Why is My Question Closed?
Predicting and Visualizing Question Status on Stack Overflow

Yixing Lao
A53077415
y1lao@cs.ucsd.edu

Chenwei Xie
A53091839
chx037@eng.ucsd.edu

Yue Wang
A53102167
yuw331@eng.ucsd.edu

Abstract

A Stack Overflow question can be closed for 4 reasons, namely "not a real question", "not constructive", "off topic" or "too localized". In this report, we are going to tackle the task of predicting whether a Stack Overflow question will be closed, and if yes, why it will be closed. We first introduce the problem settings and the dataset, then we use various visualization techniques to gain useful insights on the dataset, and finally we show how we use Gradient Boosted Tree to solve this classification problem effectively and achiving 26th / 159 performance in the public leader board on the Kaggle competition.

1. Introduction

Stack Overflow is a privately held website, it was created to be a more open alternative to earlier Q&A sites such as Experts-Exchange, it features questions and answers on a wide range of topics in computer programming. For this project, we use Gradient Boosted Tree to predict whether a question in Stack Overflow will be closed and why it is closed, we evaluate our result in a Kaggle competition [1]. In the following sections, we first perform some interesting visualization on the dataset, and then introduce the prediction algorithm we use, namely Gradient Boosted Tree. Finally, we present the results in the Kaggle competition and give discussions.

1.1. The dataset

The training data for this Kaggle Stack Overflow closeness prediction contest ranged from July 31, 2008 to July 31, 2012. And the public leader board test data ranged from August 1, 2012 to August 14, 2012 while the private leader board test data ranged from October 10, 2012 to October 23, 2012. The arrangement of the data is very reasonable because we use the previous information to predict whether a question will be closed in the future.

The dataset classes are extremly biased. Most (97.9%) questions are open while only 2.1% questions are closed. The closed reasons were: 0.91% not a real question, 0.46% not constructive, 0.52% off topic and 0.18% too localized.

The data includes the following tags: PostCreationDate, OwnerUserId, OwnerCreationDate, ReputationAtPostCreation, OwnerUndeletedAnswerCountAtPostTime, Title, BodyMarkdown, Tags, PostId, PostClosedDate and OpenStatus.

An example of the dataset is following:

- `OwnerUserId`: 136
- `OwnerCreationDate`: 08/02/2008 10:21:53
- `ReputationAtPostCreation`: 475
- `OwnerUndeletedAnswerCountAtPostTime`: 17
- `PostId`: 1496
- `PostCreationDate`: 08/04/2008 18:51:38
- `PostClosedDate`: 08/03/2012 16:38:59
- `OpenStatus`: not constructive
- `Title`: How do I fill a DataSet or a DataTable from a LINQ query resultset?
- `Tag1`: linq
- `Tag2`: web-service
- `Tag3`: c#
- `Tag4`: query
- `Tag5`: vim
- `BodyMarkdown`:
  How do you expose a LINQ query as an ASMX web service? Usually, from the business tier, I can return a typed DataSet or DataTable which can be serialized for transport over ASMX.

```
public static MyDataTable CallMySproc()
{
  string conn = ...;
  MyDatabaseDataContext db = new
      MyDatabaseDataContext(conn);
  MyDataTable dt = new MyDataTable();
  // execute a sproc via LINQ
  var query = from dr in db.MySproc().
      AsEnumerable select dr;
  // copy LINQ query resultset into a
```

```
        DataTable -this does not work !
    dt = query.CopyToDataTable();
    return dt;
}
```

How can I get the resultset of a LINQ query into a DataSet or DataTable? Alternatively, is the LINQ query serializeable so that I can expose it as an ASMX web service?

2. Visualizing the dataset

For normal feature engineering, we could only try different features of the dataset and see if a feature has potential correlation with our prediction. However, we think that visualization is a more powerful tool than trying different features, we would like to use human perception to find the potential features and regard visualization as a heuristic to determine useful features. On the other hand, we could use visualization to find the structure of the dataset. For this dataset, we use various visualization techniques such as word cloud, heat map, parallel coordinates and bar chart to show the structure of dataset as well as useful features. We show the most important and interesting graphs here and explain the structure we find behind these graphs.

2.1. Close ratio and date

We think that there is some connection between close ratio and date, to be more specific, a day, a month or a year. Here, we only show the graph between close ratio and a specific day in a moth, which is between 1 and 31. To do this, we calculate the close ratio for each single day. For example, in the first day of a month, we have n problems and m of them are closed, then we compute the close ratio m / n, to do this, we could get the close ratio of all 31 different days. From Figure 1, we can see that close ratio of varoius day is different. For example, in the fifth day of a month, close ratio is pretty high, while in the twentieth day of a month, the close ratio is pretty low, the same pattern appears in the graph of month and year, which means there is indeed some connection between close ratio and date.

2.2. Open date calendar

We can find lots of interesting information from this calendar. First, its obvious that Stack Overflow is getting more popular these years. More people ask questions at Stack Overflow, which aligns with our common sense. Six or seven years ago, when we had a coding question, we might search it at Google but now, the most possible way is to ask question or search for related questions at Stack Overflow. Another reason is with the computer science burning hot, more people get to learn programming, which makes Stack Overflow popular. Another interesting thing is people ask more question on weekdays than on weekends. Nowadays, we know most programmers balance life and work well and usually they dont work on weekends. So there is few questions on weekends. Especially, people asked more questions in the middle of December, 2011. And when we look at that time, many things changed a lot. For example, C programming language new standard came up and Node.js took off. Because of new things coming up, people had more questions about the new techniques. And another reason is people would like to getting things done as early as possible and then would take a great Christmas Day.

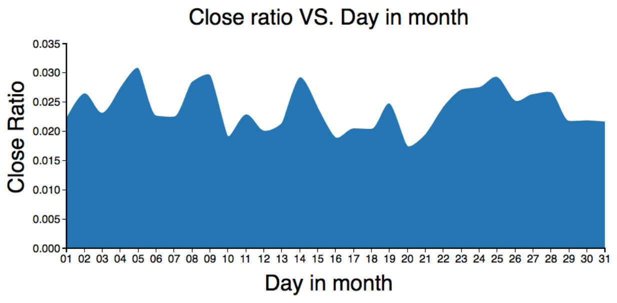

Figure 1. Close ratio's variation in a month.

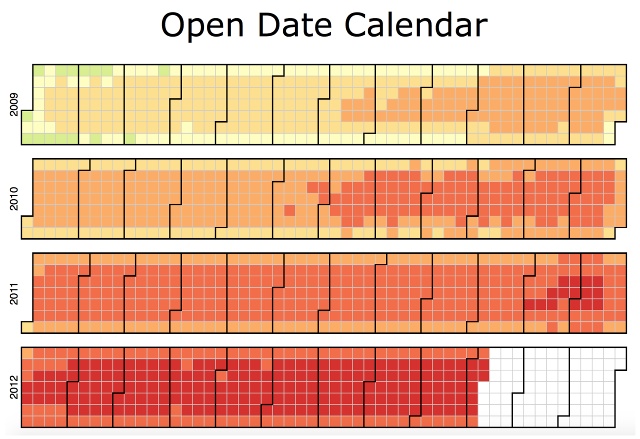

Figure 2. How many questions are created on a day.

2.3. Tag cloud

We can see from Figure 3 the most popular tag is Javascript and PHP. If a programming language is very popular or a programming language is hard to learn and use, people will ask more question about it. We know Javascript and PHP are these programming languages. With different JS techniques coming up, Javascript is sweeping the silicon valley but the grammar of it is strange to the new learners. And people always argue that if PHP is the best language. The most important thing from the tag cloud is the variance of the tag frequency is huge, which makes the tag frequency a good feature.

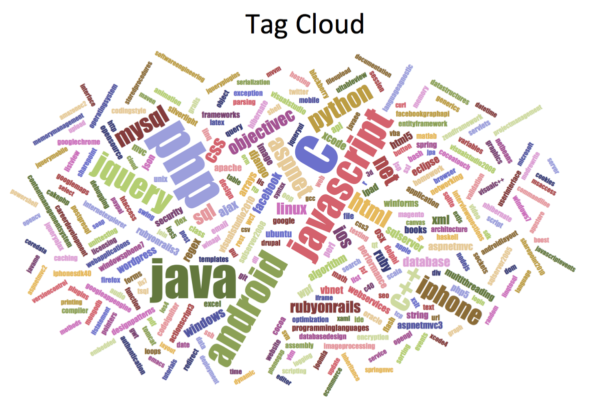
Figure 3. Tags of questions that people ask, popular tag is bigger.

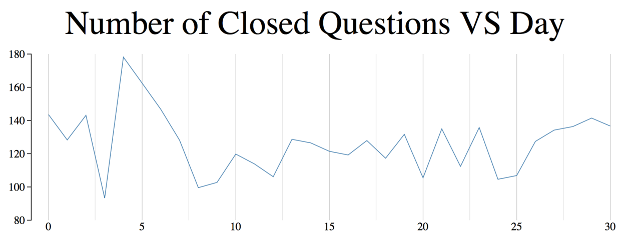

Figure 4. The number of closed questions per day.

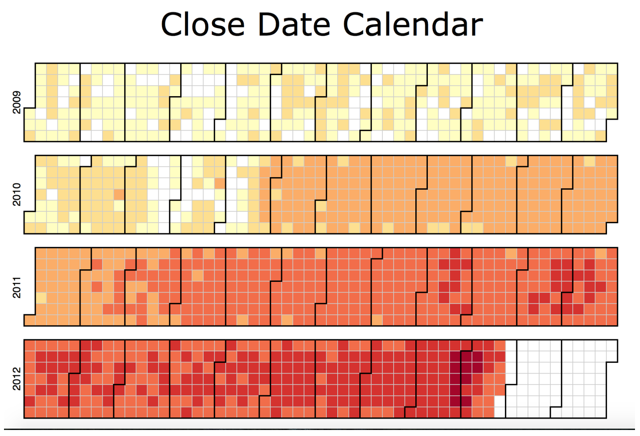

Figure 5. How many questions are closed on a day.

2.4. Number of close questions versus day

We can see from Figure 4, the variance of the number of closed questions versus day is also big, which means probably it is a good feature. And in the beginning of each month, there are more closed questions. People or Stack Overflow may check if a question is meaningless in the beginning of each month. And if it is, the question will be closed.

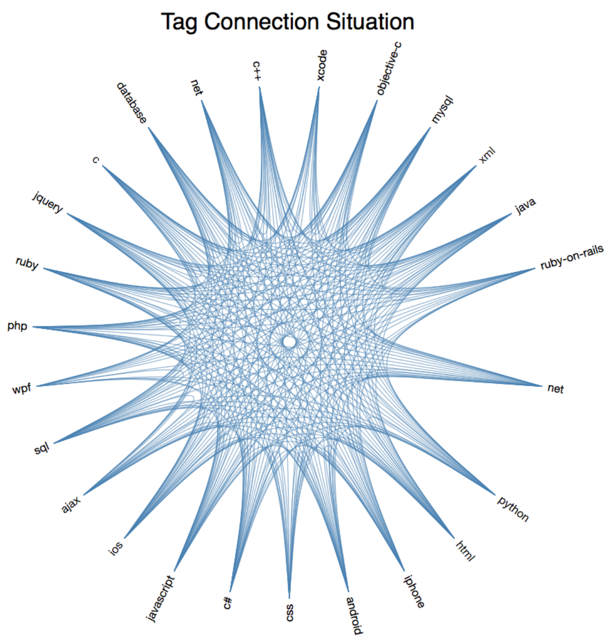

Figure 6. Connection graph of tags that appear together in a question.

2.5. Close date calendar

Figure 5 shows almost the same tendency with the open date calendar. First, we see Stack Overflow is getting more popular. And more questions are closed on weekdays, indicating people do few work on weekends. However, the tendency is not as clear as the open date one. An explanation is people ask more questions on weekdays because they work on weekdays. But they may spend some spare time(not working time) in answering or closing the questions. When people are busy working, it is not likely that they wasting time on Stack Overflow. And the same thing with the open date one, there were more closed questions in the December, 2011. When there are more questions coming up, there are more questions probably being closed. From the tendency, we will conclude that the date should be a good and important feature for predicting closeness of a question. And when doing prediction, we will try to extract more useful information from the date.

2.6. Tag connection

A problem usually belongs to more than one tags, so we want to find if some of the tags appear together. Since there are too many tags, we use the top 25 frequent tags to see the connection. To do this, we connect two tags if they appear in a same problem and get the figure. From this figure, we can see actually two tags appear together because many fields depend on each o. However, we could still find some patterns. For example, the net tag appears less together with

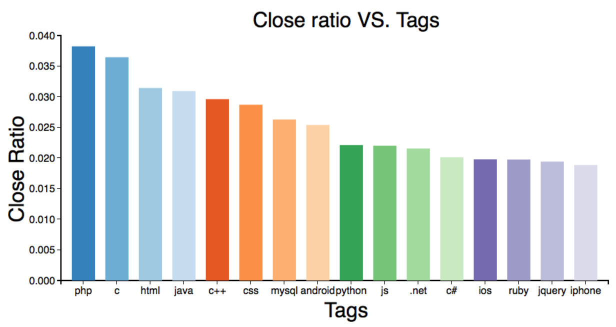

Figure 7. Closing ratio vs. Tags

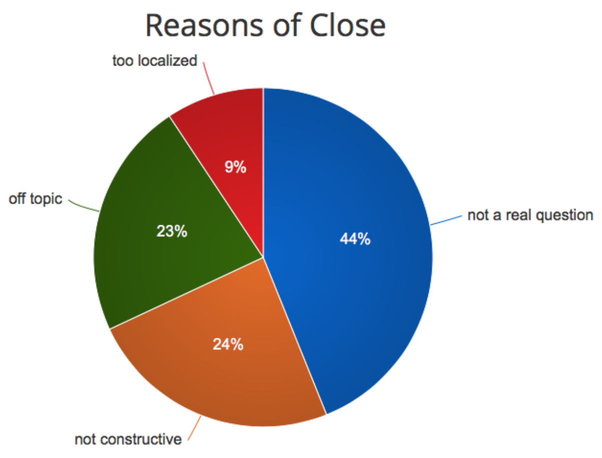

Figure 8. Percentage of the four reasons that a question is closed

other tags, which means ”net” problem is relatively independent in computer science area.

2.7. Close ratio and tag

A problem may belong to at most five tags, however, a problem may be closed because it belongs to a specific tag, so we want to find if the correlation is meaningful. We get the top 16 popular tags and compute the close ratio of those tags. From Figure 7, we can see that popular tags also have high close ratio, especially for PHP and C programming language. We know that they have a verbose and hard grammar, which makes their close ratio high. So a problem is more likely to be closed if it belongs to PHP or C tag.

2.8. Reasons of close

A problem will be closed because of four possible reasons, which are not a real question, too localized, off topic and not constructive. And we can see the proportion of these four reasons, we want to see if there is a connection between closed and the reason of close. To get the proportion, we count the number of problems closed corresponding to a specific reason, and then compute the percentage of that reason. From Figure 8, we could see some patterns. For example, most of problems are closed because they are not a real question, which is 44%, while only a few questions are closed because they are too localized, which is 9%.

3. Predicting question status

A question belongs to one of the 5 different categories, namely “open”, “not a real question”, “not constructive”, “off topic” and “too localized”, where the latter 4 indicate the question is closed. The prediction task is to classify a newly asked question into one of the 5 categories. “Newly asked” here means that the training set only contains questions asked prior to this question in terms of question posting time.

One of the challenges of dealing with this dataset is that the categories are heavily biased. As mentioned in the introduction around 97% of samples are in the “open” category, while the remaining 4 categories share only about 3% of the training data. One could easily achieve 97% categorical accuracy by simply predict “open” all the time. In our experiments, We find that the gradient boosted tree classifier is able to combat such extreme class imbalance due to the nature of boosting procedure. In the following parts of this section, we first describe the motivation of using a gradient boosting approach, then briefly describe the gradient boosted tree algorithm, and finally we present the results and discussions.

3.1. Gradient boosted classification tree

Boosting frameworks such as Adaboost [4] have the natural power to tackle the class imbalance problem [7]. This is due to the fact that it assigns mis-classification costs to data from each class, forcing the classifier to over-sample the data from minority classes or under-sample data from majority classes, since minority classes are more likely to receive wrong predictions at the earlier stages and hence be assigned higher weights.

Gradient boosting [5] take a step further from AdaBoost and combines the idea from gradient descent. Instead of fitting a new weak learner at each boosting stage to the re-sampled data based on distribution weights, gradient boosting fits weak learners to fit the residuals of prediction, or more generally the gradient of the lost function, directly. This idea allows gradient boosting to fit arbitrary differential lost functions directly without using proxies of lost functions of other forms. In our case, multi-class log loss is used. In the following subsections, we follow [2] from Tianqi Chen to review the gradient boosted classification tree algorithm.

3.1.1 Regression trees

First, we introduce the algorithm for training gradient boosted *regression* trees. Assume we have N training samples, K trees. For training sample x_i, the final prediction $\hat{y}_i$ is

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

where F is the space of functions containing all regression trees and $f_k(x_i)$ is the prediction given by the kth regression tree. Notes that different from AdaBoost, we simply sum up the predictions given by individual trees (instead of weighted average in AdaBoost), since at each iteration step, a new tree is fitted to the residual of the previous accumulative prediction (instead of fitted to re-weighted training data in AdaBoost).

The total loss function $\text{Obj}(\Theta)$ is given by the sum of training loss $L(\Theta)$ and model complexity penalty $\Omega(\Theta)$:

$$\text{Obj}(\Theta) = L(\Theta) + \Omega(\Theta),$$

where $L(\Theta)$ is given by

$$L(\Theta) = \sum_{i=1}^{N} l(y_i, \hat{y}_i),$$

and $\Omega(\Theta)$ is given by

$$\Omega(\Theta) = \sum_{k=1}^{K} \Omega(f_k).$$

3.1.2 Global view of training procedure

We start by initializing a model of constant prediction

$$\hat{y}_i^{(0)} = f_0(x_i) = 0.$$

The training is done additively, that is, we add a model f_t at iteration t, to fit the "residual" produced by the accumulative model from time 0 to time $t-1$. We will give more details about the "residual" in the coming section. So, in a global view, the training steps are as follows:

$$\begin{aligned}
\hat{y}_i^{(0)} &= f_0(x_i) = 0 \\
\hat{y}_i^{(1)} &= f_0(x_i) + f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_0(x_i) + f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
&\cdots \\
\hat{y}_i^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)
\end{aligned}$$

The model complexity grows linearly with the number of iterations. Now the remaining issue for train regression trees is how we find the optimal split for individual trees.

3.1.3 Finding the optimal split for individual trees

The goal is of this section is to find the pseudo-optimal tree f_t for boosting iteration t. At iteration t, we fit f_t to the "residual" produced by the accumulative model. In general, the "residual" in a gradient boosted tree is given by the gradient or in general the residual of any order of Taylor expansion.

At iteration t, $\hat{y}_i^{(t)}$ is approximated as

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i),$$

with the loss

$$l(y_i, \hat{y}_i^{(t)}) = l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)).$$

Recall the second order Taylor expansion:

$$f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2} f''(x)\Delta x^2,$$

with the corresponding transformation

$$\begin{cases}
f(x+\Delta x) & \leftarrow l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) \\
x & \leftarrow \hat{y}_i^{(t-1)} \\
\Delta x & \leftarrow f_t(x_i) \\
f(x) & \leftarrow l(y_i, \hat{y}_i^{(t-1)}) \\
f'(x) & \leftarrow g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}) \\
f''(x) & \leftarrow h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}),
\end{cases}$$

$l(y_i, \hat{y}_i^{(t)})$ can be written as

$$l(y_i, \hat{y}_i^{(t)}) \simeq l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i),$$

and hence the objective function can be written as

$$\begin{aligned}
\text{Obj}^{(t)} &= \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t-1)}) + \Omega(f_t) \\
&\simeq \sum_{i=1}^{N} \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i) \right] + \Omega(f_t).
\end{aligned}$$

With the constant terms removed, our new simplified objective to minimize for the purpose of calculating f_t is

$$\text{Obj}^{(t)}_{\text{new}} = \sum_{i=1}^{N} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i) \right] + \Omega(f_t).$$

Following Tianqi's notation, we further represent tree f_t as a collection of leaves:

$$f_t(x) = w_{q(x)}, w \in \mathbb{R}^T, q : \mathbb{R}^d \to \{1, 2, \dots, T\},$$

where w represents the leaf weights, and q the structure of the tree. The complexity of the tree can be defined as

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

where we penalize the number of leaves (γT) and the L2 norm of leaf weights $(\frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2)$.

Then we regroup the objective by each leaf as the sum of T independent quadratic functions

$$\begin{aligned}
\text{Obj}_{\text{new}}^{(t)} &= \sum_{i=1}^{N} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i) \right] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \\
&= \sum_{j=1}^{T} \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda) w_j^2 \right] + \gamma T \\
&= \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2 \right] + \gamma T
\end{aligned}$$

where the $I_j = \{i | q(x_i) = j\}$ is the instance set of leaf j. By setting the derivative in $(\sum_{i \in I_j} g_i) w_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda) w_j^2$ w.r.t w_j to zero, we get the optimal solution

$$w_j^* = -\frac{G_j}{H_j + \lambda},$$

and the objective

$$\text{Obj}_{\text{new}}^{(t)*} = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T.$$

In practice, it is not feasible to compute w_j^* directly, instead we use a greedy approach as building the ordinary decision trees to find the next split: 1) for each feature, sort the instances by value; 2) use linear scan to find the best threshold at that feature and finally 3) return the best split found among all features. Here the Gain of each split is given the sum of scores of left and right sub trees subtracted by the score of the unsplitted tree and penalty γ by

$$\text{Gain} = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma.$$

3.1.4 Regression tree to classification tree

For classification tree of M possible classes, M regression trees are fitted to each class. For a training sample x_i, the ground truth class probability is given by

$$y_{i,j} = \begin{cases} 1 & \text{if } x_i \text{ is of class } j \\ 0 & \text{otherwise} \end{cases},$$

and the predicted class probability is $p_{i,j}, j \in [1, M]$. The loss is calculated as the multi-class logarithmic loss

$$\text{logloss} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} \log(p_{i,j}),$$

where N is the number of observations. The multi-class logarithmic loss is a generalization of the binary class log loss (or cross-entropy loss)

$$\text{logloss}_{\text{binary}} = -\frac{1}{N} \sum_{i=1}^{N} (y_i \log(p_i) + (1 - y_i) \log(1 - p_i)).$$

Similar to regression tree, class j's tree is built as follows:

$$\begin{aligned}
\hat{y}_{i,j}^{(0)} &= f_j^{(0)}(x_i) = 0 \\
\hat{y}_{i,j}^{(1)} &= f_j^{(0)}(x_i) + f_j^{(0)}(x_i) = \hat{y}_{i,j}^{(0)} + f_j^{(1)}(x_i) \\
\hat{y}_{i,j}^{(2)} &= f_j^{(0)}(x_i) + f_j^{(1)}(x_i) + f_j^{(2)}(x_i) = \hat{y}_{i,j}^{(1)} + f_j^{(2)}(x_i) \\
&\dots \\
\hat{y}_{i,j}^{(t)} &= \sum_{k=1}^{t} f_j^{(k)}(x_i) = \hat{y}_{i,j}^{(t-1)} + f_j^{(t)}(x_i),
\end{aligned}$$

where $f_j^{(t)}(x_i)$ is fitted to the residual of class j at iteration t.

3.2. Feature engineering

We use feature of length 38 to represent each sample. The feature consists of three different categories: text feature, code feature and feature derived from various tags such as time.

All feature used are listed as follows:

0. `len-code`: length of all code segments
1. `len-first_code`: length of first code segment
2. `len-first_text`: length of the first block of text
3. `len-last_code`: length of last code segment
4. `len-last_text`: length of the last text segment
5. `len-text`: length of characters in all text segment
6. `len-title`: length of title
7. `mean-code`: mean length of code segments
8. `mean-sentence`: mean length of sentence
9. `mean-text`: mean length of text segments
10. `num-code_block`: mean length of code segments
11. `num-digit`: number of digits in text
12. `num-exclam`: number of exclams in text
13. `num-final_thanks`: number of thanks received
14. `num-i_start`: number of sentence starts with "I"
15. `num-init_cap`: number of sentence starts with capital letter
16. `num-lines`: number of lines in text
17. `num-non_word`: number or non askii characters in text

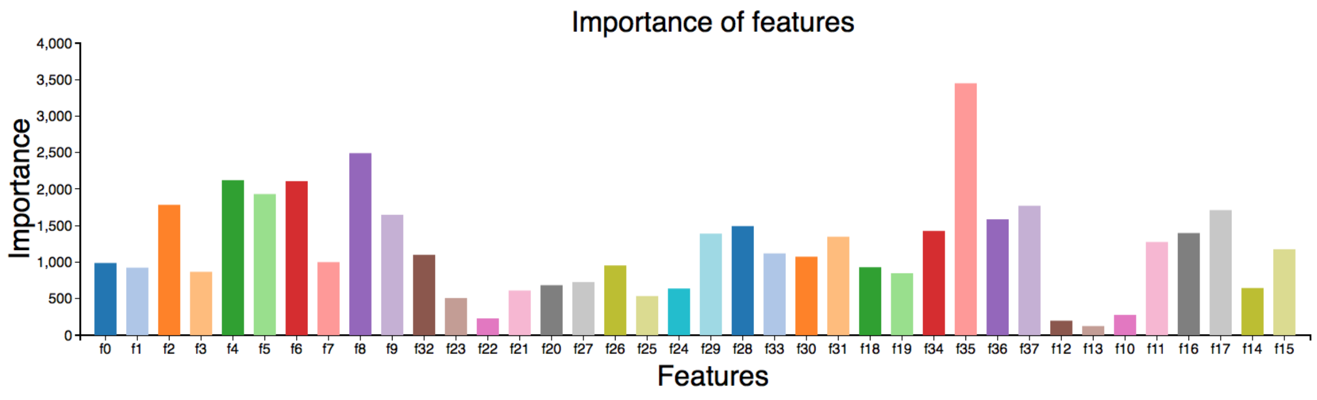

Figure 9. A figure shows the importance of different features. The most important five features is `f35:user-age`

18. `num-period`: number of period mark in text
19. `num-question`: number of question mark in text
20. `num-sentence`: number of sentence in text
21. `num-tags`: number of tags received
22. `num-text_block`: number of text segments
23. `num-url`: number of urls used
24. `ratio-exclam_sentence`: number of exclaims divided by the number of sentences
25. `ratio-first_code_code`: length of first code segments divided by total length of all code segment
26. `ratio-first_text_first_code`: length of first test segment divided by length of the first code segment
27. `ratio-first_text_text`: length of first text segment divided by the length of all text segment
28. `ratio-period_sentence`: number of period marks divided by the number of sentence
29. `ratio-question_sentence`: number of question marks divided by the number of sentence
30. `ratio-text_code`: length of all text segments divide by the length of all code segments
31. `time-day`: integer 1 to 31
32. `time-month`: integer 1 to 12
33. `time-weekday`: integer 0 to 6
34. `time-year`: integer 2008 to 2012
35. `user-age`: integer, user's age in seconds
36. `user-good_posts`: number of good posts the use receive at the posting time of this post
37. `user-reputation`: user's reputation at the posting time of this post

3.3. Experiment and evaluation

We use XGBoost's [3] implementation of gradient boosted classification tree. Compared with Scikit-learn's [6] implementation, XGBoost is distributed – it can utilize multiple cores of CPU and scale across different machines, and thus it is significantly faster than scikit-learn when running on muliti-core machines.

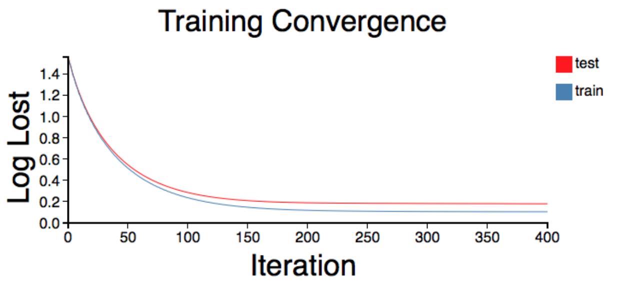

Figure 10. The changing log lost corresponds to times of iteration.

There are various hyper parameters in a gradient boosted tree. After multiple cross validation rounds, we report our results using the following parameter setting:

- `bst:max_depth`: 3
- `bst:eta`: 0.02
- `subsample`: 0.01

The performance is evaluated using multi-class logarithmic loss. For each test observation, a probability for each class is predicted. The loss is calculated as

$$\text{logloss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(p_{i,j}),$$

where N is the number of observations, M is the number of class labels, $y_{i,j}$ is 1 if observation i is in class j and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation i is in class j.

In the original Kaggle competition, the training data contains questions posted from 2008-07-31 to 2012-07-31, and the public leader board test data is from 2012-08-01 to 2012-08-14. So, the model is tested with newly posted questions rather than randomly sampled test set from the whole dataset. Since Kaggle does not release the label of it's public leader board, we first sort our data according to creation date of the question and take the first 90%

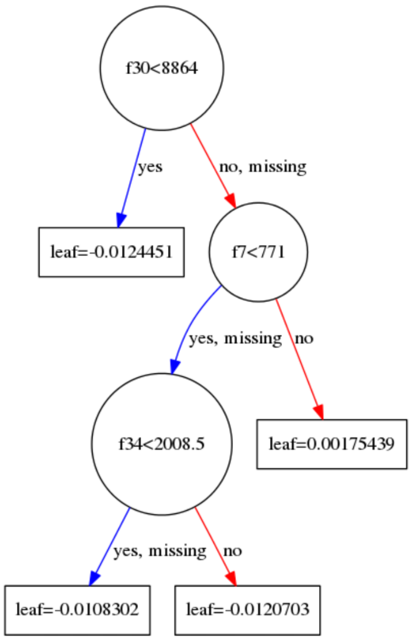

Figure 11. Example of a regression tree: class 0 at iteration 1

data as training set and 10% data as testing set. The training set contains 3,033,475 samples and the test set contains 337,053 samples.

We train our model on a Amazon AWS c4.8xlarge 32 core machines. The maximum iteration is set to 1000 with early stopping if the validation error does not decrease in 10 contiguous rounds. The training time is around 1 hour, we can see the result more clearly in Figure 10.

3.4. Results and discussion

Our best model achieve 0.17786 multi-class log loss in our 90%-10% time sorted split train / test set. Our result will rank 26th / 159 if it were submitted to the Kaggle public leaderboard. Notes that this is only an approximated result, since we can not get the label of the pubilc leaderboard, so the train and test set are different from the results posted on Kaggle. But since the train and test set are spitted by sorted question posting time, the results obtained in our dataset shall be comparable to the results in the public leader board. Please refer to the previous section for the training and evaluation details.

Figure 11 shows an example of a regression tree for class 0 at iteration 1 obtained the boosting procedure.

Here we report the feature importance output by the the gradient boosted tree. The more frequently a feature is selected for a node split, the more import that feature is. There are some interesting observations that we could find from importance of different features:

Rank	Team Name	Score
1	Malacka	0.15760
2	nbu	0.16114
	......	
24	neggert	0.17653
25	vikas	0.17766
	Our Result	***0.17786***
26	Anaconda	0.17806
27	Chaotic Experiments	0.17894
28	FZJ	0.17926
	......	
	Basic Benchmark	***0.21726***
	......	
158	Words n' Stuff	5.86013

Table 1. Our result at Kaggle

Index	Feature	Importance	Relative importance
35	`user-age`	3450	0.07644
8	`mean-sentence`	2493	0.05524
4	`len-last_text`	2123	0.04704
6	`len-title`	2110	0.04675
5	`len-text`	1933	0.04283

Table 2. The most import features

- user-age is the most important feature, since a user's age could reveal a lot of other information, such as user's reputation, and these features have a strong connection whether a question will be closed or not.
- The second to fifth important features are all about length attribute of the problem, length of sentence, length of text and so on. It is obvious that people tend to close those question which are very verbose and unclear, which usually have a very long sentence and title.
- num-exclam and num-text_block are two very unimportant features. It conforms our thinks that these two features are hard to reveal information related whether a question will be closed or not.

4. Conclusion

In this project, we answer the prediction question whether a Stack Overflow question will be closed, and if yes, why it will be closed. We first use various visualization techniques to gain useful insights on the dataset. Then we show how we show that by applying Gradient Boosted Tree and with extensive feature engineering, this prediction problem can be solved effectively. Our result is roughly comparable to the 26th / 159 model in the public leader board on Kaggle.

References

[1] Kaggle predict closed questions on stack overflow. `https://www.kaggle.com/c/predict-closed-questions-on-stack-overflow`.

[2] T. Chen. Introduction to boosted trees. `https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf`.

[3] T. C. *et al.* XGBoost. `https://github.com/dmlc/xgboost`.

[4] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[5] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[6] Scikit-learn developers. Scikit-learn gradient boosting classifier.
`http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html`.

[7] Y. Sun, M. S. Kamel, and Y. Wang. Boosting for learning multiple classes with imbalanced class distribution. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 592–602. IEEE, 2006.