test_array))
df_cm = pd.DataFrame(cm, index = [0, 1], columns = [0,1])
plt.figure(figsize=(6, 6))
sns.heatmap(df_cm, annot=True)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
```

Out[124…

```
Text(54.25999999999999, 0.5, 'True Label')
```



LogisticRegression with Lasso

In [125…

```
ls = LogisticRegression(penalty = 'l1', C=0.01, solver='liblinear') # L1 i
train_scores, valid_scores, ls = k_fold_generator(fp_array, y, 5, ls, 'ls'

print('Mean training scores', np.mean(train_scores), ',  Mean validation sc
```

```
Mean training scores 0.8222222222222222 ,  Mean validation scores 0.8220858
895705522
```

In [126…
```
print(ls.coef_)
print(ls.intercept_)
```
```
[[0. 0. 0. ... 0. 0. 0.]]
[1.08740482]
```

In [127…
```
### Coefficients of almost features is 0 - this is probably because of larg
```

In [128…
```
# Updated C value
ls = LogisticRegression(penalty = 'l1', C=0.5, solver='liblinear') # L1 is
train_scores, valid_scores, ls = k_fold_generator(fp_array, y, 5, ls, 'ls'

print('Mean training scores', np.mean(train_scores), ',  Mean validation sc
```
```
Mean training scores 0.9385876215038907 ,  Mean validation scores 0.9172914
842510052
```

In [129…
```
print(ls.coef_)
print(ls.intercept_)
```
```
[[0. 0. 0. ... 0. 0. 0.]]
[3.41162366]
```

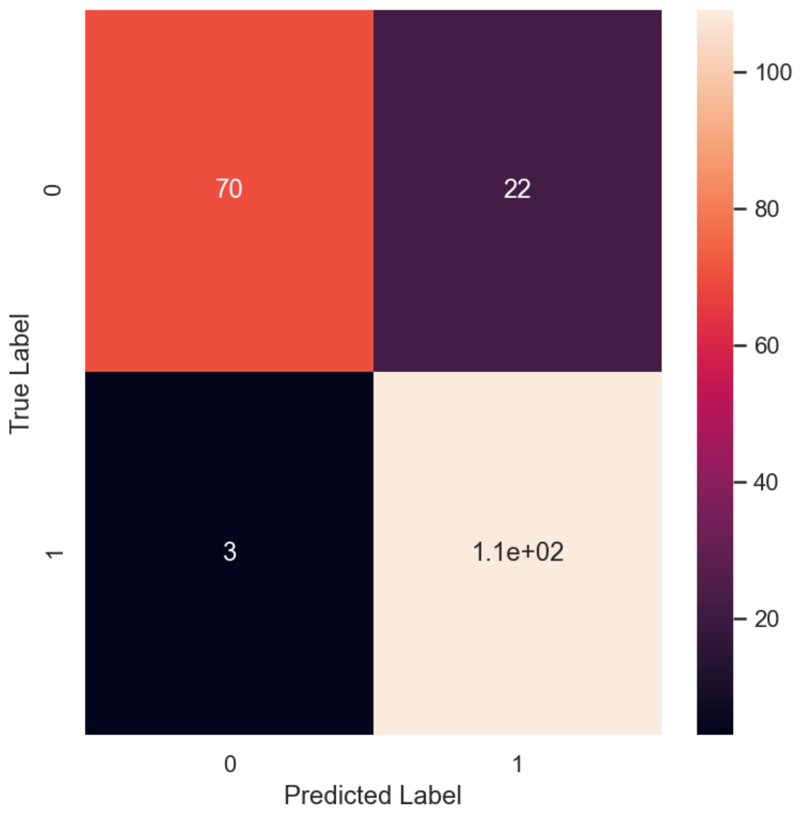
In [130…
```
# Add validation fold values to table
model_scores = model_scores.append({'Model': 'Lasso Classifier (L1)', 'Vali
```

In [131…
```
ls_predictions = ls.predict(fp_test_array)
```

In [132…
```
print(classification_report(ls_predictions, y_test))
```
```
              precision    recall  f1-score   support

         0.0       0.76      0.96      0.85        73
         1.0       0.97      0.83      0.90       131

    accuracy                           0.88       204
   macro avg       0.87      0.90      0.87       204
weighted avg       0.90      0.88      0.88       204
```

In [133…
```
# Confusion matrix
cm = confusion_matrix(y_test, ls_predictions)
df_cm = pd.DataFrame(cm, index = [0, 1], columns = [0,1])
plt.figure(figsize=(6, 6))
sns.heatmap(df_cm, annot=True)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
```

```
Out[133… Text(54.25999999999999, 0.5, 'True Label')
```



SVC Model

```
In [134… from sklearn.svm import SVC
svc = SVC(C = 0.8, kernel = 'rbf' ) # Adding small regularization paramete

train_scores, valid_scores, svc = k_fold_generator(fp_array, y, 5, svc, 'sv

print('Mean training scores', np.mean(train_scores), ',  Mean validation sc
```

```
Mean training scores 0.9380414575234244 ,  Mean validation scores 0.9091995
96775173
```

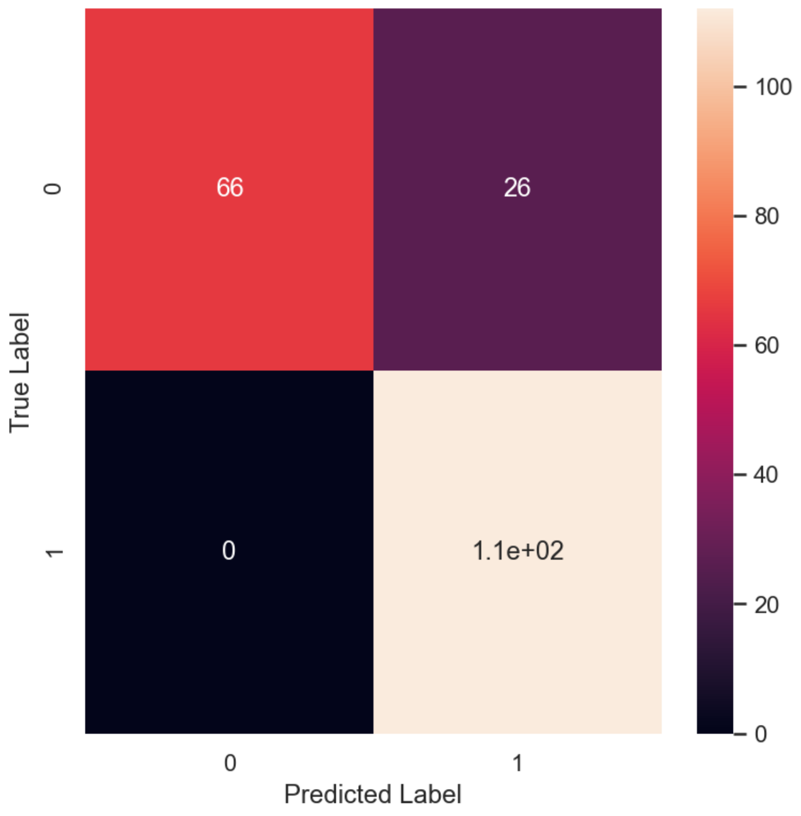
```
In [135… # Add validation fold values to table
model_scores = model_scores.append({'Model': 'Support Vector Classifier',
```

```
In [136… svc_predictions = svc.predict(fp_test_array)

print(classification_report(svc_predictions, y_test))
```

```
              precision    recall  f1-score   support

         0.0       0.72      1.00      0.84        66
         1.0       1.00      0.81      0.90       138

    accuracy                           0.87       204
   macro avg       0.86      0.91      0.87       204
weighted avg       0.91      0.87      0.88       204
```

In [137…
```
# Confusion matrix
cm = confusion_matrix(y_test, svc_predictions)
df_cm = pd.DataFrame(cm, index = [0, 1], columns = [0,1])
plt.figure(figsize=(6, 6))
sns.heatmap(df_cm, annot=True)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
```

Out[137…
```
Text(54.25999999999999, 0.5, 'True Label')
```



Gaussian Naive Bayes

In [138…
```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()

train_scores, valid_scores, gnb = k_fold_generator(fp_array, y, 5, gnb, 'g

print('Mean training scores', np.mean(train_scores), ',  Mean validation s
```

```
Mean training scores 0.9143307211931404 ,  Mean validation scores 0.8555428
984731398
```

In [139…
```
# Add validation fold values to table
model_scores = model_scores.append({'Model': 'Gaussian Naive Bayes Classif
```

In [140…
```
gnb_predictions = gnb.predict(fp_test_array)

print(classification_report(gnb_predictions, y_test))
```

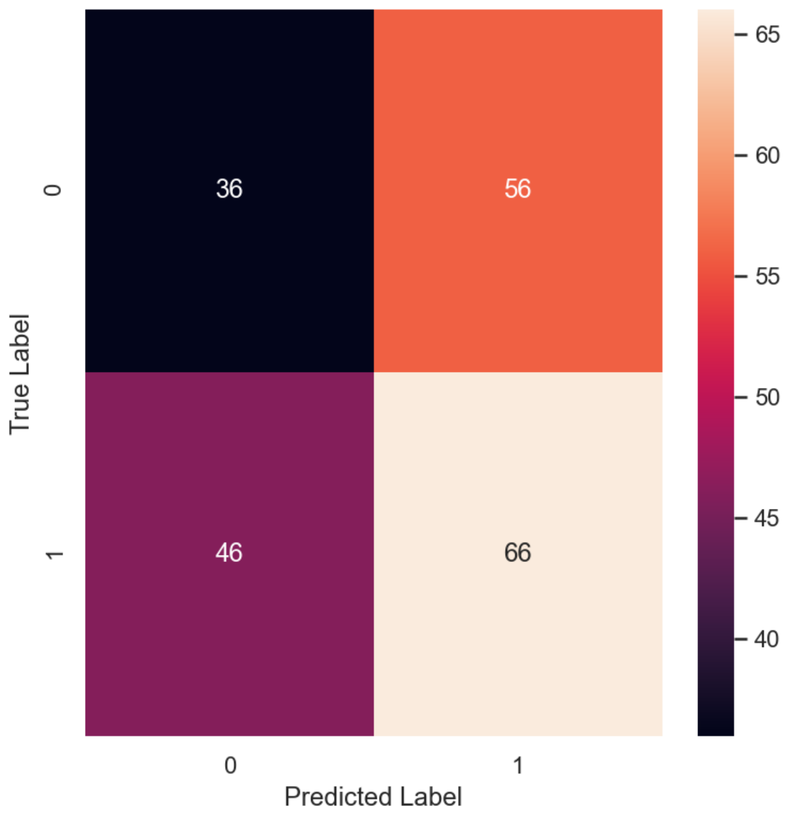
```
              precision    recall  f1-score   support

         0.0       0.39      0.44      0.41        82
         1.0       0.59      0.54      0.56       122

    accuracy                           0.50       204
   macro avg       0.49      0.49      0.49       204
weighted avg       0.51      0.50      0.50       204
```

In [141…

```
# Confusion matrix
cm = confusion_matrix(y_test, gnb_predictions)
df_cm = pd.DataFrame(cm, index = [0, 1], columns = [0,1])
plt.figure(figsize=(6, 6))
sns.heatmap(df_cm, annot=True)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
```

Out[141…

```
Text(54.25999999999999, 0.5, 'True Label')
```



In [142…

```
model_scores
```

Out[142…

	Model	Validation fold 1 Score	Validation fold 2 Score	Validation fold 3 Score	Validation fold 4 Score	Validation fold 5 Score	Mean AUPR Score
0	Logistic Regression	0.912470	0.922513	0.928506	0.937749	0.920286	0.924305
1	Lasso Classifier (L1)	0.928900	0.906990	0.920275	0.913917	0.916375	0.917291
2	Support Vector Classifier	0.900022	0.913080	0.909301	0.895114	0.928481	0.909200
3	Gaussian Naive Bayes Classifier	0.843203	0.884258	0.878238	0.826347	0.845667	0.855543

Descriptors dataframe contains 1625 molecular descriptors (including 3D descriptors) generated on the NCI database using Mordred python module

Lets examine the Descriptors dataframe to how the molecules in the data predict the permeability of the drug.

Lets examine the Descriptors dataframe to how the molecules in the data predict the permeability of the drug.

In [172…
```
import pandas as pd

bbbp_descriptors_df = pd.read_csv('/Users/jagan/Downloads/BBBP_Descriptors,
```

In [173…
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
from numpy import mean
from numpy import absolute
from numpy import sqrt
import pandas as pd
```

In [182…
```
print(bbbp_descriptors_df.shape)
bbbp_descriptors_df.head()
x=bbbp_descriptors_df.pop('p_np')
bbbp_descriptors_df.insert(1825, 'p_np', column_to_move)
bbbp_descriptors_df.head()
```

```
(2039, 1826)
```

Out[182…

	ABC	ABCGG	nAcid	nBase	SpAbs_A	SpMax_A	
0	14.389425	11.808563	0	1	multiple fragments (SpAbs_A/SpAbs)	multiple fragments (SpMax_A/SpMax)	mult (SpDia
1	16.809162	13.974216	0	0	27.07079790976892	2.2954078760731984	4.5908
2	20.758034	16.164169	1	1	33.465822016594636	2.5785233443199886	5.1570
3	15.775129	12.193243	0	1	26.569834753519054	2.2810418592150574	4.562(
4	23.095142	19.875288	1	0	36.20647450941663	2.6329498486229084	5.1219

5 rows × 1826 columns

Finding correlation between the columns

In [213…

```
correlation=bbbp_descriptors_df.corr()
l=correlation['p_np']
ll=[]

for i in l:
    if i>0:
        ll.append(i)
ll.sort()

print("n_np is highly correlated with",l[l==ll[-2]])

print("Top 10 features that have decisive role on permeability are",)

for i in range(-11,-1,1):

    print(l[l==ll[i]])
```

```
n_np is highly correlated with ATSC1c    0.50154
Name: p_np, dtype: float64
Top 10 features that have decisive role on permeability are
MATS1are    0.310021
Name: p_np, dtype: float64
MATS1pe    0.315559
Name: p_np, dtype: float64
AATSC1c    0.333928
Name: p_np, dtype: float64
Lipinski    0.334814
Name: p_np, dtype: float64
SsssCH    0.360272
Name: p_np, dtype: float64
SdssC    0.385012
Name: p_np, dtype: float64
ATSC1pe    0.40103
Name: p_np, dtype: float64
ATSC1se    0.410719
Name: p_np, dtype: float64
ATSC1are    0.4139
Name: p_np, dtype: float64
ATSC1c    0.50154
Name: p_np, dtype: float64
```

In [220…

```
#Correlation of all columns
bbbp_descriptors_df.corr()
```

Out[220…

	ABC	ABCGG	nAcid	nBase	nAromAtom	nAromBond	nAto
ABC	1.000000	0.985616	0.129367	0.154078	0.259233	0.259231	0.9589
ABCGG	0.985616	1.000000	0.155547	0.128688	0.188533	0.188625	0.9488
nAcid	0.129367	0.155547	1.000000	0.017861	0.032844	0.031539	0.0580
nBase	0.154078	0.128688	0.017861	1.000000	0.120338	0.115536	0.1976
nAromAtom	0.259233	0.188533	0.032844	0.120338	1.000000	0.998078	0.1312
...	...	...	...	...	...	...	
WPol	0.956784	0.954588	0.053085	0.101619	0.131150	0.132505	0.9175
Zagreb1	0.995069	0.983171	0.116876	0.137825	0.225857	0.226590	0.9479
Zagreb2	0.981897	0.972725	0.103635	0.121826	0.194939	0.196359	0.9302
mZagreb2	0.987636	0.977035	0.138768	0.165607	0.263376	0.262894	0.9660
p_np	-0.314142	-0.344348	-0.309663	-0.074009	-0.001848	-0.004016	-0.2722

879 rows × 879 columns

In [26]:

```
# function to coerce all data types to numeric

def coerce_to_numeric(df, column_list):
    df[column_list] = df[column_list].apply(pd.to_numeric, errors='coerce'
```

In [27]:
```
coerce_to_numeric(bbbp_descriptors_df, bbbp_descriptors_df.columns)
bbbp_descriptors_df.head()
```

Out[27]:

	ABC	ABCGG	nAcid	nBase	SpAbs_A	SpMax_A	SpDiam_A	SpAD_A	SpM
0	14.389425	11.808563	0	1	NaN	NaN	NaN	NaN	
1	16.809162	13.974216	0	0	27.070798	2.295408	4.590816	27.070798	1.17
2	20.758034	16.164169	1	1	33.465822	2.578523	5.157047	33.465822	1.2
3	15.775129	12.193243	0	1	26.569835	2.281042	4.562084	26.569835	1.26
4	23.095142	19.875288	1	0	36.206475	2.632950	5.121946	36.206475	1.24

5 rows × 1825 columns

In [28]:
```
bbbp_descriptors_df = bbbp_descriptors_df.fillna(0)
bbbp_descriptors_df.head()
```

Out[28]:

	ABC	ABCGG	nAcid	nBase	SpAbs_A	SpMax_A	SpDiam_A	SpAD_A	SpM
0	14.389425	11.808563	0	1	0.000000	0.000000	0.000000	0.000000	0.00
1	16.809162	13.974216	0	0	27.070798	2.295408	4.590816	27.070798	1.17
2	20.758034	16.164169	1	1	33.465822	2.578523	5.157047	33.465822	1.2
3	15.775129	12.193243	0	1	26.569835	2.281042	4.562084	26.569835	1.26
4	23.095142	19.875288	1	0	36.206475	2.632950	5.121946	36.206475	1.24

5 rows × 1825 columns

Lets do Scaling and Principle Component Analysis

In [31]:
```
from sklearn.decomposition import KernelPCA
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.utils import class_weight
from sklearn.metrics import roc_curve, auc, roc_auc_score, f1_score
from sklearn.model_selection import train_test_split
from sklearn.calibration import CalibratedClassifierCV
import pickle
from keras.layers import Dense, Activation, Dropout, BatchNormalization, In
from keras.models import Sequential, Model
from keras import optimizers, regularizers, initializers
from keras.callbacks import ModelCheckpoint, Callback
from keras import backend as K
import tensorflow as tf
from xgboost import XGBClassifier
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
```

In [32]:
```
bbbp_scaler1 = StandardScaler()
bbbp_scaler1.fit(bbbp_descriptors_df.values)
bbbp_descriptors_df = pd.DataFrame(bbbp_scaler1.transform(bbbp_descriptors_
                                   columns=bbbp_descriptors_df.columns)
```

In [33]:
```
NCA1 = 100
NCA2 = 50
DROPRATE = 0.2
EP = 20
BATCH_SIZE = 128
VAL_RATIO = 0.1
TEST_RATIO = 0.1
```

In [37]:
```
nca = NCA1
cn = ['col'+str(x) for x in range(nca)]
```

In [38]:
```
bbbp_transformer1 = KernelPCA(n_components=nca, kernel='rbf', n_jobs=-1)
bbbp_transformer1.fit(bbbp_descriptors_df.values)
bbbp_descriptors_df = pd.DataFrame(bbbp_transformer1.transform(bbbp_descri
                                   columns=cn)
print(bbbp_descriptors_df.shape)
bbbp_descriptors_df.head()
```

```
(2039, 100)
```

Out[38]:

	col0	col1	col2	col3	col4	col5	col6	col7
0	0.111545	0.392873	-0.003267	-0.267306	0.231641	0.444553	-0.071575	0.110802
1	-0.203233	-0.044515	0.076816	-0.029671	-0.210262	0.099158	0.029858	-0.013679
2	-0.025130	-0.191458	-0.203020	0.068156	0.073111	-0.027518	-0.036239	0.146615
3	-0.359887	-0.032916	0.080973	-0.096383	-0.261437	0.124236	-0.007101	-0.017018
4	0.191021	-0.175647	-0.227526	0.267293	0.041106	0.061673	-0.242964	-0.157042

5 rows × 100 columns

In [40]:
```
X_train, X_test, y_train, y_test = train_test_split(bbbp_descriptors_df.val
                                                     test_size=TEST_RATIO,
                                                     random_state=42,strati
```

In [41]:
```
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train,
                                                       test_size=VAL_RATIO,
                                                       random_state=42,strat
```

In [42]:
```
def Find_Optimal_Cutoff(target, predicted):
    fpr, tpr, threshold = roc_curve(target, predicted)
    i = np.arange(len(tpr))
    roc = pd.DataFrame({'tf' : pd.Series(tpr-(1-fpr), index=i), 'threshold
    roc_t = roc.iloc[(roc.tf-0).abs().argsort()[:1]]

    return list(roc_t['threshold'])
```

In [43]:
```
def Find_Optimal_threshold(target, predicted):
    target = target.reshape(-1,1)
    predicted = predicted.reshape(-1,1)

    rng = np.arange(0.0, 0.99, 0.001)
    f1s = np.zeros((rng.shape[0],predicted.shape[1]))
    for i in range(0,predicted.shape[1]):
        for j,t in enumerate(rng):
            p = np.array((predicted[:,i])>t, dtype=np.int8)
            scoref1 = f1_score(target[:,i], p, average='binary')
            f1s[j,i] = scoref1

    threshold = np.empty(predicted.shape[1])
    for i in range(predicted.shape[1]):
        threshold[i] = rng[int(np.where(f1s[:,i] == np.max(f1s[:,i]))[0][0

    return threshold
```

In [44]:
```
parameters = {'kernel':['sigmoid', 'rbf'], 'C':[1,0.5], 'gamma':[1/nca,1/np
bbbp_svc = GridSearchCV(SVC(random_state=23,class_weight='balanced'), paran
```

In [45]:
```
result = bbbp_svc.fit(X_train, y_train)
```

```
In [46]: print(result.best_estimator_)
```

```
SVC(C=1, class_weight='balanced', gamma=0.1, probability=True, random_state
=23)
```

```
In [47]: print(result.best_score_)
```

```
0.8719306019701276
```

```
In [48]: pred = bbbp_svc.predict_proba(X_valid)
```

```
In [49]: bbbp_svc_calib = CalibratedClassifierCV(bbbp_svc, cv='prefit')
         bbbp_svc_calib.fit(X_valid, y_valid)
```

```
Out[49]: CalibratedClassifierCV(base_estimator=GridSearchCV(cv=5,
                                                             estimator=SVC(class_weig
ht='balanced',
                                                                              random_sta
te=23),
                                                             n_jobs=-1,
                                                             param_grid={'C': [1, 0.5
],
                                                                          'gamma': [0.
01,
                                                                                    0.
1],
                                                                          'kernel': ['
sigmoid',
                                                                                     '
rbf'],
                                                                          'probability
': [True]},
                                                             scoring='roc_auc'),
                                cv='prefit')
```

```
In [50]: pred = bbbp_svc_calib.predict_proba(X_valid)
         pred = pred[:,1]
         pred_svc_t = np.copy(pred)
```

```
In [51]: threshold = Find_Optimal_threshold(y_valid, pred)
         print(threshold)
```

```
[0.383]
```

```
In [52]: pred = bbbp_svc_calib.predict(X_test)
         f1_score(y_test,pred)
```

```
Out[52]: 0.9102167182662538
```

```
In [53]: pred = bbbp_svc_calib.predict_proba(X_test)
         roc_auc_score(y_test,pred[:,1])
```

```
Out[53]: 0.9154647435897436
```

In [54]:
```
pred = pred[:,1]
pred_svc = np.copy(pred)
pred[pred<=threshold] = 0
pred[pred>threshold] = 1
svc_score = f1_score(y_test,pred)
print(svc_score)
```

```
0.915151515151515
```

In [55]:
```
y = np.array(bbbp_descriptors_df.loc[23].values).reshape(1, -1)
result = bbbp_svc.predict(y)
prob = bbbp_svc.predict_proba(y)
print(result)
print(prob)
print(int(prob[:,1]>threshold))
```

```
[0]
[[0.44247933 0.55752067]]
1
```

Neural Network Model

In [56]:
```
bbbp_model = Sequential()
bbbp_model.add(Dense(128, input_dim=bbbp_descriptors_df.shape[1],
                     kernel_initializer='he_uniform'))
bbbp_model.add(BatchNormalization())
bbbp_model.add(Activation('tanh'))
bbbp_model.add(Dropout(rate=DROPRATE))
bbbp_model.add(Dense(64,kernel_initializer='he_uniform'))
bbbp_model.add(BatchNormalization())
bbbp_model.add(Activation('tanh'))
bbbp_model.add(Dropout(rate=DROPRATE))
bbbp_model.add(Dense(32,kernel_initializer='he_uniform'))
bbbp_model.add(BatchNormalization())
bbbp_model.add(Activation('tanh'))
bbbp_model.add(Dropout(rate=DROPRATE))
bbbp_model.add(Dense(1,kernel_initializer='he_uniform',activation='sigmoid
```

In [57]:
```
bbbp_model.compile(loss='binary_crossentropy', optimizer='adam',metrics=['a
```

In [58]:
```
checkpoint = ModelCheckpoint('bbbp_model.h5', monitor='val_loss', verbose=1
```

In [59]:
```
unique_classes = np.unique(bbbp_df['p_np'].values.flatten())
class_weights = class_weight.compute_class_weight('balanced',unique_classes
                                                   bbbp_df['p_np'].values.fl
class_weights = {unique_classes[0]:class_weights[0],unique_classes[1]:class
```

In [60]:
```
hist = bbbp_model.fit(X_train, y_train,
                      validation_data=(X_valid,y_valid),epochs=EP, batch_si
                      class_weight=class_weights ,callbacks=[checkpoint])
```

```
Epoch 1/20
 1/13 [=>............................] - ETA: 22s - loss: 0.7719 - accuracy
: 0.5391
```

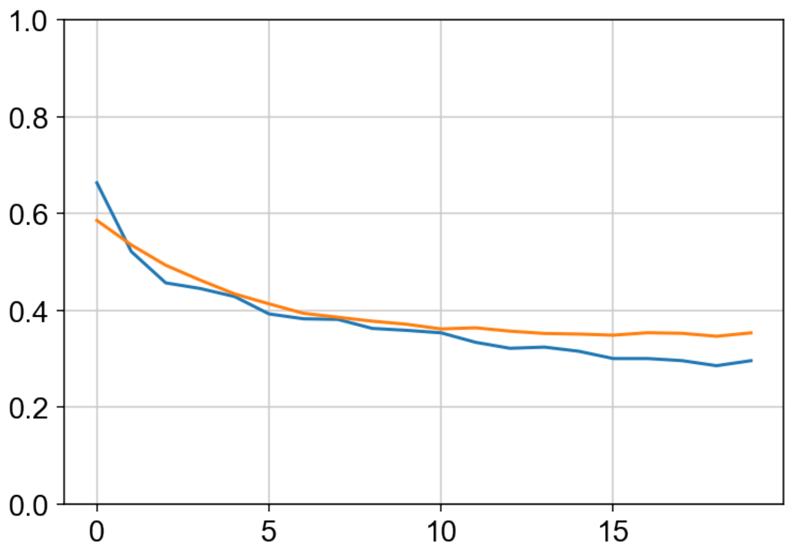
```
Epoch 1: val_loss improved from inf to 0.58582, saving model to bbbp_model.
h5
13/13 [==============================] - 2s 41ms/step - loss: 0.6636 - accu
racy: 0.5966 - val_loss: 0.5858 - val_accuracy: 0.7337
Epoch 2/20
10/13 [======================>.......] - ETA: 0s - loss: 0.5424 - accuracy:
0.6930
Epoch 2: val_loss improved from 0.58582 to 0.53519, saving model to bbbp_mo
del.h5
13/13 [==============================] - 0s 12ms/step - loss: 0.5214 - accu
racy: 0.7020 - val_loss: 0.5352 - val_accuracy: 0.7554
Epoch 3/20
10/13 [======================>.......] - ETA: 0s - loss: 0.4575 - accuracy:
0.7484
Epoch 3: val_loss improved from 0.53519 to 0.49317, saving model to bbbp_mo
del.h5
13/13 [==============================] - 0s 12ms/step - loss: 0.4567 - accu
racy: 0.7529 - val_loss: 0.4932 - val_accuracy: 0.7935
Epoch 4/20
 7/13 [===============>..............] - ETA: 0s - loss: 0.4640 - accuracy:
0.7634
Epoch 4: val_loss improved from 0.49317 to 0.46246, saving model to bbbp_mo
del.h5
13/13 [==============================] - 0s 14ms/step - loss: 0.4451 - accu
racy: 0.7595 - val_loss: 0.4625 - val_accuracy: 0.8261
Epoch 5/20
11/13 [========================>.....] - ETA: 0s - loss: 0.4343 - accuracy:
0.7727
Epoch 5: val_loss improved from 0.46246 to 0.43391, saving model to bbbp_mo
del.h5
13/13 [==============================] - 0s 10ms/step - loss: 0.4284 - accu
racy: 0.7741 - val_loss: 0.4339 - val_accuracy: 0.8478
Epoch 6/20
12/13 [==========================>...] - ETA: 0s - loss: 0.3943 - accuracy:
0.7956
Epoch 6: val_loss improved from 0.43391 to 0.41338, saving model to bbbp_mo
del.h5
13/13 [==============================] - 0s 11ms/step - loss: 0.3925 - accu
racy: 0.7971 - val_loss: 0.4134 - val_accuracy: 0.8533
Epoch 7/20
11/13 [========================>.....] - ETA: 0s - loss: 0.3775 - accuracy:
0.8288
Epoch 7: val_loss improved from 0.41338 to 0.39382, saving model to bbbp_mo
del.h5
13/13 [==============================] - 0s 11ms/step - loss: 0.3825 - accu
racy: 0.8262 - val_loss: 0.3938 - val_accuracy: 0.8587
Epoch 8/20
13/13 [==============================] - ETA: 0s - loss: 0.3812 - accuracy:
0.8225
Epoch 8: val_loss improved from 0.39382 to 0.38569, saving model to bbbp_mo
del.h5
13/13 [==============================] - 0s 9ms/step - loss: 0.3812 - accur
acy: 0.8225 - val_loss: 0.3857 - val_accuracy: 0.8587
Epoch 9/20
12/13 [==========================>...] - ETA: 0s - loss: 0.3646 - accuracy:
0.8281
Epoch 9: val_loss improved from 0.38569 to 0.37758, saving model to bbbp_mo
del.h5
13/13 [==============================] - 0s 9ms/step - loss: 0.3626 - accur
acy: 0.8328 - val_loss: 0.3776 - val_accuracy: 0.8533
Epoch 10/20
```

```
13/13 [==============================] - ETA: 0s - loss: 0.3586 - accuracy:
0.8304
Epoch 10: val_loss improved from 0.37758 to 0.37118, saving model to bbbp_m
odel.h5
13/13 [==============================] - 0s 9ms/step - loss: 0.3586 - accur
acy: 0.8304 - val_loss: 0.3712 - val_accuracy: 0.8641
Epoch 11/20
12/13 [==========================>...] - ETA: 0s - loss: 0.3575 - accuracy:
0.8301
Epoch 11: val_loss improved from 0.37118 to 0.36148, saving model to bbbp_m
odel.h5
13/13 [==============================] - 0s 10ms/step - loss: 0.3535 - accu
racy: 0.8310 - val_loss: 0.3615 - val_accuracy: 0.8587
Epoch 12/20
11/13 [========================>.....] - ETA: 0s - loss: 0.3459 - accuracy:
0.8594
Epoch 12: val_loss did not improve from 0.36148
13/13 [==============================] - 0s 8ms/step - loss: 0.3339 - accur
acy: 0.8625 - val_loss: 0.3638 - val_accuracy: 0.8587
Epoch 13/20
13/13 [==============================] - ETA: 0s - loss: 0.3215 - accuracy:
0.8413
Epoch 13: val_loss improved from 0.36148 to 0.35691, saving model to bbbp_m
odel.h5
13/13 [==============================] - 0s 9ms/step - loss: 0.3215 - accur
acy: 0.8413 - val_loss: 0.3569 - val_accuracy: 0.8696
Epoch 14/20
13/13 [==============================] - ETA: 0s - loss: 0.3239 - accuracy:
0.8443
Epoch 14: val_loss improved from 0.35691 to 0.35217, saving model to bbbp_m
odel.h5
13/13 [==============================] - 0s 9ms/step - loss: 0.3239 - accur
acy: 0.8443 - val_loss: 0.3522 - val_accuracy: 0.8696
Epoch 15/20
11/13 [========================>.....] - ETA: 0s - loss: 0.3195 - accuracy:
0.8601
Epoch 15: val_loss improved from 0.35217 to 0.35090, saving model to bbbp_m
odel.h5
13/13 [==============================] - 0s 10ms/step - loss: 0.3154 - accu
racy: 0.8613 - val_loss: 0.3509 - val_accuracy: 0.8533
Epoch 16/20
12/13 [==========================>...] - ETA: 0s - loss: 0.2989 - accuracy:
0.8698
Epoch 16: val_loss improved from 0.35090 to 0.34860, saving model to bbbp_m
odel.h5
13/13 [==============================] - 0s 10ms/step - loss: 0.3003 - accu
racy: 0.8680 - val_loss: 0.3486 - val_accuracy: 0.8641
Epoch 17/20
 1/13 [=>............................] - ETA: 0s - loss: 0.2948 - accuracy:
0.8516
Epoch 17: val_loss did not improve from 0.34860
13/13 [==============================] - 0s 6ms/step - loss: 0.3003 - accur
acy: 0.8571 - val_loss: 0.3537 - val_accuracy: 0.8533
Epoch 18/20
 1/13 [=>............................] - ETA: 0s - loss: 0.2723 - accuracy:
0.8516
Epoch 18: val_loss did not improve from 0.34860
13/13 [==============================] - 0s 6ms/step - loss: 0.2959 - accur
acy: 0.8667 - val_loss: 0.3524 - val_accuracy: 0.8587
Epoch 19/20
 1/13 [=>............................] - ETA: 0s - loss: 0.2190 - accuracy:
```

```
0.9062
Epoch 19: val_loss improved from 0.34860 to 0.34636, saving model to bbbp_m
odel.h5
13/13 [==============================] - 0s 8ms/step - loss: 0.2855 - accur
acy: 0.8649 - val_loss: 0.3464 - val_accuracy: 0.8750
Epoch 20/20
 1/13 [=>............................] - ETA: 0s - loss: 0.3882 - accuracy:
0.8750
Epoch 20: val_loss did not improve from 0.34636
13/13 [==============================] - 0s 6ms/step - loss: 0.2959 - accur
acy: 0.8752 - val_loss: 0.3534 - val_accuracy: 0.8587
```

In [61]:
```
plt.ylim(0., 1.0)
plt.plot(hist.epoch, hist.history["loss"], label="Train loss")
plt.plot(hist.epoch, hist.history["val_loss"], label="Valid loss")
```

Out[61]:
```
[<matplotlib.lines.Line2D at 0x7f7c50825580>]
```



In [62]:
```
bbbp_model.load_weights('bbbp_model.h5')
```

In [63]:
```
pred = bbbp_model.predict(X_valid)
pred_nn_t = np.copy(pred)
```

In [64]:
```
threshold = Find_Optimal_threshold(y_valid, pred)
print(threshold)
```

```
[0.227]
```

In [65]:
```
pred = bbbp_model.predict(X_test)
pred_nn = np.copy(pred)
roc_auc_score(y_test,pred)
```

Out[65]:
```
0.9026442307692307
```

In [66]:
```
pred[pred<=threshold] = 0
pred[pred>threshold] = 1
nn_score = f1_score(y_test,pred)
print(nn_score)
```

```
0.9207317073170732
```

In [67]:
```
prob = bbbp_model.predict(y)
print(prob)
print(int(prob>=threshold))
```

```
[[0.21266645]]
0
```

Gradient Boosting of Keras Model with SVC

In [68]:
```
inp = bbbp_model.input
out = bbbp_model.layers[-2].output
bbbp_model_gb = Model(inp, out)
```

In [69]:
```
X_train = bbbp_model_gb.predict(X_train)
X_valid = bbbp_model_gb.predict(X_valid)
X_test = bbbp_model_gb.predict(X_test)
```

In [70]:
```
data = np.concatenate((X_train,X_test,X_valid),axis=0)
```

In [71]:
```
bbbp_scaler2 = StandardScaler()
bbbp_scaler2.fit(data)
X_train = bbbp_scaler2.transform(X_train)
X_valid = bbbp_scaler2.transform(X_valid)
X_test = bbbp_scaler2.transform(X_test)
```

In [72]:
```
data = np.concatenate((X_train,X_test,X_valid),axis=0)
```

In [73]:
```
nca = NCA2
```

In [74]:
```
bbbp_transformer2 = KernelPCA(n_components=nca, kernel='rbf', n_jobs=-1)
bbbp_transformer2.fit(data)
X_train = bbbp_transformer2.transform(X_train)
X_valid = bbbp_transformer2.transform(X_valid)
X_test = bbbp_transformer2.transform(X_test)
```

In [75]:
```
nca = X_train.shape[1]
parameters = {'kernel':['sigmoid', 'rbf'], 'C':[1,0.5], 'gamma':[1/nca,1/np
bbbp_svc_gb = GridSearchCV(SVC(random_state=23,class_weight='balanced'), pa
```

In [76]:
```
result = bbbp_svc_gb.fit(X_train, y_train)
```

In [77]:
```
print(result.best_estimator_)
```

```
SVC(C=0.5, class_weight='balanced', gamma=0.02, probability=True,
    random_state=23)
```

In [78]:
```
print(result.best_score_)
```

```
0.9451653400764073
```

In [79]:
```
pred = bbbp_svc_gb.predict_proba(X_valid)
```

In [80]:
```
bbbp_svc_gb_calib = CalibratedClassifierCV(bbbp_svc_gb, cv='prefit')
bbbp_svc_gb_calib.fit(X_valid, y_valid)
```

Out[80]:
```
CalibratedClassifierCV(base_estimator=GridSearchCV(cv=5,
                                                   estimator=SVC(class_weig
ht='balanced',
                                                                 random_sta
te=23),
                                                   n_jobs=-1,
                                                   param_grid={'C': [1, 0.5
],
                                                               'gamma': [0.
02,
                                                                         0.
1414213562373095],
                                                               'kernel': ['
sigmoid',
                                                                          '
rbf'],
                                                               'probability
': [True]},
                                                   scoring='roc_auc'),
                       cv='prefit')
```

In [81]:
```
pred = bbbp_svc_gb_calib.predict_proba(X_valid)
pred = pred[:,1]
pred_svc_gb_t = np.copy(pred)
```

In [82]:
```
threshold = Find_Optimal_threshold(y_valid, pred)
print(threshold)
```

```
[0.294]
```

In [83]:
```
pred = bbbp_svc_gb_calib.predict(X_test)
f1_score(y_test,pred)
```

Out[83]:
```
0.9226006191950465
```

In [84]:
```
pred = bbbp_svc_gb_calib.predict_proba(X_test)
roc_auc_score(y_test,pred[:,1])
```

Out[84]:
```
0.9038461538461539
```

In [85]:
```
pred = pred[:,1]
pred_svc_gb = np.copy(pred)
pred[pred<=threshold] = 0
pred[pred>threshold] = 1
svc_gb_score = f1_score(y_test,pred)
print(svc_gb_score)
```

```
0.9221556886227544
```

In [86]:
```
y = np.array(X_train[23,:])
y = y.reshape(-1, nca)
result = bbbp_svc_gb_calib.predict(y)
prob = bbbp_svc_gb_calib.predict_proba(y)
print(result)
print(prob)
print(int(prob[:,1]>=threshold))
```

```
[1]
[[0.05011165 0.94988835]]
1
```

Gradient Boosting of Keras Model with XGBoost

In [90]:
```
parameters = {'learning_rate':[0.05,0.1,0.15],'n_estimators':[75,100,125],
               'booster':['gbtree','dart'],'reg_alpha':[0.,0.1,0.05],'reg_

bbbp_xgb_gb = GridSearchCV(XGBClassifier(random_state=32), parameters, cv=
```

In [91]:
```
result = bbbp_xgb_gb.fit(X_train, y_train)
```

In [92]:
```
print(result.best_estimator_)
```

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.1, max_delta_step=0, max_depth=3,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=75, n_jobs=0, num_parallel_tree=1, random_state=
32,
              reg_alpha=0.1, reg_lambda=0.5, scale_pos_weight=1, subsample=
1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

In [93]:
```
print(result.best_score_)
```

```
0.940998815998816
```

In [94]:
```
pred = bbbp_xgb_gb.predict_proba(X_valid)
```

In [95]:
```
bbbp_xgb_gb_calib = CalibratedClassifierCV(bbbp_xgb_gb, cv='prefit')
bbbp_xgb_gb_calib.fit(X_valid, y_valid)
```

Out[95]:
```
CalibratedClassifierCV(base_estimator=GridSearchCV(cv=5,
                                                   estimator=XGBClassifier(
base_score=None,

booster=None,

colsample_bylevel=None,

colsample_bynode=None,

colsample_bytree=None,
gamma=None,

gpu_id=None,

importance_type='gain',

interaction_constraints=None,

learning_rate=None,

max_delta_step=None,

max_depth=None,

min_child_weight=None,

missing=nan,

monotone_co...

random_state=32,

reg_alpha=None,

reg_lambda=None,

scale_pos_weight=None,

subsample=None,

tree_method=None,

validate_parameters=None,

verbosity=None),
                                                   n_jobs=-1,
                                                   param_grid={'booster': [
'gbtree',

'dart'],
                                                               'learning_ra
te': [0.05,

0.1,

0.15],
                                                               'max_depth':
[3,

4,

5],
                                                               'n_estimator
s': [75,

100,

125],
                                                               'reg_alpha':
[0.0,
0.1,

0.05],
                                                     'reg_lambda'
: [0.0,

0.1,

0.5,

1.0]},
                                        scoring='roc_auc'),
         cv='prefit')
```

In [96]:
```
pred = bbbp_xgb_gb.predict_proba(X_valid)
pred = pred[:,1]
pred_xgb_gb_t= np.copy(pred)
```

In [97]:
```
threshold = Find_Optimal_threshold(y_valid, pred)
print(threshold)
```

```
[0.348]
```

In [98]:
```
pred = bbbp_xgb_gb_calib.predict(X_test)
f1_score(y_test,pred)
```

Out[98]:
```
0.9259259259259259
```

In [99]:
```
pred = bbbp_xgb_gb_calib.predict_proba(X_test)
roc_auc_score(y_test,pred[:,1])
```

Out[99]:
```
0.9180021367521367
```

In [100…
```
pred = pred[:,1]
pred_xgb_gb = np.copy(pred)
pred[pred<=threshold] = 0
pred[pred>threshold] = 1
xgb_gb_score = f1_score(y_test,pred)
print(xgb_gb_score)
```

```
0.9333333333333333
```

In [101…
```
result = bbbp_xgb_gb_calib.predict(y)
prob = bbbp_xgb_gb_calib.predict_proba(y)
print(result)
print(prob)
print(int(prob[:,1]>=threshold))
```

```
[1]
[[0.05323249 0.94676751]]
1
```

In [102…
```
pred = (pred_svc_t+pred_nn_t.flatten()+pred_svc_gb_t+pred_xgb_gb_t)/4
```

In [103…
```
threshold = Find_Optimal_threshold(y_valid, pred)
print(threshold)
```

```
[0.202]
```

In [104…

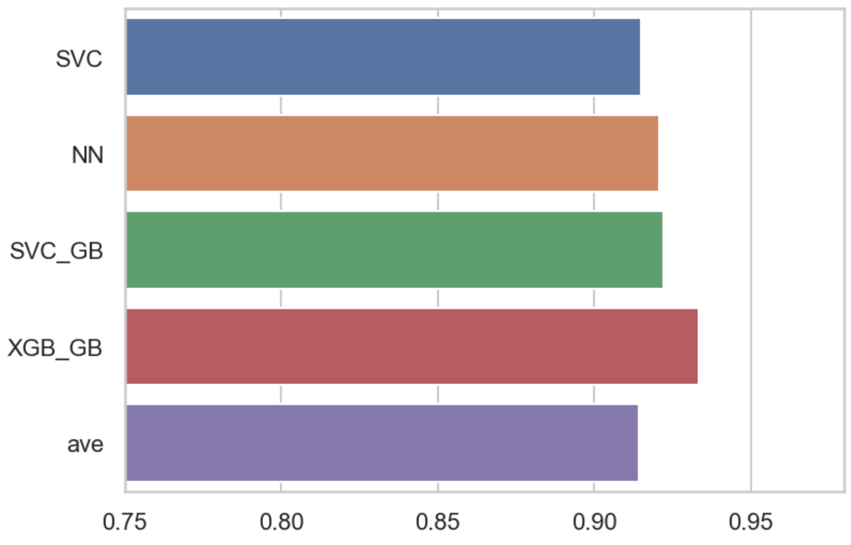
```
pred = (pred_svc+pred_nn.flatten()+pred_svc_gb+pred_xgb_gb)/4
pred[pred<=threshold] = 0
pred[pred>threshold] = 1
ave_score = f1_score(y_test,pred)
```

In [106…

```
sns.set(style="whitegrid")
ax = sns.barplot(x=[svc_score,nn_score,svc_gb_score,xgb_gb_score,ave_score
                 y=['SVC','NN','SVC_GB','XGB_GB','ave'])
ax.set(xlim=(0.75, None))
```

Out[106…

```
[(0.75, 0.98)]
```



In [ ]: