Machine Learning Final Project Report
Dropout Prediction in MOOCs

Chun-Chih Wang, Yi-Lin Cheng, Yih-Chieh Hsu
Department of Computer Science and Information Engineering, National Taiwan University
{r04922066, r04922058, r04944001}@ntu.edu.tw

ABSTRACT

This report describes the solution of *ML or WT?* team in NTU Machine Learning Final Project (Fall, 2015), which aims to predict the dropouts of online MOOC courses.

1. INTRODUCTION

We first elaborate our effort on feature engineering and the models we used. Next, we describe how we used these models combining with other machine learning methods to improve our performance. Finally, we present experimental results and discussions among the models.

2. FEATURE ENGINEERING

We keep some of the features provided by TA, and add our own features. We have spent a lot of time on discovering more useful features. In the end, we find there are six types of our features performing very well in the competition in addition to TA's, and there are thirty dimensions in total of our final features.

2.1 TA Features

Among 17 features provided by TA, we remove three features, *chapter_cnt, sequential_cnt* and *video_cnt.* In this way, they perform better when combined with our features.

2.2 First-Access-Last-Access Difference (day-diff)

This feature captures the active duration among thirty days, which can be written as following,

$$\text{day-diff} = \text{first-access date} - \text{last-access date}$$

Note that when day-diff is larger, the longer the user is active during thirty days. Those users with day-diff equal to one, are considered more likely to drop on the course since they might have a tendency not to log in in the following ten days.

2.3 Valid-Video-Log-Count (valid-vlog-cnt)

We calculate the time differences between those video logs and their next logs. If they are larger than a minute but smaller than two hours, then we consider them as valid video events. This shows whether a user actually watches videos or not.

2.4 Log-Day-Histogram (log-day)

We count the number of days each user logged in MOOC within an interval. We set the interval to seven days to simulate a week. Hence, there are five weeks in total, with only two days in the last week. By doing this, we want to capture the behavior of each user through the histogram.

2.5 Time-Related Features

Since dropout is defined by whether the user logged in or not in the following ten days, we think "time" is an important indicator. Therefore, we came up with four time-related features, *First-Access Difference, Last-Access Difference, Course-Start Difference* and *Course-Access Difference.* They are all calculated in seconds and divided by 86400 to present in unit of day.

2.5.1 *First-Access Difference (first-access-diff)*

$$\text{first-access-diff} = \text{first-access time} - \min(\text{first-access time})$$

min(first-access time) is the smallest first-access time among all users, and the *first-access-diff* of a user is the difference between the smallest first-access time and the user's first access time.

2.5.2 *Last-Access Difference (last-access-diff)*

$$\text{last-access-diff} = \text{last-access time} - \min(\text{last-access time})$$

min(last-access time) is the smallest last-access time among all users, and the *last-access-diff* of a user is the difference between the smallest last-access time and the user's last access time.

2.5.3 *Course-Start Difference (course-start-diff)*

$$\text{course-start-diff} = \text{course-start time} - \min(\text{course-start time})$$

min(course-start time) is the smallest course-start time among all users, and the *course-start-diff* of a course is the difference between the smallest course-start time and the course's course-start time.

2.5.4 *Course-Access Difference (course-access-diff)*

$$\text{course-access-diff} = \text{first-access time} - \min(\text{course-start time})$$

Calculate the difference between the user's first-access time and the corresponding course-start time.

2.6 Object Count in Course

We count the number of different objects (discussion, problem, video) in each course. It is not fair to directly compare the number of logs among each type of objects. Instead, comparing to the total number of different objects in a course can bring us more correct results. Moreover, because there are some missing objects in *object.csv*, we combine all provided files, including *log_train.csv*, *log_test.csv*, *object.csv* and count all unique objects in the same course. Hence, there are three features, *discussion-obj, problem-obj, video-obj*.

2.7 Longest Offline/Online Day

We compute the Longest-Offline-Day (*longest-offline*) and Longest-Online-Day (*longest-online*). Longest-Offline-Day is the largest number of days that a user had not logged in continuously, while Longest-Online-Day is the largest number of days that a user had logged in continuously. These two features somehow have similar meanings with *day-diff* but could capture more details of a user's behavior.

2.8 Experiment

Table 1 and Table 2 show how our features performed in this competition. We add each type of features accumulatively. For example, the third row, "Before 2.3" means we use features in Section 2.1, 2.2, 2.3, that is, TA's, day-diff, valid-vlog-cnt, to verify usefulness of our features. Then, it is followed by CV, public score and private score of each track.

	CV	Public	Private
Before 2.1	0.948280	0.959433	0.960796
Before 2.2	0.949790	0.958882	0.962716
Before 2.3	0.949884	0.959281	0.962648
Before 2.4	0.950039	0.958850	0.962304
Before 2.5	0.958273	0.967123	0.967403
Before 2.6	0.958332	0.967588	0.967588
Before 2.7	0.958332	0.967517	0.967158

Table 1: Evaluation of features in track 1

	CV	Public	Private
Before 2.1	0.865182	0.875095	0.880920
Before 2.2	0.868718	0.875326	0.881102
Before 2.3	0.869008	0.876196	0.882107
Before 2.4	0.869880	0.875985	0.882326
Before 2.5	0.876091	0.884886	0.888642
Before 2.6	0.876900	0.884115	0.889476
Before 2.7	0.876931	0.883062	0.890716

Table 2: Evaluation of features in track 2

3. MODELS

In this section we introduce how we compose our models for dropout prediction and how we evaluate them. All the following experiments are running in Python using Scikit-learn [1] 1.4. We adopt 5-fold cross-validation to evaluate the performance of each model, and use different error measures based on different tracks:

- Track 1: Average Precision
  Track 1 uses AP as a metric on ranked list sorted by the probability estimate that each enrollment would be a dropout. And the score is based on this formula:

  $$AP(g) = \frac{1}{M}\sum_{n=1}^{M}\frac{dropout_n(g)}{n}$$

  ,where $dropout_n(g)$ = number of dropouts in top n enrollments.
  According to the above formula, we could realize that what we need to do is to put the most possible dropout enrollments in front of non-dropout enrollments (especially in top-M estimates) to maximize the AP score.

- Track 2: Weighted Accuracy
  Track 2 uses weighted accuracy as a metric based on the result of binary classification. The basic idea is when one student takes more courses on the platform, we have more data/information about him/her. The score is based on this formula:

  $$WA(g) = \frac{\sum_{n=1}^{N} c_n^{-1}\,[\![g(n^{th}enrollment) = y_n]\!]}{\sum_{n=1}^{N} c_n^{-1}}$$

  ,where c_n = total number of courses the student of the n^{th} enrollment takes.
  It means that wrong predictions on different enrollments receive different penalties, hence we assign sample weights in track 2 by the following approach.

Benefited from Scikit-learn, we can easily adopt cross-validation by its *cross_validation* module. In all of the models mentioned in the following section, we set the same scoring metrics and sample weights based on tracks, where

- in track 1, we specify *scoring* parameter of *cross_val_score* method to *average_precision*, and set *sample_weight* parameter of *fit* to *None*, indicating all samples have the same weight.

- in track 2, we specify *scoring* parameter of *cross_val_score* method to *accuracy*, and set *sample_weight* parameter of *fit* to the reciprocal of number of courses of the user of each enrollment takes.

3.1 Logistic Regression (LR)

Logistic regression is a common approach that widely used for binary classification. Based on "linear first" principle, we first try the logistic regression model. Using *LogisticRegression* class in *linear_model* module, we could call *predict* and *predict_proba* method for track 2 and track 1, respectively, to get the classification result and the probability estimate. *LogisticRegression* implements regularized logistic regression using the Liblinear [2] library, where Liblinear solves a L2-regularized unconstrained optimization problem:

$$\min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^{\intercal}\mathbf{w} + C\sum_{n=1}^{N}\log\left(1+\exp\left(-y_n\mathbf{w}^{\intercal}\mathbf{x}_n\right)\right)$$

,where $C > 0$ is a penalty parameter.

By adjusting the C parameter while remaining all other parameters as default, we get numerical results in Table 3 by cross-validation.

C	track 1	track 2
0.01	**0.940562**	**0.858069**
0.1	0.938043	0.854190
1	0.938168	0.854325
10	0.938227	0.854066
100	0.938578	0.855061

Table 3: Performance of Logistic Regression by varying C in track 1 and track 2

3.2 Support Vector Machine (SVM)

Here, we try to exploit support vector machine (SVM) models to solve the binary classification problem. Given a set of training examples, each marked one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier. Because SVM models only provide the classification results, we only use it in track 2.

3.2.1 Linear SVM

Again, we start from a linear one. Based on *svm* module in Scikit-learn, we use its *LinearSVC* class as our basic model. *LinearSVC* is implemented in terms of Liblinear [2].

Given a set of instance-label pairs $(x_i, y_i),\ i = 1, 2, ..., n$ and weights $\mathbf{w}$, Liblinear solves the following unconstrained optimization problem:

$$\min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^{\intercal}\mathbf{w} + C\sum_{n=1}^{N} \max\left(1 - y_n\mathbf{w}^{\intercal}\mathbf{x_n}, 0\right)^2$$

,where $C > 0$ is a penalty parameter.

By adjusting the C parameter while remaining all other parameters as default, we get numerical results in Table 4 by cross-validation.

C	track 2
0.01	0.864342
0.1	0.864446
1	**0.864550**
10	0.861221
100	0.836313

Table 4: Performance of linear SVM by varying C

3.2.2 Non-linear SVM

We then proceed to use *SVC* class in *svm* module, which is implemented in terms of LibSVM [3]. LibSVM solves the following optimization problem:

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & \frac{1}{2}\mathbf{w}^{\intercal}\mathbf{w} + C\sum_{n=1}^{N}\xi_n \\ \text{subject to} \quad & y_n\left(\mathbf{w}^{\intercal}\phi\left(\mathbf{x_n}\right) + b\right) \geq 1 - \xi_n \\ & \xi_n \geq 0 \end{aligned}$$

Here training examples $\mathbf{x_i}$ are mapped into a higher dimensional space by transformation function ϕ. In our case, we use the RBF kernel, where

$$K\left(\mathbf{x_i}, \mathbf{x_j}\right) = \exp\left(-\gamma \left\|\mathbf{x_i} - \mathbf{x_j}\right\|^2\right),\ \gamma > 0$$

By adjusting the penalty parameter C and γ in RBF kernel while remaining all other parameters as default, we get numerical results in Table 5 by cross-validation.

$C\backslash\gamma$	0.01	0.1	1	10	100
0.01	0.862766	0.857498	0.793320	0.793320	0.793320
0.1	0.865877	0.865390	0.793320	0.793320	0.793320
1	0.868190	**0.868843**	0.849057	0.792946	0.793029
10	0.868293	0.863036	0.840316	0.785014	0.787886
100	0.866489	0.854512	0.831190	0.777018	0.785086

Table 5: Performance of SVM by varying C and γ in track 2

3.3 Ensemble Models

Although the SVM models were popular, they don't provide efficient computation on large-scale data. We try to use the following ensemble models to test whether they're effective methods to solve such a classification problem. The *random_state* parameters of all of the following models are set to 1126 for consistency.

3.3.1 Random Forest (RF)

A random forest model is constituted of a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The other mathematical details are ignored here due to space limitation. Using *RandomForestClassifier* class in *ensemble* module, we could call *predict* and *predict_proba* method for track 2 and track 1, respectively, to get the classification results and the probability estimates. We control the number of estimators, which is the number of trees in the forest, and the maximum depth of the tree to get the following experimental results in Table 6 and Table 7.

#estimators\max_depth	4	8	12
300	0.949010	0.954217	0.957043
600	0.948916	0.954269	0.957142
900	0.948960	0.954254	**0.957145**
1200	0.948970	0.954250	0.957141

Table 6: Performance of Random Forest by varying #estimators and tree depth in track 1

#estimators\max_depth	4	8	12
300	0.866344	0.872960	0.874515
600	0.866572	0.873043	**0.874526**
900	0.866727	0.873074	0.874505
1200	0.866738	0.873146	0.874411

Table 7: Performance of Random Forest by varying #estimators and tree depth in track 2

3.3.2 AdaBoost-Decision Tree (AdaBoost)

An AdaBoost model fits a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases. Using *AdaBoostClassifier* class in *ensemble* module, we could call *predict* and *predict_proba* method for track 2 and track 1, respectively, to get the classification results and the probability estimates. We control the number of estimators and the maximum depth of base estimator, where base estimator is decision tree classifier, to get the following experimental results. Table 8 shows the performance in track 1 and track 2.

#estimators	track 1	track 2
200	0.956061	0.872929
500	**0.956378**	0.87383
1000	0.956151	**0.874049**
2000	0.955650	0.873852

Table 8: Performance of AdaBoost-DT by varying #estimators in track 1 and track 2

3.3.3 Gradient-Boosted Decision Tree (GBDT)

A gradient boosted decision tree model builds the model in a stage-wise fashion, where base learner is decision tree. Using *GradientBoostingClassifier* class in *ensemble* module, we could call *predict* and *predict_proba* method for track 2 and track 1, respectively, to get the classification results and the probability estimates. We control the number of estimators and the maximum depth of decision trees to get the following experimental results in Table 9 and Table 10.

#estimators\max_depth	2	4	6
200	0.955743	0.958041	0.958164
400	0.957107	**0.958207**	0.957289
600	0.957636	0.958069	0.956295
800	0.957892	0.957713	0.955460

Table 9: Performance of GBDT by varying #estimators and tree depth in track 1

#estimators\max_depth	2	4	6
200	0.873924	0.876589	0.876890
400	0.875324	0.876631	0.875168
600	0.875915	0.875967	0.873872
800	0.876174	0.875448	0.872638

Table 10: Performance of GBDT by varying #estimators and tree depth in track 2

4. METHODS

Besides using different models for training, we also tried other methods to achieve better performance. Methods we used can be divided into three main categories, which are *clustering*, *blending* and *predict track 2 by results of track 1.*

In this section, we are going to explain our implementations of these three methods. Features used throughout this section are 30-dimension features presented in Section 2.

4.1 Clustering

Since similar enrollments may have similar behaviors, we grouped them together and trained each group individually by GBDT. Parameters used for 5-fold CV were n_estimators =[300,400], max_depth=[2,3,4], max_features=[“auto”, 0.5][1]. Finally, we combined results from each group together to produce the final output.

4.1.1 Cluster by k-means

There are lots of different clustering algorithms, we chose one of the most popular algorithms - k-means clustering. Enrollments were clustered into 3 groups and 5 groups. Results are shown in Table 11 and Table 12.

Group ID	#Train	#Test	Track 1 CV	Track 2 CV
1	24757	8687	0.951302	0.867213
2	46353	6381	0.965614	0.896468
3	25327	9040	0.945757	0.852568

Table 11: Performance of clustering by k-means (3 groups)

Group ID	#Train	#Test	Track 1 CV	Track 2 CV
1	25327	2714	0.945757	0.852568
2	11953	3411	0.962038	0.890070
3	15700	6961	0.969203	0.908408
4	24754	4641	0.951302	0.867213
5	18700	6381	0.963473	0.889465

Table 12: Performance of clustering by k-means (5 groups)

4.1.2 Cluster by day-diff

Our goal is to predict whether a user will have logs in the 31st-40th days from the start date of the course. Thus, we considered day-diff as an important feature. Enrollments were clustered into three groups: day-diff=[0,9], day-diff=[10,19], day-diff=[20,29]. Results are shown in Table 13.

Day-diff	#Train	#Test	Track 1 CV	Track 2 CV
0-9	25327	2714	0.945757	0.852568
10-19	11953	3411	0.962038	0.890070
20-29	15700	6961	0.969203	0.908408

Table 13: Performance of clustering by day-diff

4.1.3 Cluster by Courses

Since each course has it’s own properties, another way to cluster is dividing enrollments by course ID. There are 39 different courses in the data set. Due to space limitation, we won’t list the CV scores for each course here.

The public and private scores of above three clustering approaches shown in Table 14 were not as good as expected. The results show low accuracy compared to using GBDT to

[1]*max_features* controls the number of features considered when looking for the best split. “auto” means max_features = sqrt(n_features) and 0.5 means max_features = 0.5*(n_features).

train data without clustering. One of the reasons may be that after clustering, number of enrollments in each group was too few. Though the results are not good enough, they can still be used as sources for blending.

Method	track 1		track 2	
	Public	Private	Public	Private
GBDT	0.967417	0.967013	0.884253	0.890386
K-means (3)	0.949614	0.949724	0.850224	0.861440
K-means (5)	0.954104	0.955305	0.848983	0.852844
Day-diff	0.965269	0.966933	0.884276	0.888722
Course	0.953704	0.956637	0.879821	0.885130

Table 14: Performance of clustering

4.2 Blending

Blending techniques are widely used in machine learning competitions. By blending outputs from diverse hypotheses, we may obtain more accurate results.

4.2.1 Uniform Blending

Uniform blending can be used in both track 1 and track 2. For track 1, we blended outputs by

$$G(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} g_t(\mathbf{x})$$

Since different models may output different ranges of probability, we scaled outputs to [0,1] by

$$y_i = \frac{y_i - \min_{n\in N}(y_n)}{\max_{n\in N}(y_n) - \min_{n\in N(y_n)}}$$

before blending. For track 2, we used uniform voting to blend the outputs. By blending five outputs from Section 4.1, we obtained public/private score=0.963274/0.964305 for track 1, and public/private score=0.883242/0.889053 for track 2.

We also blended results from three different models: GBDT, RF and AdaBoost. The results are shown in Table 15 below. Parameters used were

- GBDT: n_estimators=400, max_depth=4, max_features=0.5
- RF: n_estimators=300, max_depth=12
- AdaBoost: n_estimators=500

Method	track 1		track 2	
	Public	Private	Public	Private
GBDT	0.967417	0.967013	0.884253	0.890386
RF	0.965330	0.967821	0.881911	0.886911
AdaBoost	0.965938	0.965091	0.879266	0.885910
Uniform Blending	0.967153	0.967423	0.883327	0.889910

Table 15: Performance of uniform blending

4.2.2 Stacking (Stacked Generalization)

Stacking involves training a learning algorithm to combine the predictions of several other learning algorithms. The original inputs are *training_data*, *training_truth*, *testing_data*. For each single model, our algorithm involves three steps:

1. Partition *training_data* into five folds: *train_in_1*, *train_in_2*, ... , *train_in_5*
2. For i = [1,5]
   (a) Train g^- using four folds other than i-th fold.
   (b) Use g^- to predict i-th fold and output *train_out_i*.
   (c) Use g^- to predict *testing_data* (without partitioned) and output *test_out_i*.
3. Combine *train_out_i* into *train_out*. Take the mean of *test_out_i* to form *test_out*.

We used three models: GBDT, RF and AdaBoost for stacking, thus created three different *train_out* and *test_out*. Then, we ran LR on these three *train_out* and *test_out* to get the final prediction. By using Stacking on track 1, we obtained public/private score=0.967305/0.967956.

4.3 Predict Track 2 by Results of Track 1

A naïve way to convert probability into track 2's 0/1 prediction is using 0.5 as threshold. Dropout probabilities less than 0.5 are predicted as 0, otherwise 1. However, we may obtain a better result by finding a more appropriate threshold. Our approach was to sort track 1's best result (with the highest public score) together with track 2's best result by track 1's prediction. We wanted to set the threshold at point p where enrollments were mainly predicted as 0 for track 2 while track 1's predicted probabilities are less than p.

Our implementation was:

1. Define sorted enrollments as e_1,e_2,...,e_N, where e_1 has the lowest probability in track 1's prediction.
2. Change track 2's prediction from 0/1 to -1/+1.
3. Define $sum_i = \sum_{n=1}^{i}$ {e_n's predict value in track 2}, and $t = \arg\min_i sum_i$
4. Set threshold p = {e_t's predict value in track 1}

By setting a more accurate threshold, we improved the performance from public score=0.884253(threshold=0.5) to public score=0.885174.

5. EVALUATION

In this section we evaluate the efficiency, scalability and performance of each model. Our experiments are conducted on a single machine with 24-core processor and 32GB RAM running Ubuntu 15.04. The code files and logs are available at our github repository [4].

5.1 Efficiency

Table 16 shows the fitting time of each model spent on training data. For most models, they spent more time as N increases, with some strange results. We think these unexpected results may result from the unstability, or sudden overloading of our workstation. From Table 16 , we can find that linear models, such as linear SVM and LR, cost much less time compared to other models, while RF and SVM cost most time among these models. Because SVM needs to construct testing data with RBF kernel, it costs most predicting time compared to others. “Paralleled RF” means RF paralleled with five cores while “RF” is not paralleledd.

Method	Fitting time (s)	Predicting time (s)
LR	1.075121	0.004284
Linear SVM	27.084655	0.002636
SVM	1242.82098	81.870829
RF	1116.150098	2.638958
Paralleled RF	526.611482	3.127591
AdaBoost	148.405722	1.679088
GBDT	853.110712	0.202299

Table 16: Efficiency of each model

5.2 Scalability

We divide all training data into 9 groups, and test the fitting time of each model with $\frac{N}{9}$, $\frac{2N}{9}$, ..., N enrollments. The following figures show scalability of each model described in Section 3. Figure 5.2 shows the time each model spent. Figure 5.2 and Figure 5.2 show the ratio of time each model spent compared to running $\frac{N}{9}$ training data, with SVM and without SVM respectively. SVM has the poorest scalability compared to other models while logistic regression has the best scalability. However, paralleled RF has poorer scalability compared to unparalleled RF.

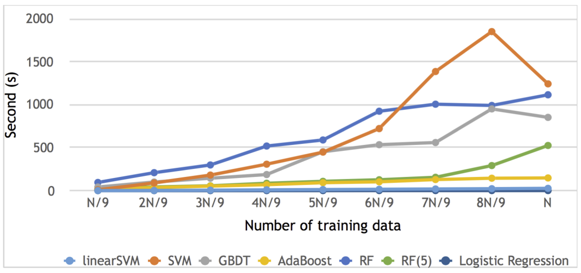

Figure 1: Running time of each model with different number of training data

5.3 Performance

Table 17 shows public score and private score of our models with best parameters in Section 3. Since SVM and linear SVM can not output probability, we only evaluate them in track 2. In conclusion, GBDT outperforms all the other models in public score and private score in track 2. In this table, we could also find that although linear models cost less time when fitting training data, their performance are not as good as the other ones.

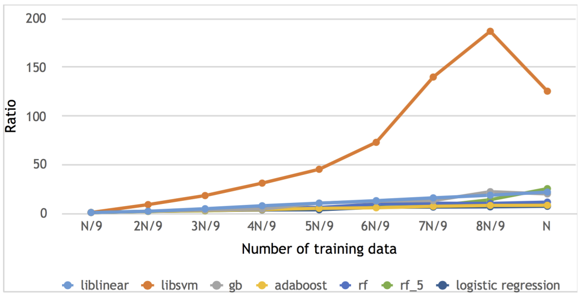

Figure 2: Ratio of time each model spent with different number of training data

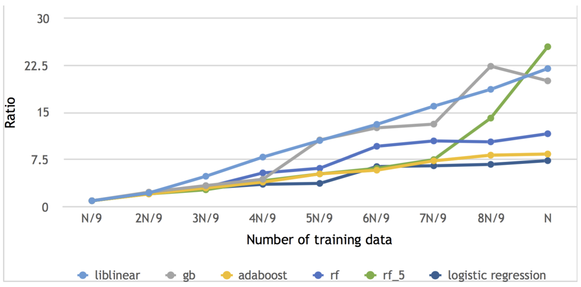

Figure 3: ratio of time each model spent with different number of training data (without SVM)

Method	track 1		track 2	
	Public	Private	Public	Private
LR	0.942858	0.94606	0.861871	0.871228
Linear SVM	N/A	N/A	0.870710	0.878081
SVM	N/A	N/A	0.877173	0.881194
RF	0.96482	0.967951	0.882287	0.886508
AdaBoost	0.965918	0.965125	0.879256	0.886560
GBDT	0.967517	0.967517	0.884398	0.888709

Table 17: Performance of each model in track 1 and track 2

6. CONCLUSION

We elaborate many useful features in this competition, and compare the efficiency, scalability, performance between different models. By applying blending and Stacking methods, we achieve private score = 0.967723 (rank 18) in track 1 and private score = 0.890957 (rank 8) in track 2 on the final scoreboard.

7. REFERENCES

[1] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[2] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. J. Mach. Learn. Res. 9 (June 2008), 1871-1874.

[3] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. ACM Trans. Intell. Syst. Technol. 2, 3, Article 27 (May 2011), 27 pages.

[4] ML-Final-Project. https://github.com/ylc1218/ML-Final-Project