Active Learning for Fast Drug Discovery

Anqi Cui* Jeff Schneider

March 2010
CMU-ML-10-103

Active Learning for Fast Drug Discovery

Anqi Cui[*] **Jeff Schneider**[†]

March 2010
CMU-ML-10-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]State Key Lab of Intelligent Technology and Systems, Department of Computer Science and Technology, Tsinghua University, Beijing, P.R.China
[†]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Keywords: Active learning, Bandit problems, Drug discovery, Function approximator, Gaussian process, Regression tree

Abstract

The drug discovery always costs a lot of time and money. Chemists have to do many experiments to screen useful compounds. In this paper, we present active learning algorithms for fast drug discovery. The algorithms decrease the number of experiments required to find out the best performance compounds among plenty of possible trials. The problem is a traditional exploration vs. exploitation dilemma and our approach is based on the multi-armed bandit problem and other function approximators. We propose the expected improvement estimation as a method to measure the unknown compounds. Some traditional models including UCB algorithms, Gaussian process, regression trees and so on are also used for our problem. Our results show that the algorithms present in this paper significantly raise the best performance of compounds found within a certain number of picks. The number of picks needed to first discover the best compound is also reduced to about half of random method's cost.

1 Introduction

The time and cost for the discovery and development of new therapeutics is currently too large. Before new drugs are able to be tested clinically, scientists have to develop assays for the target disease, screen up to a million compounds, undergo lead optimizations and test in animals. Each of these steps usually costs a long time and much money.

The key in the development is to find out the specific compounds that are effective to the certain bacteria or disease. Since the molecular structures of the compounds are usually very complicated, it is difficult for biochemists to fully understand the relationship between the structure and performance. Thus, a huge number of chemical tests have to be made, while perhaps most of them are needless.

In order to take less compounds screening, we develop new active learning algorithms that try to find out the compounds with higher performances in less tests. More formally, assume we have T, a large set of data points under consideration. Only a few data points in T, say $T_k \subset T$, are labeled (i.e. the performances are known). The algorithms could determine the data points $T_c \in T \backslash T_k$ that are chosen to be labeled (iteratively is allowed). With the new training data set $T_k \bigcup T_c$, the algorithms can find out the best data point $x^* \in T$, while $|T_c|$ is as small as possible[6].

The challenge in this problem is the small $|T_k|$ that leads to the exploration versus exploitation dilemma. If we consider we have enough information with current training data set, we can build our model to exploit the x^*. However, the model is often less fitted with few training data points, thus the exploitations predict badly. We have to explore for new data points with more unknown information. This results in more costs, but more accurate models. The balance between exploration and exploitation is important in active learning problems.

In this work, we first apply the Multi-armed Bandit Problem model to the problem. We also develop several function approximators to generate estimations for finding the best data point, including Gaussian Process Regression model, Regression Trees, SVMs, etc.

The features of the data points are generated by the chemical structures of the compounds. We use OpenBabel[2] to extract some common features used in QSAR works, including molecular weight (MW), logarithmic partition coefficient (logP), molar refractivity (MR), topological polar surface area (TPSA) and fingerprints[5] (FP2).The first four features are continuous values. The fingerprints are strings of 1,024 0/1-bits.

The report is organized as following: Section 1 is the introduction to the problem. Section 2 gives some related work in active learning problems. In section 3, we will describe the algorithms and methods we have developed. Section 4 gives a brief introduction of the dataset. We will show our experiment results in section 5 and the analysis will be given in section 6. The conclusion and future work will be discussed in section 7.

2 Related Work

Finding active compounds plays an important role in drug discovery. However, most compounds are usually inactive and lots of experiments are useless. Thus, active learning algorithms are applied to select the possible active compounds. In the work of M. K. Warmuth et al.[8], SVMs are used to classify the compounds as positive and negative. This method focuses more on the coverage of input space rather than the numerical performances of the compounds. As a goal of our problem, in order to discover the maximum performance of compounds, other models should be applied.

Bandit problems are one of the simplest forms of active learning. A multi-armed bandit problem is considered as an abstract model of active learning by introducing the ‘slot machines’; their rewards are stochastic. We shouldn’t believe that our current reward always represents the machine (esp. at the beginning), while to some extent, some hidden characteristics that machine really determine its reward overall. Therefore, we need to design the *policy*, or *allocation strategy* carefully to reach a higher overall reward (lower regret).

P. Auer et al.[1] designed several policies by associating a quantity called *upper confidence index*. They also proved that their policies achieve logarithmic regret uniformly over time, which is the lower bound in theory. In this paper, we first change our problem into a similar multi-armed bandit problem and then use these policies to solve it. We also examine the distribution of each arm and estimate the *expected improvement* to determine which arm to choose next.

However, bandit problems require us to divide the whole dataset into machines with different performances. The partition affects the performance a lot, because if both good data points and bad data points are put in a same machine, the machine will act as an average one – neither good nor bad. Thus, we consider each data point as a single machine; the machine is removed as soon as it is played once (i.e. choose without replacement). In order to estimate the rewards of unknown machines, we use some function approximators to calculate the expectations and their confidence intervals.

Using Gaussian process for non-linear regression is a simple method to estimate both the mean and variance of data points and usually involves only the choice of covariance and likelihood function. C.E. Rasmussen and C.K.I. Williams have shown detailed study of Gaussian process in the book[7]. In this paper, we’ll show our efforts on choosing appropriate parameters for the functions and the corresponding results.

Other common models used in machine learning areas, such as SVMs, neural networks and decision trees, don’t provide an estimation of confidence intervals explicitly. In the famous CART model (Classification And Regression Trees)[3], data points congregate around the leaves which reflect the samples’ distributions. For other models, there are few improvements for confidence intervals’ estimation.

3 Active Learning Algorithms

We provide two kinds of algorithms in this paper. One is based on the multi-armed bandit problem, and the other is based on function approximators. We compare our algorithms with the random strategy, which chooses data points randomly until it finds the best one (just by chance).

3.1 Algorithms for multi-armed bandit problem

For the multi-armed bandit problem method, we first divide the whole dataset T into different groups $T = \bigcup_{i=0}^{K} T_i$. Each group is considered as an arm (a slot machine) – a K-armed bandit problem. When an arm is pulled, the machine randomly picks a data point within its group and take the performance of the point as the reward. The data point is then removed before that arm is pulled again (pick without replacement).

3.1.1 UCB policies

The policies for choosing the machines we used in this paper are called UCB strategies[1], including UCB-tuned, UCB2 and ϵ_n-GREEDY.

Some of the UCB strategies involve parameters, including the α in UCB2 and the c, d in ϵ_n-GREEDY. In our experiments, we fixed α to 0.1 since the paper shows that "UCB2 is relatively insensitive to the choice of its parameter α". d is also set to

$$d = \mu^* - \max_{i:\mu_i<\mu^*} \mu_i$$

as the paper suggested. For the parameter c, we find that it's also insensitive according to our experiments.

As mentioned before, the strategy of dividing the dataset into groups is also important, since in multi-armed bandit problem, each machine is considered as a whole. If good data points and bad data points both occur in a same machine, it would be no much better than random strategies.

We divided our dataset depending on each of the four continuous features. For each feature, we tried both equal-frequency binning and equal-width binning.

Since we are more interested on the data points with higher performances, we also transformed their performances y into inverse logarithmic scale y', that is,

$$y' = \frac{-\log(1+\epsilon-y)+\log(1+\epsilon)}{\log(1+\frac{1}{\epsilon})} \tag{1}$$

where $0 < \epsilon \ll 1$ is used to map the original 1 to a very large number $-\log(\epsilon)$. Then its normalized to the original range $[0, 1]$. Figure 1 gives an illustrated explanation to this transformation. Not only in this figure, all the transformations related in this paper use $\epsilon = 0.01$.

3.1.2 Expected improvement estimate

The UCB policies estimate the Upper Confidence Bound of each arm. In order to put weight from the performance value, we use the *Expected Improvement* to estimate each arm. It is defined as the following:

$$\int_{y_{\text{best}}}^{+\infty} (t - y_{\text{best}}) \cdot \mathbf{pdf}(t)\mathrm{d}t \tag{2}$$

where y_{best} is the best performance y found so far and $\mathbf{pdf}(\cdot)$ is a specific probability density function. In this paper, we present two pdf's:

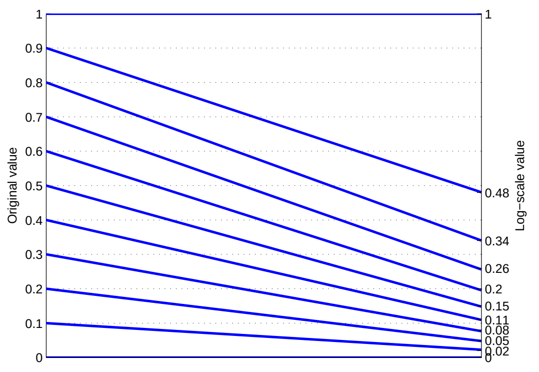


Figure 1: Inverse logarithmic transformation.

- Gaussian distribution: $\mathbf{pdf}(x) = f(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}$
- Gamma distribution: $\mathbf{pdf}(x) = f(x;k,\theta) = x^{k-1}\frac{e^{-x/\theta}}{\Gamma(k)\,\theta^k}$

The parameters in each distribution are determined by the samples we already picked from an arm $\{(x_i, y_i)\}$, i.e.,

- Gaussian distribution: $\mu = \bar{y}, \sigma = s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2}$
- Gamma distribution: $k = \bar{y}^2/s, \theta = s/\bar{y}$

3.2 Function approximators

For the function approximator method, the points in the dataset T is considered separately. We begin from randomly choosing a small set T_{init} of data points as the initial training data. Then the algorithm runs in iterations. In each iteration, with the currently picked dataset T_k, we use a function approximator to estimate the expected performances and their confidence intervals of all the unpicked data points (i.e. not labeled yet) in $T \backslash T_k$. Then we can get the *expected improvement* of each data point. The highest point will be picked and put into T_k and that ends this iteration. Figure 2 shows this algorithm.

The details of the function approximators are shown below.

3.2.1 Gaussian process regression

Gaussian process can be used for non-linear regression. We use the work[7] for our problem, as shown in the following.

```
Dataset:         $(x_i, y_i) \in T$
Initialization:  Randomly pick 10 data points to be the initial $T_k$.
                 The best data point found so far $y_{\text{best}} \leftarrow 0$
Loop:            Until $y_{\text{best}} = 1$ (the max possible value of $y$'s) or $T \backslash T_k = \phi$
1:               $T_{\text{train}} \leftarrow T_k$, $T_{\text{test}} \leftarrow T \backslash T_k$
2:               Use $T_{\text{train}}$ to train the function approximator $g$
3:               $\forall x_i \in T_{\text{test}}$, use $g$ to estimate the $\hat{y}_i$ (get the mean $\hat{\mu}_i$ and standard deviation $\hat{\sigma}_i$)
4:               $\forall x_i$ with $\hat{\mu}_i$ and $\hat{\sigma}_i$, calculate its expected improvement $s_i$
5:               Pick the point $x_j$ with the maximum $s_j$:
                 $T_k \leftarrow T_k \cup \{x_j\}$
                 if $y_j > y_{\text{best}}$:
                 $y_{\text{best}} \leftarrow y_j$
                 endif
```

Figure 2: Pseudocode of function approximator method.

Given inputs X, targets $\mathbf{y}$, test input $\mathbf{x}_*$ and the parameters of the Gaussian process model: k as covariance function and σ_n^2 as noise level, we can predict the targets of $\mathbf{x}_*$ at mean $\bar{f}_*$ and variance $\mathbb{V}[f_*]$, by:

$$\begin{aligned}\bar{f}_* &= \mathbf{k}_*^\top (K + \sigma_n^2 I)^{-1} \mathbf{y}, \\ \mathbb{V}[f_*] &= k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{k}_*^\top (K + \sigma_n^2 I)^{-1} \mathbf{k}_*\end{aligned}$$

We use the squared-exponential covariance function,

$$k_y(x_p, x_q) = \sigma_f^2 \exp(-\frac{1}{2l^2}(x_p - x_q)^2) + \sigma_n^2 \delta_{pq}$$

The hyperparameters are the signal variance σ_f, the noise variance σ_n and the length-scale l. Their values are determined by maximizing the likelihood function L. We use two likelihood functions, the log marginal likelihood L_M and the leave-one-out likelihood L_{LOO}, defined by:

$$\begin{aligned}L_M(X, \mathbf{y}, \theta) &= \log p(\mathbf{y}|X, \theta) = -\frac{1}{2}\mathbf{y}^\top K_y^{-1} \mathbf{y} - \frac{1}{2} \log |K_y| - \frac{n}{2} \log 2\pi \\ L_{LOO}(X, \mathbf{y}, \theta) &= \sum_{i=1}^{n} \log p(y_i | X, \mathbf{y}_{-i}, \theta)\end{aligned}$$

In step 2 and 3 of Figure 2, we first search[1] for the good hyperparameters that maximize the likelihood function. In order to avoid the local optimum, we start from different initial points in the space. After the hyperparameters are found, we use the equation above to get the mean and variance of T_{test}, thus calculate the expected improvement and make the decision.

The features involved in this method are the four continuous features. The fifth feature, fingerprints, consists of 1,024 bits. When taking each bit as a single feature, the input space becomes high dimensional. The searching algorithm mentioned above is very computational complicated for high dimension. Therefore, we use some other models to deal with this problem.

[1]The searching algorithm is also provided on book[7]'s website, http://www.gaussianprocess.org/gpml/code/matlab/doc/

```
Input:      $T_{\text{train}}, T_{\text{test}}$
Bootstrap:  Loop 30 times ($j = 1, 2, \ldots 30$):
  1:          Sample $|T_{\text{train}}|$ examples from $T_{\text{train}}$, with replacement. These bootstrap samples make up $T'_{\text{train},j}$
  2:          Use $T'_{\text{train},j}$ to generate a regression tree $g_j$
  3:          $\forall x_i \in T_{\text{test}}$, calculate which node in $g_j$ it belongs to and get the number of samples in that node $n_{ij}$,
              mean $\hat{\mu}_{ij}$ and variance $\hat{\sigma}^2_{ij}$
Aggregate:  $\forall x_i \in T_{\text{test}}$, calculate the overall estimated mean and variance by:
            $\hat{\mu}_i = \frac{\sum_j (\hat{\mu}_{ij} \cdot n_{ij})}{\sum_j n_{ij}}$
            $\hat{\sigma}^2_i = \frac{\sum_j (\hat{\sigma}^2_{ij}(n_{ij}-1) + n_{ij}(\hat{\mu}_i - \hat{\mu}_{ij})^2)}{(\sum_j n_{ij}) - 1}$
Output:     $\hat{\mu}_i, \hat{\sigma}_i$
```

Figure 3: Pseudocode of regression tree approximating method.

3.2.2 Classification problem

We first use SVM and dicision tree, since they work well on high dimensions. In order to mine the features thoroughly, We change the original regression model into a classification problem. We divided the dataset into two parts: after sorted by their performances, the top 10% data points are considered as positive points; the others are negative points. We tested the accuracy, precision and recall by a 10-fold cross validation.

The models we tested are SVM with linear kernel, SVM with Tanimoto kernel (defined by the Tanimoto coefficient[4]) and the classification tree. The features are the four continuous features and the fingerprints.

3.2.3 Regression trees with fingerprints

Since the means and variances are required to estimate the expected improvement, the regression tree model is used to calculate the estimations. Different from classification tree, regression tree holds training data on the leaves of the tree. Each leaf node contains several data points, thus has a mean and variance. For the test data points, each of them would belong to a specific leaf; the mean and variance of that leaf is used to calculate the overall mean and variance. This function approximator also works with the bagging method, as Figure 3 shows.

3.2.4 Bit-based integrals

Another method to deal with the 1,024 bits is to estimate each of them separately. For all the data points in the picked dataset T_k, different bits may have different number of 0's or 1's. If only a single value appears in the bit, we know nothing about that bit for the other value. In traditional information methods (either Entropy or Gini index), the model just exploits existing information. Since this bit contains no more information (i.e., the information gain is 0), it'll never be chosen as an interested bit for splitting nodes. However, in the exploration vs. exploitation problem, these bits should also be explored in case the other value makes improvement sometime. Thus, we need to put more weight on these bits so that the model would like to discover them.

Our method is to put a small prior on each bit. When calculating the variance of that bit, the denominator is a number smaller than 1, thus the estimated variance would be large and increase the

```
Input:       $T_{\text{train}}, T_{\text{test}}$
Parameters:  $0 < \epsilon_n, \epsilon_m, \epsilon_s < 1$
Loop:        For the $j$-th bit ($j = 1, 2, \ldots, 1024$) $x_i^{(j)}$ of all $x_i \in T_{\text{train}}$:
    1:       Divide the $T_{\text{train}}$ into two parts $T_{[0]}$ and $T_{[1]}$ where $T_{[a]} = \{(x_i, y_i) \in T_{\text{train}} | x_i^{(j)} = a\}$
    2:       Calculate $n_{[a],j}$, $\hat{\mu}_{[a],j}$, and $\hat{\sigma}^2_{[a],j}$ for $a = 0, 1$, where
             $n_{[a],j} = |T_{[a]}|$
             $\hat{\mu}_{[a],j} =$ the mean of $y_i$'s where $(x_i, y_i) \in T_{[a]}$
             $\hat{\sigma}^2_{[a],j} =$ the variance of $y_i$'s
             $*$ If $n_{[a],j}$ is 0, set $\hat{\mu}_{[a],j}$ and $\hat{\sigma}^2_{[a],j}$ to be zero.
    3:       Add the prior to these estimations:
             $n'_{[a],j} = n_{[a],j} + \epsilon_n$
             $\hat{\mu}'_{[a],j} = (\hat{\mu}_{[a],j} \cdot n_{[a],j} + \epsilon_m)/n'_{[a],j}$
             $\hat{\sigma'}^2_{[a],j} = \hat{\sigma}^2_{[a],j} + \epsilon_s$
    4:       Use equation 2 to integrate for this bit. The integrals are $s^{(j)}_{[a]}$.
Estimate:    $\forall x_i \in T_{\text{test}}$, calculate the overall expected improvement $s_i$ by equation 3.
Output:      $s_i$
```

Figure 4: Pseudocode of bit-based integrals method.

probability when calculating the integral of equation 2. The **pdf** we used is Student's t-distribution:

$$\textbf{pdf}(t) = f(t;\nu) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\,\Gamma(\frac{\nu}{2})}\left(1 + \frac{t^2}{\nu}\right)^{-(\frac{\nu+1}{2})}$$

where $t = \frac{x-\hat{\mu}}{\hat{\sigma}/\sqrt{n}}$, since this distribution puts more weight than Gaussian distribution for the undiscovered bits.

After we get the integrals of each bit ($s^{(j)}_{[a]}, a = 0, 1$), the overall expected improvement for each test data point is the sum of each bit's integral:

$$s_i = \sum_{j=1}^{1024} s^{(j)}_{[x_i^{(j)}]} \tag{3}$$

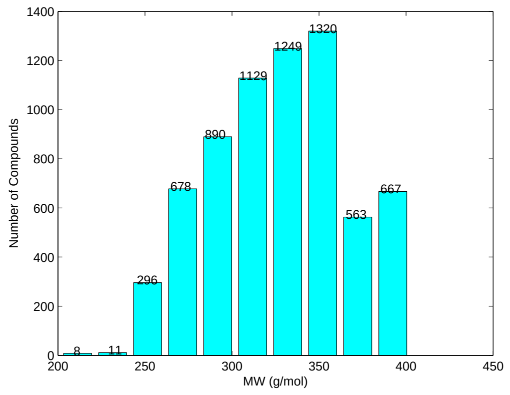
The pseudocode of this algorithm is shown in Figure 4.

4 Dataset

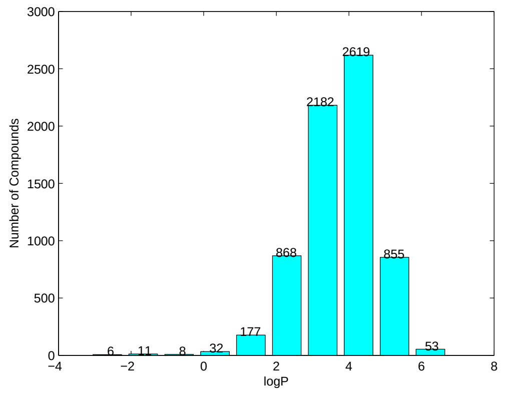
The high-throughput screening data were obtained and provided by Scott Franzblau in the Department of Medicinal Chemistry and Pharmacognosy at the University of Illinois at Chicago. The compounds for the study were purchased by the International Drug Discovery Institute (www.i-ddi.org).

There are 6,811 compound structures and their corresponding performances against a certain bacteria, presented in percentages. As mentioned before, the features we got from the compounds are MW, logP, MR, TPSA and the fingerprints (FP2). The figures shown in Figure 5 are the distributions of these features.

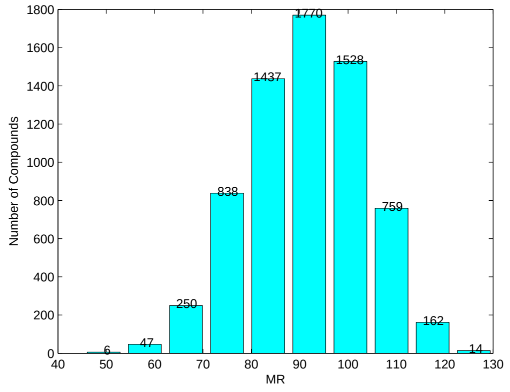
The target value is the percentage of each compound's performance against the certain bacteria. As shown in Figure 6, Most of the target values are small (zero).



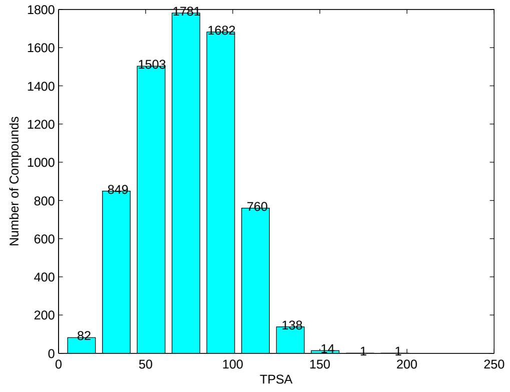
(a) MW



(b) logP



(c) MR



(d) TPSA

Figure 5: Distributions of the four features.

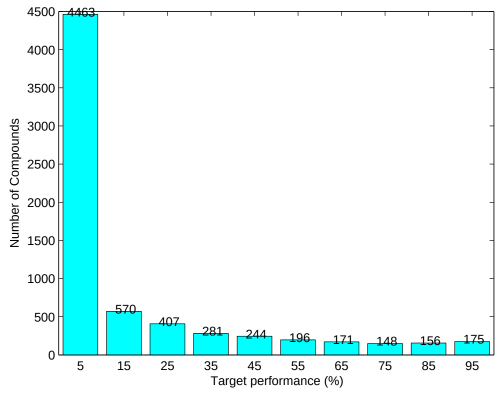


Figure 6: Distributions of the target.

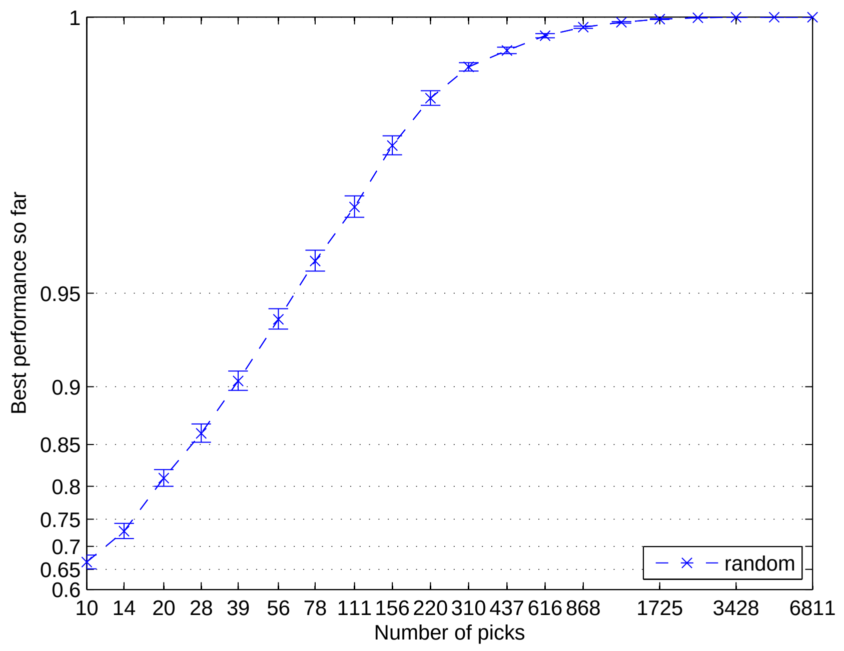


Figure 7: Random algorithm result.

The average of targets is 0.155 (15.5%), and the variance is about 0.066.

The goal of our problem is to find the compounds with the highest performance (100%) as fast as possible. However, there are only 7 such compounds. Theoretically, the average number of random picks required is $\frac{6811+1}{7+1} = 851.5$. In the following sections, we will show that our algorithms can significantly reduce it to about 300.

5 Experimental Results

In this section, we'll show the experimental results of different algorithms seperately. An overall comparison of good alogrithms will be shown in the next section.

Since the algorithms involve random selection, we run each of the algorithms for 1,000 times and get the average of its performances (unless otherwise noted). We also provide a 95% confidence interval of this average to show if the difference is significant.

Figure 7 shows the performance of random algorithm.

The figures in this section are all similar to this figure. The x-axis is the number of picks. Since most algorithms start with initial 10 picks randomly, the axes begin from 10. Some of the algorithms are computational complex, so we only run those algorithms to about 1,224 picks since the most important part to see the differences are from 10 to 1,000 picks. The y-axis shows the best performance found so far. In order to show the difference clearly, we draw them in the inverse log scale, so that the higher part is stretched to be wider.

Table 1: Partition info of feature TPSA

(a) Equal-width binning

Arm#	#Data	Feature range	Target range	Target mean
0	82	[3.24, 22.75]	[0, 0.9993]	0.2196
1	849	[23.47, 43.12]	[0, 0.999]	0.2130
2	1503	[43.26, 63.13]	[0, 1]	0.1910
3	1781	[63.21, 83.12]	[0, 0.9969]	0.1635
4	1682	[83.22, 103.1]	[0, 1]	0.1184
5	760	[103.11, 123.06]	[0, 1]	0.09196
6	138	[123.32, 142.45]	[0, 0.9888]	0.0710
7	14	[144.66, 159.51]	[0, 0.684]	0.1102
8	1	[166.79, 166.79]	[0, 0]	0.0
9	1	[202.98, 202.98]	[0.1353, 0.1353]	0.1353

(b) Equal-frequency binning

Arm#	#Data	Feature range	Target range	Target mean
0	682	[3.24, 39.44]	[0, 0.9993]	0.1867
1	681	[39.44, 49.85]	[0, 0.9992]	0.2091
2	681	[49.85, 57.34]	[0, 1]	0.2106
3	681	[57.34, 66.4]	[0, 0.9969]	0.1989
4	681	[66.4, 71.79]	[0, 0.9959]	0.1400
5	681	[71.79, 81.79]	[0, 0.9963]	0.1566
6	681	[81.79, 87.72]	[0, 1]	0.1248
7	681	[87.72, 97.64]	[0, 1]	0.1605
8	681	[97.64, 108.33]	[0, 0.9978]	0.0847
9	681	[108.33, 202.98]	[0, 1]	0.0808

5.1 Multi-armed bandit algorithms

5.1.1 UCB policies

In UCB algorithms, the dataset is divided into 10 groups according to the feature of each data point. Some statistical values of partitions with TPSA are listed in Table 1. Partitions with other features are similar.

We also take the inverse log transform on the target values so that the algorithm considers higher targets much better.

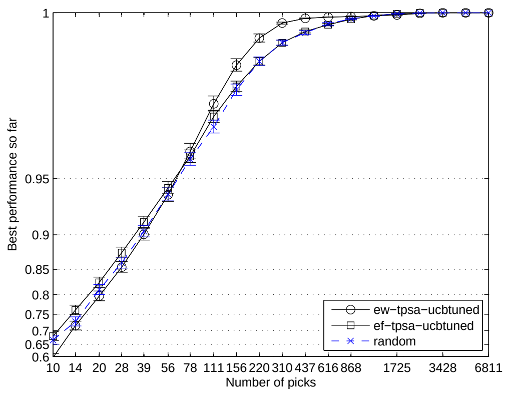
There are two ways for binning, four features, two kinds of target values (with or without inverse log transform) and three UCB algorithms. Therefore, there are $2 \cdot 4 \cdot 2 \cdot 3 = 48$ possible combinations of the experiment. However, we will only change one dimension at a time; hence only four figures are shown in Figure 8. The parameter c in ϵ_n-GREEDY policy is 0.1.

The parameter c in ϵ_n-GREEDY policy hardly affects the result, as shown in Figure 9.

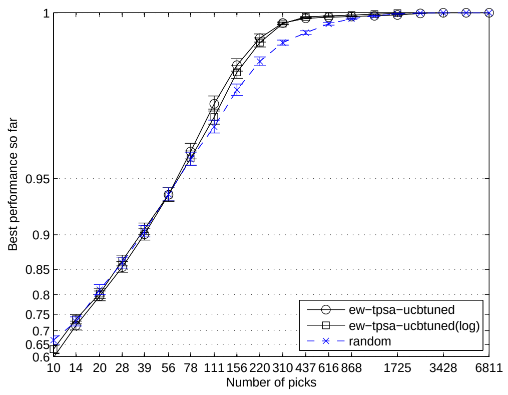
The results show the UCB algorithms raise the performance between about 100 picks and 1,000 picks. The best is UCB-tuned policy with equal-width binning and TPSA partition. It finds the same level of data point as random with only half picks ($616 \rightarrow 310$). However, before 100 picks, it doesn't show a significant improvement than random.

5.1.2 Expected improvement estimate

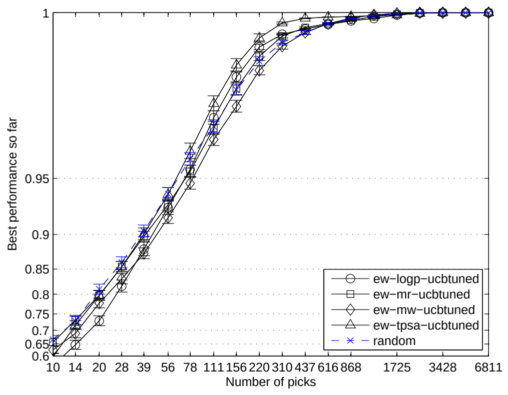
Figure 10 shows the results of expected improvement estimate algorithm.



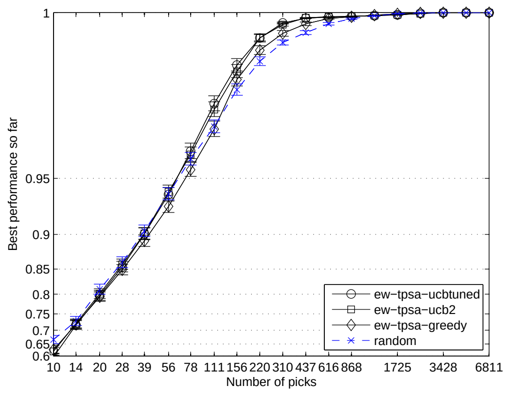
(a) Equal-width/frequency



(b) nolog/log



(c) Four features



(d) UCB policies

Figure 8: Comparison of UCB algorithms.

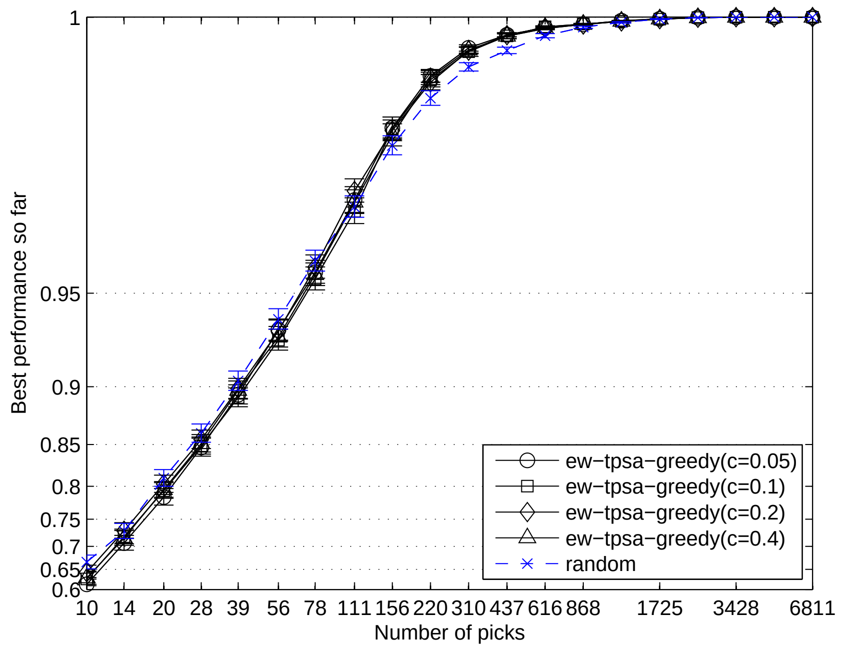


Figure 9: Different parameter c in ϵ_n-GREEDY policy.

The best combination, equal-width binning, TPSA feature and Gaussian distribution, performs better than random at as early as about 50 picks.

5.2 Function approximators

5.2.1 Gaussian process regression

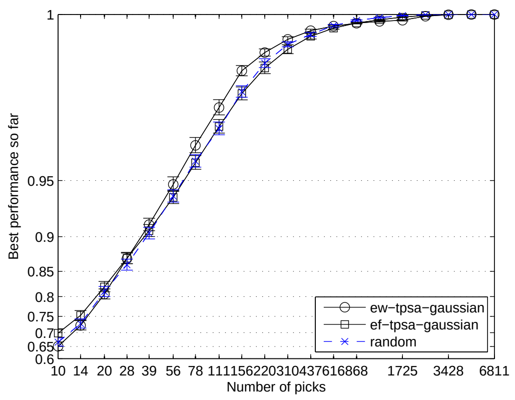
The hyperparameters of the covariance function in Gaussian process regression play an important role in fitting. If the hyperparameters are not chosen properly, the model would overfit or less fit the data.

Though we search for the proper hyperparameters with multiple initials, the likelihood function still affects since it's the measurement of the hyperparameters. Especially in smaller datasets, the marginal likelihood measures worse than the leave-one-out likelihood. A typical situation of fitting is shown in Figure 11.

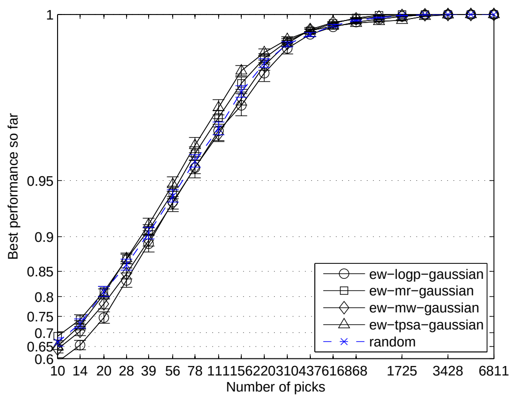
The log hyperparameters shown in Figure 11 are searched parameters that maximize each likelihood function. It is obvious that the marginal likelihood one less fits the data points. The corresponding values of likelihood is shown in Table 2. We see that the marginal likelihood function will not choose the "better" parameter since it thinks that's worse.

The time cost for searching parameters and calculating the integrals[2] is really huge. We only

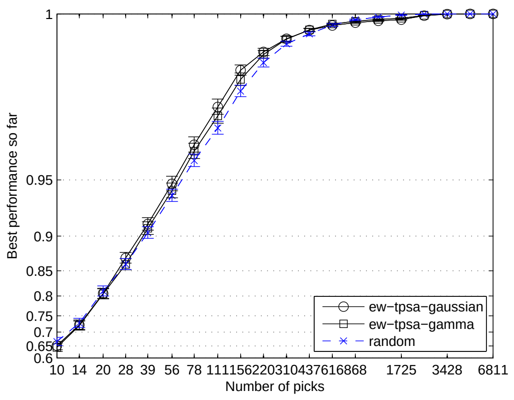
[2]When more than one test data have the same integral, we pick the next data just from the beginning to the end



(a) Equal-width/frequency



(b) Four features

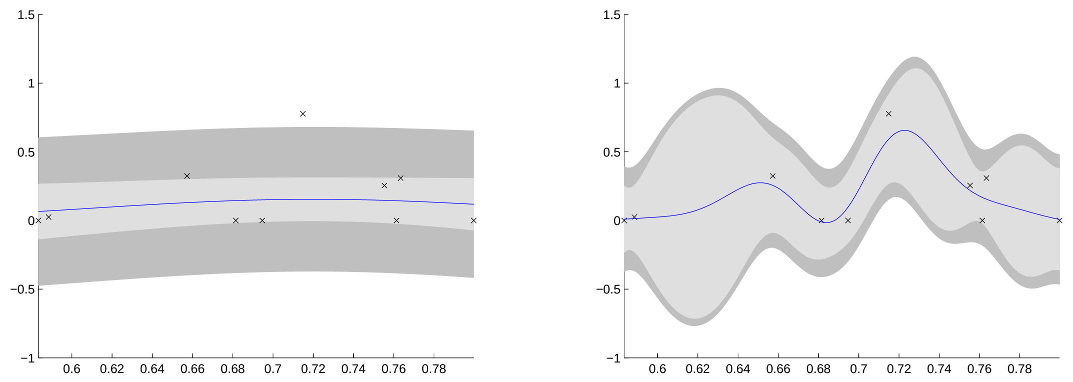


(c) Gaussian/Gamma

Figure 10: Comparison of expected improvement estimate algorithms.

Table 2: Likelihood with different function and parameters

Parameter	Likelihood value with function:	
	Marginal likelihood	Leave-one-out Likelihood
-2.15,-1.95,-1.39	-1.2463	-0.87068
-3.87,-0.86,-1.92	-2.0934	-0.73581



(a) Marginal likelihood $(-2.15, -1.95, -1.39)$ (b) Leave-one-out likelihood $(-3.87, -0.86, -1.92)$

Figure 11: An example of fitting with different likelihood functions.

Table 3: Testing results of Tanimoto kernel SVM

Fold number																			
1		2		3		4		5		6		7		8		9		10	
3	1	1	0	2	0	0	1	0	1	3	0	2	1	1	0	10	2	3	0
86	591	77	603	77	602	59	621	64	616	54	624	85	593	89	591	39	630	26	653

calculate for 100 runs, and only for the first 111 picks. Since this method can easily deal with multi-features, we also test the four features together. The comparison is shown in Figure 12.

This result also shows a good performance from about 50 picks.

5.2.2 Classification problem

As described in previous section, we first divide the dataset into: top 10% data points as positive and the others as negative. With the 10-fold cross validation, the distributions of positive and negative samples in the folds are shown in Figure 13. The number on each bar (fold) is the count of positive points.

When testing with the linear kernel SVM, all the predictions are negative. Even when testing with the Tanimoto kernel, the positive predictions are still rare. The results are listed in Table 3. The four numbers in each block are arranged by:

True Positive	False Positive
False Negative	True Negative

The average accuracy is 90.28%, precision 72.5% and recall 4.66%.

10-fold cross validation with the classification tree still has a low performance. The average accuracy is 84.23%, precision 24.83% and recall 27.07%.

in order due to some implementation strategy. It would be better if they are chosen randomly. However, the latter algorithms use this random strategy.

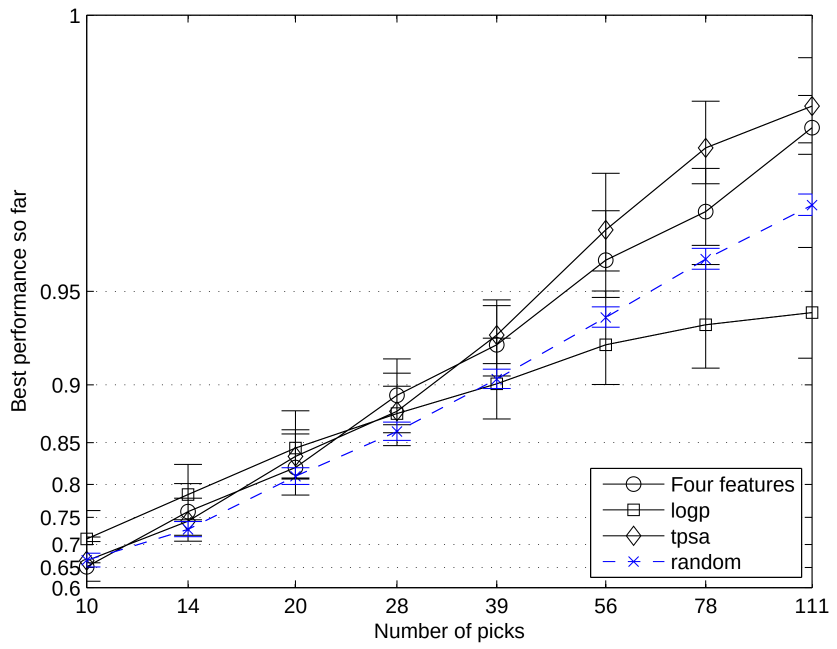


Figure 12: Gaussian process regression result.

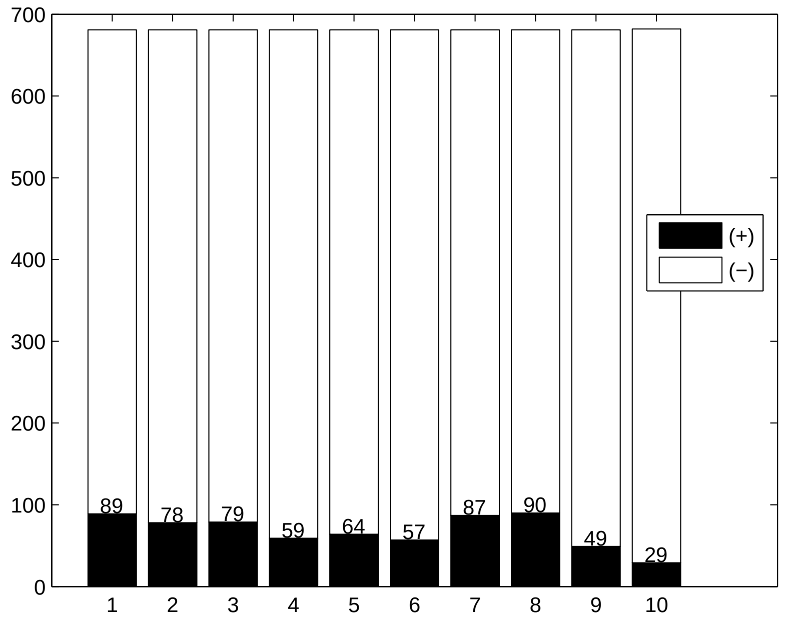


Figure 13: Distribution of positive samples in folds.

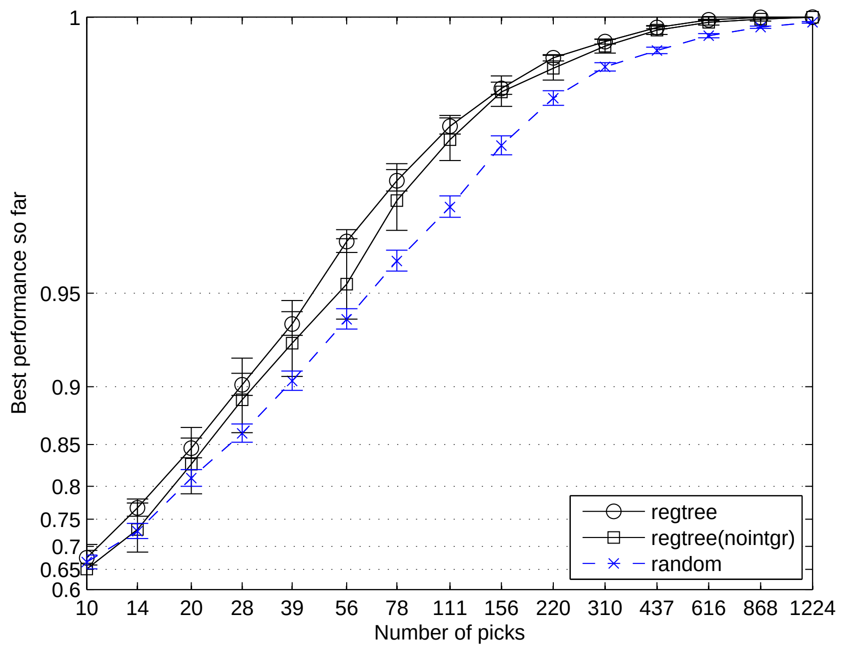


Figure 14: Regression tree and bit-based integrals algorithm result.

Therefore, these classification methods can't meet our requirements.

5.2.3 Regression trees

This algorithm uses the regreesion tree with bagging to estimate the mean and variance of the test data points. The expected improvement integral is used for determining which data to pick. We also compare this method with another similar algorithm – use only the mean of bagging results to choose the next data point (i.e. without the integral). The result is shown in Figure 14. The simple algorithm has only 100 runs (the original has 1,000 runs), but it still shows the integral improves the algorithm especially between 50 to 150 picks.

It shows a big improvement even before just 100 picks. For example, when we pick only 28 data points, we can reach the level 0.9 (performance 90%) while random needs $39 - 28 = 11$ more picks.

5.2.4 Bit-based integrals

The line "bitintgr" in Figure 15 shows the result of bit-based integrals algorithm. It also has 1,000 runs and 1,224 picks per run. In this figure, we also compare it with the regression tree algorithm to show the better performance around 156 picks.

This algorithm works worse than random at the beginning, but raises at a high slope at around 50 picks and soon it is much better than random, and is even better than regression tree.

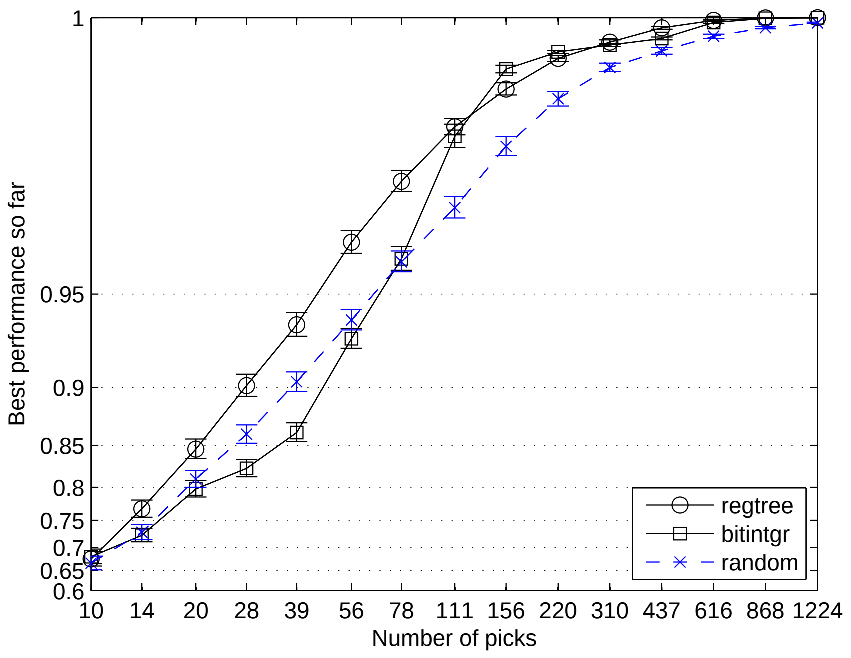


Figure 15: Regression tree and bit-based integrals algorithm result.

6 Evaluation

6.1 Overall comparison

In this subsection, we provide an overall comparison of the algorithms mentioned in this paper, including UCB-tuned, expected improvement estimation, Gaussian process regression, regression tree and bit-based integrals. Figure 16 shows the comparison. In order to show them clearly, we removed the confidence interval bars which are already shown in previous figures.

The figure shows that the regression tree algorithm has a good overall performance, from less than 50 picks to 1,000 picks. The reason may be the structure of trees that can fully exploit the information in each bit of the fingerprints.

The Gaussian process regression method also performs well. However, the huge computational cost reduces its availability on practical problems.

The bit-based integrals algorithm is the best between 100 and 200 picks. It's much worse at the beginning, perhaps it explores too much that it wastes some useful existing information.

Though the expected improvement estimation algorithm and the UCB algorithms are not as good as the above three algorithms, they also show some improvements towards random selection. The fast computation of UCB is a highlight among these algorithms.

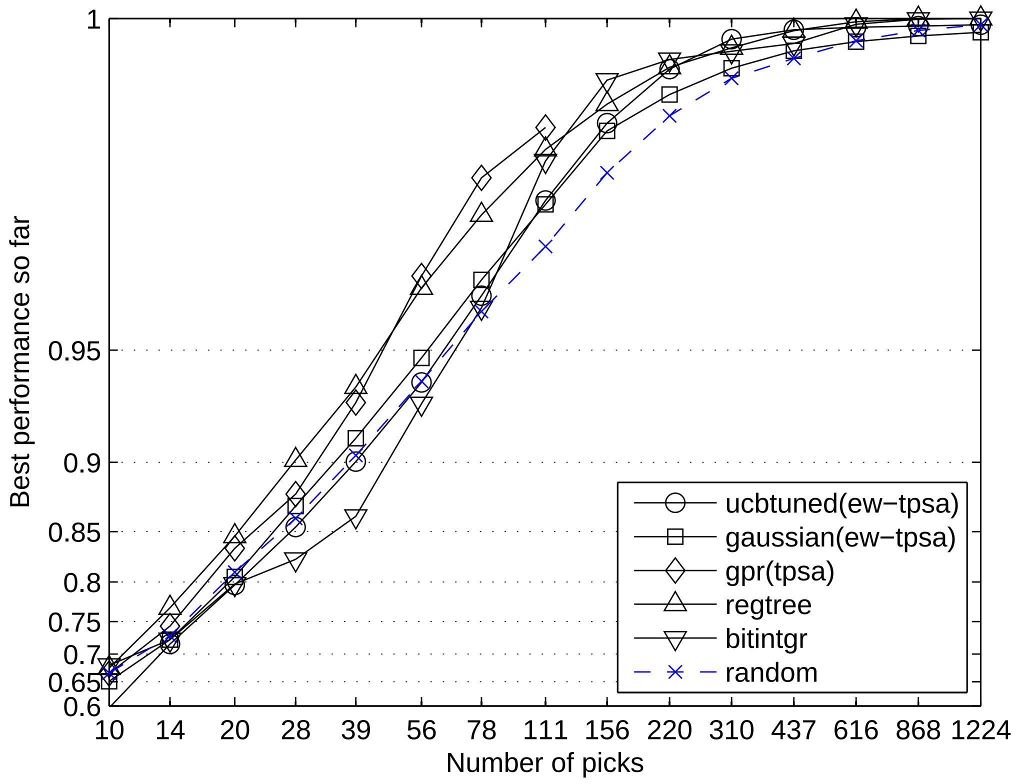


Figure 16: Algorithms comparison.

Table 4: #Picks of first finding the best target

Algorithm	Avg. picks	95% CI bound
Random	805.4	$\pm$ 44.3
Random (Theoretically)	851.5	
UCB-tuned	1340.7	$\pm$ 53.4
Expected improvement(Gaussian)	1403.4	$\pm$ 66.6
Gaussian process regression[a]	N/A	N/A
Regression trees[b]	308.2	$\pm$ 12.4
Regression trees(without integral)[c]	360.9	$\pm$ 50.0
Bit-based integrals[d]	353.5	$\pm$ 14.1

[a] Only 8 out of 100 runs discovered the best target within 111 picks, thus the result isn't convincing.
[b] 998 out of 1,000 runs discovered the best target within 1,224 picks.
[c] All 100 runs discovered the best target within 1,224 picks.
[d] All 1,000 runs discovered the best target within 1,224 picks.

6.2 Time cost for finding the best target

We also evaluate the algorithms by the number of picks they first discover a best target. The smaller the number is, the better the algorithm works. Table 4 gives the results. Only the result of the best combination is listed. In some algorithms, since we stopped them earlier (at 111 picks or 1,224 picks, instead of 6,811 picks), we only count the ones that discover the best target before that limit.

Under this consideration, the regression trees algorithm and the bit-based integrals algorithm discover the best target the fastest. It costs only 36% of random picks to reach the goal.

The UCB algorithms and the expected improvement algorithm rely on the partition of arms, thus result in a bad performance. If the dataset are divided more wisely (maybe consider the fingerprints), they may work better.

7 Conclusion and Future Work

In this work we describe the active learning problem for fast drug discovery. We present algorithms that reduce the number of actual experiments required to discover high performance compounds. Our algorithms are efficient, that they only take about half of the experiments needed by random algorithm. The results are evaluated by a high confidence and present a significant improvement. Although some of the algorithms cost much in computation, they still save the time and cost from actual chemical experiments.

Some of the experiments, esp. the Gaussian process regression algorithm, still need more runs to shorten the confidence interval. More picks in each run are also needed, since our preliminary results suggest that it performs better than other algorithms. However, the computational or implementation method must be improved in the next step so that the algorithm can be applied in practical problems.

The bit-based integrals algorithm performs also strangely at the beginning, that it is much worse than random algorithms especially from 20 to 50 picks. Some deeper analyses should be presented.

For the current methods, some theoretical analyses are encouraged for discovering the reason that they perform well. For new methods, we think that the balance between exploration and exploitation should be considered again, so that the algorithm would neither explore too much (like the bit-based integrals algorithm) nor exploit too much (like the regression trees algorithm).

References

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256, 2002.

[2] Open Babel. http://openbabel.org/.

[3] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[4] Tanimoto Coefficient. http://www.qsarworld.com/files/tanimoto_coefficient-1.pdf.

[5] Tutorial: Fingerprints. http://openbabel.org/wiki/tutorial:fingerprints.

[6] Supervised learning. http://en.wikipedia.org/wiki/supervised_learning.

[7] C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[8] M.K. Warmuth, J. Liao, Gunnar Rätsch, M. Mathieson, S. Putta, and C. Lemmen. Active learning with support vector machines in the drug discovery process. *Journal of Chemical Information and Computer Sciences*, 43(2):667–673, 2003.

MACHINE LEARNING
DEPARTMENT

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Carnegie Mellon.