CS 573 Final Report: Toxic Molecule Prediction

Yifan Fei
(Zayne Carrick on Kaggle)

Computer Science
Purdue University

Instructor: Prof. Bruno Ribeiro

Spring 2018

Section 0:

1

Yes, I discussed the project with others but came up with my own answers. Their name(s) are Yupeng Han and Meng Liu.

2

I have used online resources to help me answer this question, but I came up with my own answers. Here is a list of the websites I have used in this homework:

XGBoost parameter tuning: https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

Imbalanced dataset: https://stats.stackexchange.com/questions/243207/what-is-the-proper-usage-of-scale-pos-weight-in-xgboost-for-imbalanced-datasets

XGboost Tutorial: http://xgboost.readthedocs.io/en/latest/parameter.html

XGBoost code analysis: https://blog.csdn.net/chedan541300521/article/details/54895880

Section 1: Feature Engineering

0: Data summarization

First of all, we should get familiar with the data type and all the statistical information. As is shown by pandas, the original training data has 12 columns including 2 ID columns, (index and inchi key), 7 numerical columns(Maximum Degree, Minimum Degree, Molecular Weight, Number of H-Bond Donors, Number of Rings, Number of Rotatable Bonds and Polar Surface Area), 2 graph features(Graph and smiles) and 1 target column.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7464 entries, 0 to 7463
Data columns (total 12 columns):
index                       7464 non-null int64
Maximum Degree              7464 non-null float64
Minimum Degree              7464 non-null float64
Molecular Weight            7464 non-null float64
Number of H-Bond Donors     7464 non-null float64
Number of Rings             7464 non-null float64
Number of Rotatable Bonds   7464 non-null float64
Polar Surface Area          7464 non-null float64
inchi_key                   7464 non-null object
Graph                       7464 non-null object
smiles                      7464 non-null object
target                      7464 non-null int64
dtypes: float64(7), int64(2), object(3)
memory usage: 699.8+ KB
```

(a) train dataset information

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1867 entries, 0 to 1866
Data columns (total 11 columns):
index                       1867 non-null int64
Maximum Degree              1867 non-null float64
Minimum Degree              1867 non-null float64
Molecular Weight            1867 non-null float64
Number of H-Bond Donors     1867 non-null float64
Number of Rings             1867 non-null float64
Number of Rotatable Bonds   1867 non-null float64
Polar Surface Area          1867 non-null float64
inchi_key                   1867 non-null object
Graph                       1867 non-null object
smiles                      1867 non-null object
dtypes: float64(7), int64(1), object(3)
memory usage: 160.5+ KB
```

(b) test dataset information

Figure 1: dataset information

	index	Maximum Degree	Minimum Degree	Molecular Weight	\
count	7464.000000	7464.000000	7464.000000	7464.000000	
mean	3731.500000	3.288183	0.875804	278.941072	
std	2154.815537	0.556245	0.390472	169.849479	
min	0.000000	0.000000	0.000000	41.053000	
25%	1865.750000	3.000000	1.000000	164.248000	
50%	3731.500000	3.000000	1.000000	240.444000	
75%	5597.250000	4.000000	1.000000	346.467000	
max	7463.000000	4.000000	2.000000	1950.681000	

	Number of H-Bond Donors	Number of Rings	Number of Rotatable Bonds	\
count	7464.000000	7464.000000	7464.000000	
mean	1.252412	1.728162	4.208199	
std	2.012638	1.679037	4.525664	
min	0.000000	0.000000	0.000000	
25%	0.000000	1.000000	1.000000	
50%	1.000000	1.000000	3.000000	
75%	2.000000	3.000000	6.000000	
max	36.000000	30.000000	47.000000	

	Polar Surface Area	target
count	7464.000000	7464.000000
mean	60.600695	0.040729
std	62.155964	0.197675
min	0.000000	0.000000
25%	26.300000	0.000000
50%	46.530000	0.000000
75%	77.430000	0.000000
max	1095.850000	1.000000

	inchi_key	Graph	\
count	7464	7464	
unique	7464	5086	
top	NCGC00257173-01	[((0; 1); 1.0); ((1; 2); 1.5); ((2; 3); 1.5); ...	
freq	1	28	

	smiles
count	7464
unique	6215
top	CC1=CC=C(O)C=C1
freq	5

Figure 2: describion on training data

In addition, notice that there are no missing value issues in this project. But there are duplicate data when you can see the uniqueness issue of smiles and Graph if you neglect ID columns. We will try both deleting duplicate data and retaining duplicate data in the cross validation later to see which one is better.

1: Data visualization

This part is mainly done with seaborn package. For the original features, if they are numerical, I did box-plot to see the distribution. I also plot the histogram for each feature under the condition whether the target is one or zero. This comparison helps us see the feature effect clearly. The plots are shown below.

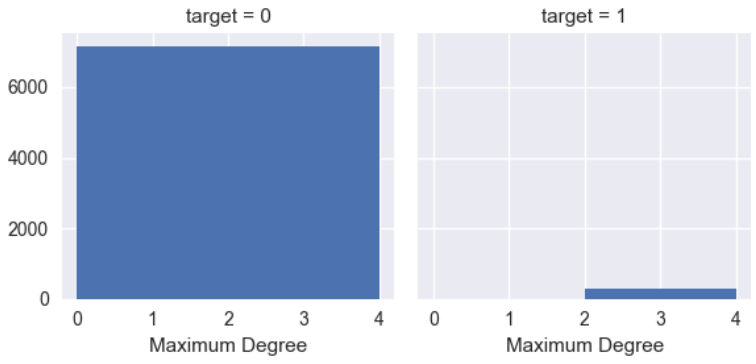

Figure 3: Maximum Degree vs target

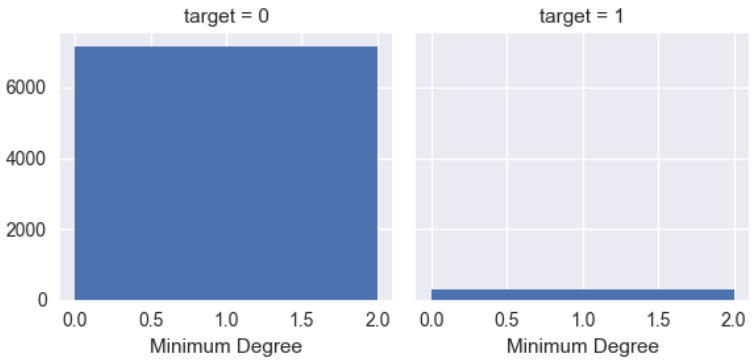

Figure 4: Minimum Degree vs target

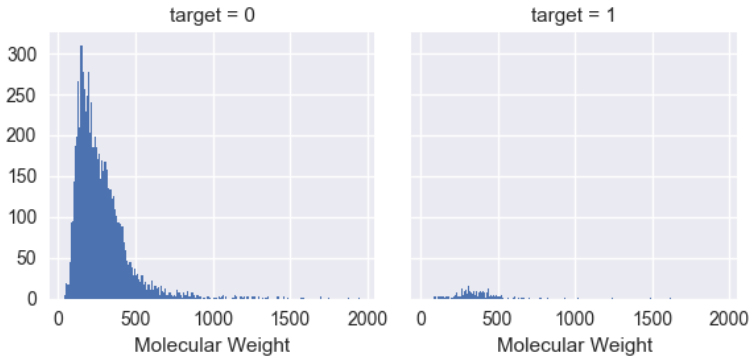

Figure 5: Molecular Weight vs target

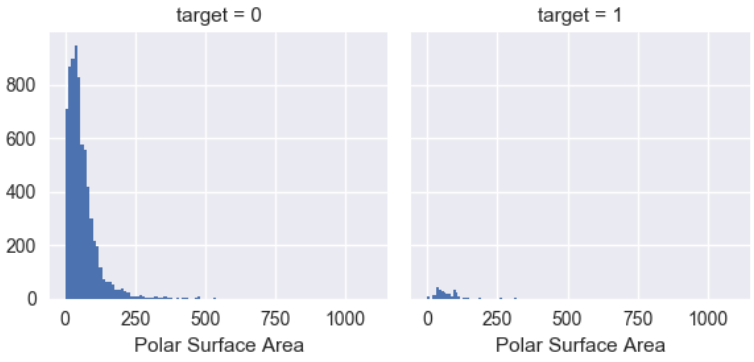

Figure 6: Polar Surface Area vs target

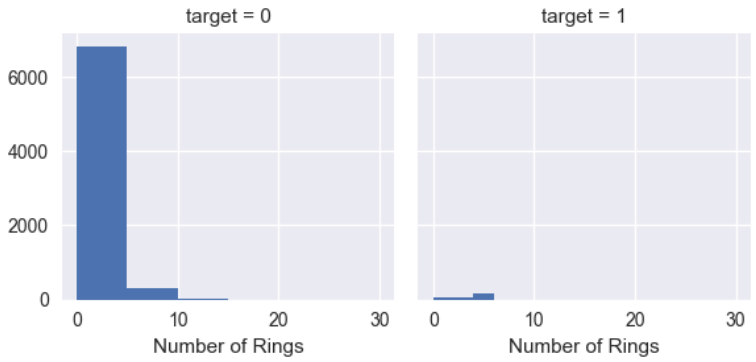

Figure 7: Number of rings vs target

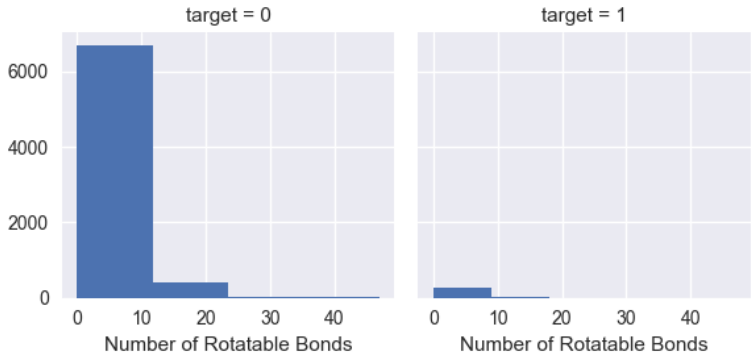

Figure 8: Number of Rotatable Bonds vs target

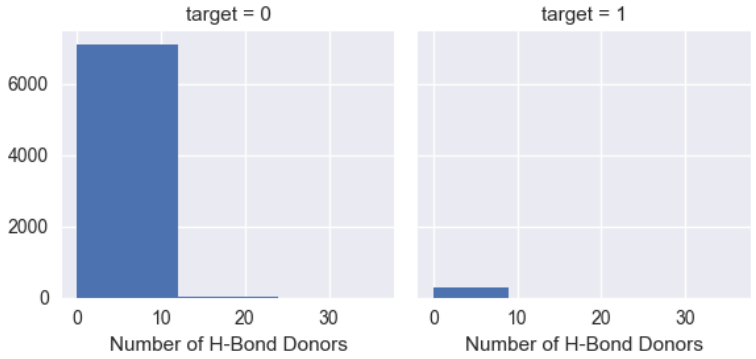

Figure 9: Number of H-Bond-Donors vs target

To analyze the correlation between features, I also plot the scatter plots by correlation matrix to find some relationship on original features. Pick one column and one row we can easily find the relationship between two features. Here the green points are toxic and blue points are not toxic. This will be useful for feature merging in the next part.

Figure 10: correlation matrix

2: Input Columns: Feature list

This part will introduce about how the features were generated and merged to get the final training and testing dataframe. As is shown by pandas, the original training data has 12 columns including 2 ID columns, (index and inchi key), 7 numerical columns(Maximum Degree, Minimum Degree, Molecular Weight, Number of H-Bond Donors, Number of Rings, Number of Rotatable Bonds and Polar Surface Area), 2 graph features(Graph and smiles) and 1 target column. Igraph and radit package was used to deal with the graph features.

First of all, the maximum vertex betweenness and the maximum edge betweenness for each molecule was calculated by igraph package as two new features. Betweenness is a statistical topology metric of the centrality of a node for the molecular networks. Specifically, it is the sum of the fractions of shortest paths that pass through a node. In other words, betweenness centrality is a measure of the extent that a node lays on the paths between other nodes. This measure is important because it may indicate the influence within the network that this node plays in controlling information transfer between other nodes. Thus I extract betweenness list from each molecule and get the maximum value, minimum value, mean value and standard deviation of the numerical list and do cross validation to determine if they are useful. Finally, I get the maximum vertex betweenness and the maximum edge betweenness as two features. The python code is attached.

```
train_df['Vertex_betweenness'] = train_df[
    'igraph'].map(lambda x: x.betweenness())
train_df['Edge_betweenness'] = train_df[
    'igraph'].map(lambda x: x.edge_betweenness())

train_df['Vertex_betweenness'] = train_df[
    'Vertex_betweenness'].apply(np.asarray)
train_df['Max_Vertex_betweenness'] = train_df[
    'Vertex_betweenness'].apply(np.max)
train_df['Min_Vertex_betweenness'] = train_df[
    'Vertex_betweenness'].apply(np.min)
train_df['Mean_Vertex_betweenness'] = train_df['Vertex_betweenness'].apply(np.mean)

train_df['Edge_betweenness'] = train_df['Edge_betweenness'].apply(np.asarray)
train_df['Max_Edge_betweenness'] = train_df['Edge_betweenness'].apply(np_max)
train_df['Min_Edge_betweenness'] = train_df['Edge_betweenness'].apply(np_min)
train_df['Mean_Edge_betweenness'] = train_df['Edge_betweenness'].apply(np_mean)
```

Figure 11: vertex betweenness and edge betweenness

Then I try to merge existed features like getting the ratio of Molecular Weight and Polar Surface Area based on the cross relationship. In the same time I many new features were created. I tried the density, diameter, average path length, molecule entropy, average Polar area, average weight. I even use regular expression to extract the bond type information and get the percentage of single bond, double bond and triple bond for each molecule. Though I did nested 10-fold cross validation locally to test whether these new features were good or bad, and got some magic features locally, the results on lead board were not ideal. Many combinations were tried. The following pictures are the feature list and feature importance if we consider all the features.

```
train_df['average_path_length'] = train_df[
    'igraph'].map(lambda x: x.average_path_length())
train_df['density'] = train_df[
    'igraph'].map(lambda x: x.density())
train_df['diameter'] = train_df[
    'igraph'].map(lambda x: x.diameter())

train_df['number of vertex'] = train_df[
    'igraph'].map(lambda x: x.vcount())
train_df['number of edges'] = train_df[
    'igraph'].map(lambda x: x.ecount())

train_df['Average area'] = train_df[[
    'Polar Surface Area','number of vertex']].apply(Cal_density,axis=1)
train_df['Average weight'] = train_df[[
    'Molecular Weight','number of vertex']].apply(Cal_density,axis=1)
```

Figure 12: Generation of new features and merging

```
drop_list = ['inchi_key', 'smiles', 'Graph', 'mol', 'igraph', 'Bond_info', 'Vertex_betweenness',
             'Edge_betweenness','Min_Vertex_betweenness','m_fps','m_fps_intarray',
            'Min_Edge_betweenness',
             # 'Maximum Degree',
             # 'Minimum Degree',
             'number of vertex',
             'number of edges',
             'diameter', #  after drop perofrmance increase
             'density',
             # 'Number of Rings',
             'average_path_length',
             'Average area',
             # 'Polar Surface Area',
             'Average weight',
             # 'Molecular Weight',
             'Mean_Vertex_betweenness',
             'Mean_Edge_betweenness',
             'Bond1_percentage',
             'Bond2_percentage',
             'Bond3_percentage'
             ]
```

Figure 13: Feature list and Drop list

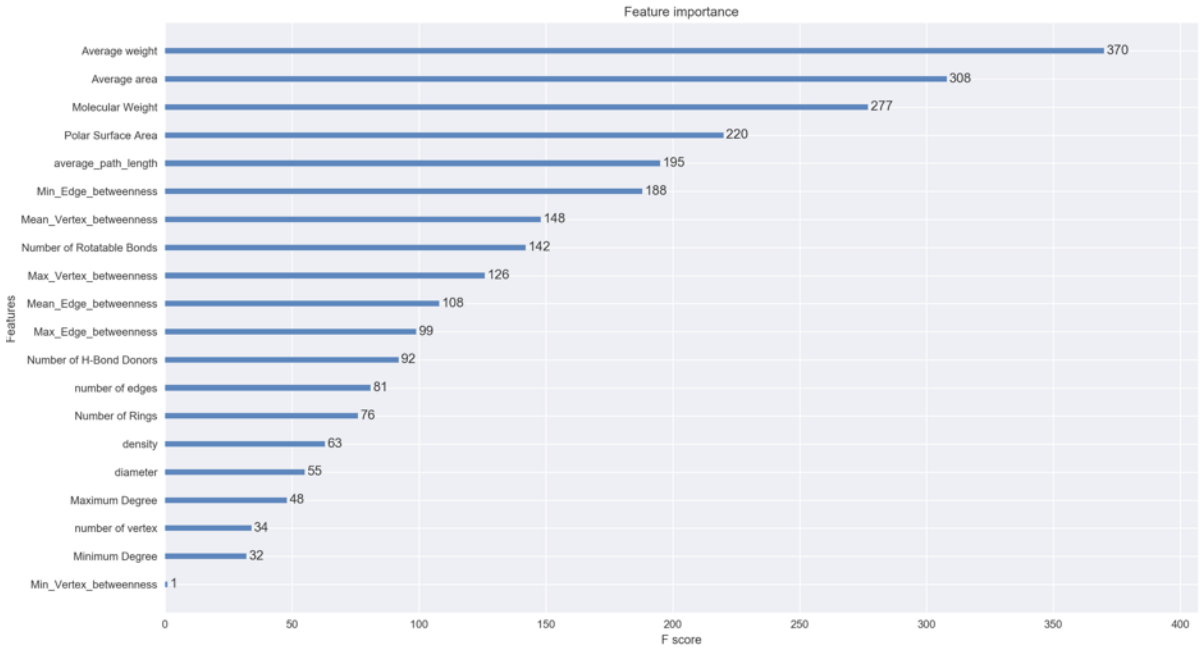

Figure 14: Feature importance

In the end, a magical feature, called the molecular fingerprint was calculated by radit package. Molecular fingerprints are representations of chemical structures originally designed to assist in chemical database substructure searching but later used for analysis tasks, such as similarity searching, clustering, and classification. There are many similarity metrics include Tanimoto, Dice, Cosine, Sokal, Russel, Kulczynski, McConnaughey, and Tversky. I first tried the default one, the Tanimoto similarity. I generate a 7464 by 7464 matrix to store the similarity between each two training data, and use these as 7464 features. The result is bad. The reason is that the similarity between nontoxic molecules is meaningless for this problem. Notice that there are only about 300 toxic molecules in training data. Then I try a similarity matrix, comparing all molecules with only toxic one. The code is attached. In this way it saves much computation because the features are less and meaningful. In addition I tried the maximum similarity value from the similarity list. It did not improve the result much somehow.

```python
def fingerprint(fps1,fps2_list):
    similarity_list = []
    for i in fps2_list:
        similarity = DataStructs.FingerprintSimilarity(fps1,i)
        similarity_list.append(similarity)
        max_sim = np.max(np.asarray(similarity_list))
        # mean_sim = np.mean(np.asarray(similarity_list))
    return max_sim

train_df['fps'] = train_df['mol'].apply(FingerprintMols.FingerprintMol)
print('Fingerprints')
toxic_list = train_df.groupby(['target']).get_group(1)['fps'].tolist()
    # print('toxic_list:',toxic_list)
train_df['Max_similarity'] = train_df['fps'].map(lambda x: fingerprint(x,toxic_list))
```

Figure 15: Generation of Tanimoto Similarity

Then I try another similarity. Morgan Fingerprints (Circular Fingerprints) is used here. This fingerprint is built by applying the Morgan algorithm to a set of user-supplied atom invariants. When generating Morgan fingerprints, the radius of the fingerprint must also be provided. Here radius is set to be 2. And my algorithm calculate the fingerprint as bit vectors. Here I use 2048 bits to represent every molecule by transfer the fingerprint into int array, and split the unique fingerprint feature into 2048 new features. It is amazing to see the bit stream of a molecule is used as the fingerprint. Clever way! I did not calculate the similarity then and just leave it to the model. From my perspective, the model can do more than just computing the similarity using inner product. In the end my guess was right. The progress was significant and improve my score on LB from 0.85 to 0.87.

```python
fp_len = 2048
fp_feature_list = []
for i in range(fp_len):
    fp_feature_list.append('m_fp_'+ str(i))

train_df['m_fps'] = train_df['mol'].map(lambda x: (AllChem.GetMorganFingerprintAsBitVect(x,2,nBits=fp_len)).ToBitString())
train_df['m_fps_intarray'] = train_df['m_fps'].apply(stringlist2intarray)
train_df[fp_feature_list] = pd.DataFrame(train_df['m_fps_intarray'].values.tolist(), columns=fp_feature_list)
```

Figure 16: Generation of Morgan Fingerprints

In conclusion, I used all the original features and maximum betweenness, maximum edge betweenness and Morgan Fingerprints. While all the other features have only one column, Morgan Fingerprints can have as many columns as the designer want. Here 2048 is enough to represent almost all molecule information. So the total features number is:

$$n_{feature} = 7 + 2 + 2048 = 2057$$

Section 2: Model: XGboost[1]

1: Score function derivation

XGBoost was used for this regression task.

The objective function is like:

$$Obj(\theta) = L(\theta) + \Omega(\theta)$$

The first part is the training loss, while the second part is Regularization(Also known as complexity of a tree). here I use logistic loss and L2 norm.

$$l(y_i, \hat{y}_i) = y_i ln(1 + e^{-\hat{y}_i}) + (1 - y_i) ln(1 + e^{\hat{y}_i})$$

$$\Omega(w) = \lambda||w||^2$$

So the objective function becomes:

$$Obj(y_i, \hat{y_i}) = \sum_{i=1}^{n} y_i ln(1 + e^{-w^T x_i}) + (1 - y_i) ln(1 + e^{w^T x_i}) + \lambda||w||^2$$

To include the complexity of the tree, it finally becomes:

$$Obj(y_i, \hat{y_i}) = \sum_{i=1}^{n} y_i ln(1 + e^{-w^T x_i}) + (1 - y_i) ln(1 + e^{w^T x_i}) + \gamma T + 1/2\lambda \sum_{j=1}^{T} ||w_j||^2$$

Complexity penalization is a novel point. Notice that compared with GBDT, XGBoost add regularization to the objective function so that it is less possible for the model to overfit the data. The regularization is the sum of the complexity penalties of the individual trees in the additive model. When the regularization parameter is zero, the objective is similar with the traditional gradient tree boosting.

The complexity can also be seen to be related to the relative difference of the leaf weights $w_1, ..., w_T$. For example, consider first a tree where all the weights are identical. This would be a globally constant model. If the weights are different, the model can be more complex.

The model is a random forest containing many small classification and regression trees. This kind of ensemble is like random forest algorithm using bagging. It introduces the regression tree, with one score in each leaf value, and the prediction is the sum of scores of all the tree. For example, if there are K trees:

$$\hat{y_i} = \sum_{k=1}^{K} f_k(x_i), f_k \in F$$

.

While $F = f(x) = w_q(x)(q : R^m \leftarrow T, w \in R^T)$.

Boosting Method: Newton Boosting

On this point, the difference between GBDT and XGBoost is that GBDT learns weights in real number space, xgboost learns functions in function space. This kind of boosting can be interpreted as a **Newton method in function space** and therefore name it Newton boosting. We can see that Newton's method is a second-order method while gradient descent is a first-order method.

Xgboost learns by boosting obviously. the prediction at round t is:

$$\hat{y_i}^t = \hat{y_i}^{(t-1)} + f_t(x_i)$$

So the objective function becomes:

$$Obj(y_i, \hat{y_i}) = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant$$

Rather than taking 1st-order Taylor expansion in GBDT, it takes 2nd-order Taylor expansion of the loss function:

$$l(y_i, \hat{y_i}^{(t-1)} + f_t(x_i)) = l(y_i, \hat{y_i}^{(t-1)}) + g_i f_t(x_i) + 1/2 h_i f_t^2(x_i)$$

where $g_i = \partial_{\hat{y_i}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)})$, $h_i = \partial^2_{\hat{y_i}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)})$

The learning of function only depend on the objective via g_i and h_i. After dropping the constant, the objective function becomes:

$$Obj(y_i, \hat{y_i}) = \sum_{i=1}^{n} [l(y_i, \hat{y_i}^{(t-1)}) + g_i f_t(x_i) + 1/2 h_i f_t^2(x_i)] + \Omega(f_t)$$

$$Obj(y_i, \hat{y_i}) = \sum_{i=1}^{n} [g_i w_q(x_i) + 1/2 h_i w_q^2(x_i)] + \gamma T + (1/2)\lambda \sum_{j=1}^{T} ||w_j||^2$$

Define the instance set in leaf j as $I_j = i|q(x_i) = j$

$$Obj(y_i, \hat{y_i}) = \sum_{i=1}^{n} [(\sum_{i \in I_j} g_i) w_j + 1/2 (\sum_{i \in I_j} h_i) w_j^2] + \gamma T + (1/2)\lambda \sum_{j=1}^{T} ||w_j||^2$$

Merging w_j together, we got the **final objective function** with trees as basis functions:

$$Obj(y_i, \hat{y_i}) = \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + 1/2 (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$

Notice that this equation is very similar with GBDT, but since we use 2nd order Taylor expansion this loss function is more complex. XGBoost uses gradient decent to find the minimum loss function. In both cases, we learn the tree structure first, and then learn the weight of the tree.

Learning the weight:

define $G_j = \sum_{i \in I_j} g_i$, $H_j = \sum_{i \in I_j} h_i$, and assume the structure of tree is fixed, we can get the optimal weight and **minimum loss by setting derivative as 0**:

$$w_j^* = -\frac{G_j}{H_j + \lambda}, Obj^* = -0.5 \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$$

Learning the Structure:

Then XGBoost use this minimum loss difference to add a split:

Gain = the score of left child + the score of right child - the score with no split

$$Gain = \frac{1}{2} [\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H - R + \lambda}] - \gamma$$

The greedy algorithm for splitting[2] use left to right linear scan over sorted instance to decide the best split along the feature. Notice that the gain can be negative when the training loss reduction is smaller than the regularization. So there is trade-off between simplicity and prediction accuracy. Usually, Pre-stopping in the traditional decision tree algorithm prevent overfitting by stopping split if the best split has negative gain, but some split may benefit future splits even if the gain is negative, so this is short-sighted. Post-Pruning in XGBoost recursively prune all the leaf splits with negative gain after growing a tree to maximum depth first. An example is that sometimes a split of negative loss say

-1 may be followed by a split of positive loss +10. decision tree would stop as it encounters -1. But XGBoost will go deeper first and it will see a combined effect of +9 of the split and keep both.

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ *to* m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted(I, by* $\mathbf{x}_{jk}$*)* **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

Figure 17: Exact Greedy Algorithm for Split Finding

Finally the Newton Boosting algorithm[2] is describe as:

Algorithm 3: Newton tree boosting

Input : Data set $\mathcal{D}$.
A loss function L.
The number of iterations M.
The learning rate η.
The number of terminal nodes T

1 Initialize $\hat{f}^{(0)}(x) = \hat{f}_0(x) = \hat{\theta}_0 = \arg\min_{\theta} \sum_{i=1}^{n} L(y_i, \theta)$;
2 **for** m = *1,2,..,M* **do**
3 $\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
4 $\hat{h}_m(x_i) = \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
5 Determine the structure $\{\hat{R}_{jm}\}_{j=1}^{T}$ by selecting splits which maximize $Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}\right]$;
6 Determine the leaf weights $\{\hat{w}_{jm}\}_{j=1}^{T}$ for the learnt structure by $\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}}$;
7 $\hat{f}_m(x) = \eta \sum_{j=1}^{T} \hat{w}_{jm} \mathrm{I}(x \in \hat{R}_{jm})$;
8 $\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$;
9 **end**

Output: $\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x)$

Figure 18: Newton Boosting

In conclusion, there are two optimization similar with GBDT. First, the model learns the tree structures which best fits the second-order Taylor expansion of the loss function. In the next equation we can see at each iteration, the loss is first scaled by Hessian, then $\frac{(\sum_{i \in I_j} g_i)}{(\sum_{i \in I_j} h_i)}$ is like the "gradient" in newton boosting.

$$w_j, I_j = argmin_{w_j, I_j} Obj(y_i, \hat{y_i}) = argmin_{w_j, I_j} \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + 1/2 (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$

$$w_j, I_j = argmin_{w_j, I_j} \sum_{j=1}^{T} (\sum_{i \in I_j} h_i [\frac{(\sum_{i \in I_j} g_i)}{(\sum_{i \in I_j} h_i)} w_j + 1/2 (1 + \frac{\lambda}{(\sum_{i \in I_j} h_i)}) w_j^2] + \gamma T$$

Next, the model learns the leaf weights to assign in the terminal nodes of the learned tree structure.

$$w_j^* = -\frac{G_j}{H_j + \lambda}, Obj^* = -0.5 \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$$

2: Parameters explanation and How they make influence on objective function

1. **min_child_weight [default=1]:** Defines the minimum sum of weights of all observations required in a child. This parameter is to prevent over-fitting. Higher values prevent a model from learning relations which may be highly specific to the particular sample selected for a tree. The size of the leaf weights $w_1, ..., w_T$ will also be related to model complexity as stated before. Smaller weight will yield models closer to the global constant, while larger weight will give us more complex models. For Newton boosting in XGBoost, the sum of observation weights is the sum of the Hessians in the terminal node.

2. **n_estimators:** Number of boosted trees to fit. This parameter n corresponds to the number of boosting iterations. Let F_n[2] denote the function space of an additive tree model consisting of n trees. Since the function space of the individual trees models are not closed under addition, we have that:

   $$F_1 \subset F_2 \subset ... \subset F_n$$

   Thus, increasing the number of iterations, i.e. adding more trees, will increase the representational ability and complexity of the model, but may overfit if it is too big.

3. **max_depth [default=6]:** As stated before in the Post-Pruning, it is the maximum depth of a single tree, used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample. Viewing it from the objective function, deeper depth means more number of terminal nodes T in the function below and you can see penalization is included in the objective.

   $$Obj(y_i, \hat{y_i}) = \sum_{i=1}^{n} y_i ln(1 + e^{-w^T x_i}) + (1 - y_i) ln(1 + e^{w^T x_i}) + \gamma T + 1/2 \lambda \sum_{j=1}^{T} |w_j|^2$$

   The more terminal nodes T the tree has, the more complex functions it can fit. As T increases, there will be fewer observations in each region, yielding to variance in estimation of the leaf weights.[**Xbg2**]

4. **gamma [default=0]:** A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.

   When regularization is not considered, the objective is like:

   $$Gain = \frac{1}{2} [\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{(G_L + G_R)^2}{H_L + H - R}] - \gamma$$

   Penalization of the number of terminal nodes will increase the probability of obtaining splits with negative gain. After pruning, we will thus tend to get more shallow trees. This parameter thus affects the learning of the tree structure.

5. **subsample [default=1]:** Denotes the fraction of observations to be randomly samples for each tree. Lower values make the algorithm more conservative and prevents over-fitting. Specifically, it is like bagging, which reduces the variance by random sampling. Note that subsampling draws a random sample of the data without replacement, while bootstrapping draws a random sample of the data with replacement. It is also called row subsampling.

   One equation to understand the effect of randomization parameter[2] is:

$$Var[\hat{f}(x)] = Var[\sum_{m=1}^{M} \hat{f_m}(x)] = \sum_{m=1}^{M} Var[\hat{f_m}(x)] + \sum_{m=1}^{M} \sum_{m' \neq m} Cov[\hat{f_m}(x), \hat{f'_m}(x)]$$

   Increasing randomization will increase the variance of individual trees, but decrease the co-variance between the trees. In the end, the overall variance of the ensemble model with many trees is reduced.

6. **colsample_bytree [default=1]:** Denotes the fraction of feature columns to be randomly samples for each tree. This is similar to random forest, which combines bagging with a randomized tree learning algorithm that only considers a random subset of the predictors each time a split is considered. By using part of features in different trees, it reduces the variance of the model through averaging. it is also called column subsampling.

7. **lambda [default=1]:** It is L2 regularization term on weights to prevent over-fitting. Here lambda controls the strength of the penalization.

$$Gain = \frac{1}{2}[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H - R + \lambda}] - \gamma$$

$$w_j^* = -\frac{G_j}{H_j + \lambda}, Obj^* = -0.5 \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T$$

   L2 regularization can be seen as a prior belief that the leaf weights should be small. Increasing the lambda means make the leaf weight smaller. This is like the shrink of learning rate. While learning rate shrinks all leaf weights by a same factor, the lambda will shrink the leaf weights by varying degree. Plus it will make influence on the structure on the tree because it changes the gain so that different splits end up been taken. Actually, it avoids split since the gain is smaller when lambda is bigger. In this case, L2 regularization increases the probability of obtaining a splits with negative gains. It can thus also affect the number of terminal nodes. In a word, L2 regularization not only change the leaf weight differently, but also alter the tree structure by changing the gain.

8. **scale_pos_weight [default=1]:** It deals with unbalanced data. It equals sum(negative cases) / sum(positive cases). The approach is called cost sensitive, the idea is to force the model to take care of the rare event by increasing the loss if it fails to correctly predict them. In this project the toxic molecule is much fewer than the nontoxic ones.

   Specifically, in the original code RegLossObj.GetGradient does this:

$$if(info.labels[i] == 1.0f)w* = param\_.scale\_pos\_weight$$

$$out\_gpair- > at(i) = bst\_gpair(Loss :: FirstOrderGradient(p, info.labels[i]) * w,$$
$$Loss :: SecondOrderGradient(p, info.labels[i]) * w);$$

   so a gradient of a positive sample would be more influential on the loss function.

```
class RegLossObj : public ObjFunction{
...
void GetGradient(const std::vector<bst_float> &preds,
                 const MetaInfo &info,
                 int iter,
                 std::vector<bst_gpair> *out_gpair) override {
...
    out_gpair->resize(preds.size());
    // check if label in range
    bool label_correct = true;
    // start calculating gradient
    const omp_ulong ndata = static_cast<omp_ulong>(preds.size());
    #pragma omp parallel for schedule(static)
    for (omp_ulong i = 0; i < ndata; ++i) {
      bst_float p = Loss::PredTransform(preds[i]);
      bst_float w = info.GetWeight(i);
      if (info.labels[i] == 1.0f) w *= param_.scale_pos_weight;
      if (!Loss::CheckLabel(info.labels[i])) label_correct = false;
      out_gpair->at(i) = bst_gpair(Loss::FirstOrderGradient(p, info.labels[i]) * w,
                                   Loss::SecondOrderGradient(p, info.labels[i]) * w);

    }
...
}
```

Figure 19: Original code of objective function related with scale_pos_weight

9. **max_delta_step**: The learning rate introduces a 'relative' regularization (multiplying the weight by a constant factor) but in extreme cases where hessian is nearly zero (like when we have very unbalanced classes) this isn't enough because the weights (in which computation the hessian is in denominator) becomes to nearly infinite. So what max_delta_steps do is to introduce an 'absolute' regularization capping the weight before apply learning rate correction. From the original code you can see the gradient dw is constrained in the range of

   $[-max\_delta\_step, max\_delta\_step]$, which is an absolute regularization.

```
// calculate the cost of loss function
template <typename TrainingParams, typename T>
XGBOOST_DEVICE inline T CalcGain(const TrainingParams &p, T sum_grad, T sum_hess) {
  if (sum_hess < p.min_child_weight) {
    return T(0.0);
}
  if (p.max_delta_step == 0.0f) {
    if (p.reg_alpha == 0.0f) {
      return Sqr(sum_grad) / (sum_hess + p.reg_lambda);
    } else {
      return Sqr(ThresholdL1(sum_grad, p.reg_alpha)) /
             (sum_hess + p.reg_lambda);
    }
  } else {
    T w = CalcWeight(p, sum_grad, sum_hess);
    T ret = sum_grad * w + T(0.5) * (sum_hess + p.reg_lambda) * Sqr(w);
    if (p.reg_alpha == 0.0f) {
      return T(-2.0) * ret;
    } else {
      return T(-2.0) * (ret + p.reg_alpha * std::abs(w));
    }
  }
}
```

(a) CalcGainGivenWeight()

```
// calculate weight given the statistics
template <typename TrainingParams, typename T>
XGBOOST_DEVICE inline T CalcWeight(const TrainingParams &p, T sum_grad,
                                   T sum_hess) {
  if (sum_hess < p.min_child_weight) {
    return 0.0;
}
  T dw;
  if (p.reg_alpha == 0.0f) {
    dw = -sum_grad / (sum_hess + p.reg_lambda);
  } else {
    dw = -ThresholdL1(sum_grad, p.reg_alpha) / (sum_hess + p.reg_lambda);
  }
  if (p.max_delta_step != 0.0f) {
    if (dw > p.max_delta_step) {
      dw = p.max_delta_step;
}
    if (dw < -p.max_delta_step) {
      dw = -p.max_delta_step;
}
  }
  return dw;
}
```

(b) CalcWeight()

Figure 20: Original code for max_delta_step

10. **objective [default=reg:linear]** here I use **binary:logistic**, which is logistic regression for binary classification and returns predicted probability.
11. **learning rate(shrinkage or eta)** $\epsilon(default = 0.3) : y^{(t)} = y^{(t)} + \epsilon f_t(x_i)$.usually set around 0.1 smaller than 1, which means that XGBoost do not make full optimization in each step and reserve chance for future rounds, it helps prevent overfitting and making the model more robust. However, lowering learning rate comes at the cost of greater computational demand.
12. **seed [default=0]:** It is the random number seed to generate reproducible results.

3: Hyperparameters Tunning

Grid search was used here to tune hyperparameters. Multiple hypothesis test was used here to help us compare the hyperparameter combination and choose the best one.

Tuning process:

1. To make the grid search efficient, first choose a relatively high learning rate and small iteration(n_estimator).
2. Tune tree-specific parameters (max_depth, min_child_weight, gamma, subsample, colsample_bytree) for decided learning rate and number of trees.
3. Tune regularization parameters (lambda, alpha) for XGBoost which can help reduce model complexity and enhance performance.
4. Lower the learning rate and raise the n_estimator.

Here is part of code about what I was tunning step by step.

```
def hyperparameter_tunning(train_df, test_df):
    # tunning

    X_train = train_df.drop(["target", 'index'], axis=1)
    Y_train = train_df["target"]

    param_test1 = {
     'max_depth':[1,2,3],
     'min_child_weight':[4,5,6],
    }
    gsearch1 = GridSearchCV(estimator = XGBRegressor(learning_rate =0.1, n_estimators=100, max_depth=5,
     min_child_weight=1, gamma=0, subsample=0.8, colsample_bytree=0.8,
     objective= 'binary:logistic', nthread=4, max_delta_step = 1, seed=27),
     param_grid = param_test1, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
    print('in test1')
    gsearch1.fit(X_train,Y_train)
    print(gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_)
```

Figure 21: tuning example 1

```
  param_test3 = {
      'gamma':[i/10.0 for i in range(0,5)]
      'subsample':[i/10.0 for i in range(6,10)],
      'colsample_bytree':[i/10.0 for i in range(6,10)]
      'reg_alpha':[0,0.1,1,10,100]
      'reg_lambda':[0.5,1,2,5],
      'max_delta_step': [1,3,5,7,9],
      # 'learning_rate':[0.1,0.05,0.02,0.01,0.005],
      # 'n_estimators':[100,200,500,1000,2000]
  }

  gsearch3 = GridSearchCV(estimator = XGBRegressor(learning_rate =0.1, n_estimators=100, max_depth=2,
   min_child_weight=5, gamma=0.4, subsample=0.8, colsample_bytree=0.8, reg_lambda = 1,
   objective= 'binary:logistic', nthread=4, max_delta_step = 1, seed=27),
   param_grid = param_test3, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
  print('test3')
  gsearch3.fit(X_train, Y_train)
  print(gsearch3.grid_scores_, gsearch3.best_params_, gsearch3.best_score_)
```

Figure 22: tuning example 2

The final parameter for the tuning is:

Table 1: **XGBoost Hyperparameters**

Hyperparameters	Default	My value
learning rate	0.3	0.05
n estimators	-	200
Max depth	6	4
Min child weight	1	8
gamma	1	9
subsample	1	0.85
colsample bytree	1	0.8
nthread	4	4
scale pos weight	1	11
max delta step	0	1

```
xgb = XGBRegressor(learning_rate=0.052338, n_estimators=200, max_depth=4, reg_lambda = 1,
            min_child_weight=8, gamma=9, subsample=0.8523367, colsample_bytree=0.8,
            objective='binary:logistic', nthread=4, max_delta_step = 1, scale_pos_weight=11,
            seed=27)
xgb.fit(X_train, Y_train)
Y_pred = xgb.predict(X_test)
```

Figure 23: Tunned parameter

3: Result and Conclusion

1: Result and Analysis

By using XGboost with just numerical features I achieve 0.85, by mining graph infomation I got 0.86, by adding sparse fingerprint features I got 0.87, and by parameter tuning I finally achieve 0.88347 on public leaderboard. The progress is natural and reasonable. However, I only got 0.82461 on private leaderboard, which means that I overfit the data on public leaderboard. The reason I guess is that I overdo the hyperparameter tunning. As taught in the class, multiple hypotheses will increase the likelihood a false hypothesis is declared true. Another reason is that I use 2048 bits for fingerprint, which makes the feature matrix sparse. That might leads to overfitting, too. Maybe 512 bits representation of a molecule is better.

2: How to improve

Try to use Neural network

Actually this project is extracted from a fingerprint paper. In that paper, the author used neural network and got good results. Neural network can generate more complex model and may help us got good grade.

Stacking Models

Stacking models is another way to ensemble. Stacking combine predictions from different models to generate a final prediction, and the more models we include the better it performs. Better still, because ensembles combine baseline predictions, they perform at least as well as the best baseline model. In this report I only use a single model, XGBoost. Though it is very powerful, stacking can theoretically improve the results as stated. Due to time limitation, I did not tune parameters on other models. But there is a python package called vecstack which can help us do stacking automatically if you set all the models' parameter properly. Here is the sturcture of the stacking for this project. For the first layer,

I may use 4 models, decison tree, random forest, logistic regression and xgboost. On the second layer, I use logistic regression to train the prediction of the first layer. The structure is very similar to a 2-layer-Neural-network.

```
# stacking

decision_tree = DecisionTreeRegressor()
random_forest = RandomForestRegressor(n_estimators=100)
logreg = LogisticRegression()

# Initialize 1-st level models.
models = [xgb, decision_tree, random_forest,logreg]

# Compute stacking features
def RMSLE(y, pred):
    return mean_squared_error(y, pred) ** 0.5

S_train, S_test = stacking(models, X_train, Y_train, X_test,regression = True,
                           n_folds = 5, shuffle = True, metric = RMSLE, random_state = 2018, verbose = 2)

# Initialize 2-nd level model
model = logreg

# Fit 2-nd level model
model = model.fit(S_train, Y_train)
```

Figure 24: stacking with 2 layer

More work on features

More features can be mined. I only use betweenness and fingerprint. Though in the feature engineering I create and merge many features, and get good result on my PC, they are discarded because of the bad performance on public leaderboard. What I learn from this experience is that I should trust my local cross validation more so that I may get better results. Public leaderboard score is just 30% of the test data and should not be payed too much attention on. Notice that some other students got relatively poor rank on public leaderboard but went ahead much on private leaderboard. I guess they must trust their cross validation more than the overfitting result on public leaderboard.

References

[1] Tianqi Chen. “XGBoost: A Scalable Tree Boosting System”. In: (2016).
[2] Didrik Nielsen. “Tree Boosting With XGBoost”. In: (2016).