Final Report

1. Abstract

The main approach has two parts in our report. The first part is the idea of data preprocessing; and the second part is the analysis of the performance on those preprocessing data in our chosen algorithms.

2. Preprocessing Data

First we define some term in our report:

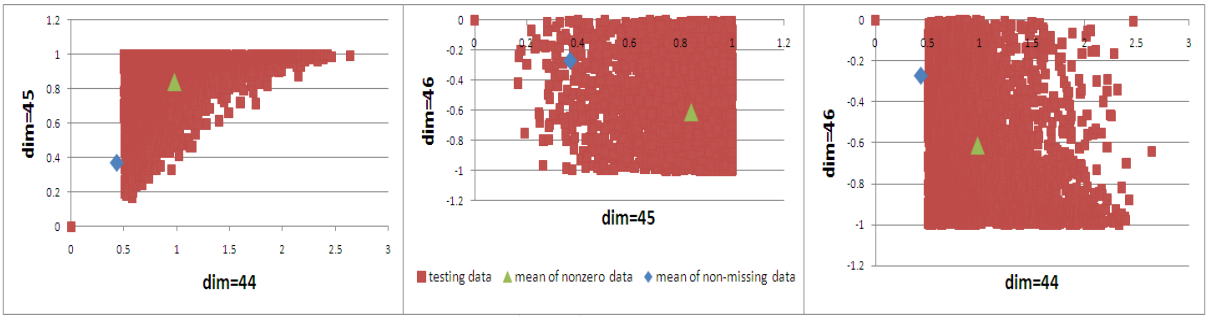
nonzero data = all data – missing data – zero-value data

non-missing data = all data – missing data

We briefly list our finding follow:

Data Type	Data Dimension												
	20	21	22	29	44	45	46	47	48	49	50	51	55
Missing	68%	68%	68%	58%	28%	28%	28%	0%	0%	0%	0%	0%	36%
Zero-Value	0%	0%	0%	42%	**43%**	**43%**	**43%**	100%	100%	100%	100%	100%	64%
None-Zero	32%	32%	32%	0%	29%	29%	29%	0%	0%	0%	0%	0%	0%

(1) In dimension = {44, 45, 46}, there are about 28% data are missing, **43% data are zero-value data**, the others are non-zero data with a noticeable pattern. The zero-value data seems strange to us just like missing data, see figure follow:



Therefore we design several different preprocessing strategies listed below:

(a) Use original data directly as training data as control group.

(b) Ignore the values of dimension = {29, 55, 47, 48, 49, 50, 51}. i.e., using the remaining 71-dimensional data as training data.

(c) Ignore the values of dimension = {29, 55, 47, 48, 49, 50, 51, 20, 21, 22, 44, 45, 46}. i.e., using the remaining 65-dimensional data as training data.

(d) Ignore the values of dimension = {29, 55, 47, 48, 49, 50, 51}.

If dimension = {20, 21, 22}, for each dimension, fill the values of missing data with the mean of nonzero data. If dimension = {44, 45, 46}, for each dimension, fill the values of **missing data and zero-value data** with the mean of **nonzero data**.

(e) Ignore the values of dimension = {29, 55, 47, 48, 49, 50, 51}.
If dimension = {20, 21, 22}, for each dimension, fill the values of missing data with the mean of nonzero data. If dimension = {44, 45, 46}, for each dimension, fill the values of **missing data** with the mean of **non-missing data**.

3. Algorithm 1: Support Vector Machine

The tool we use is *LIBSVM*. We use the *easy.py* in *LIBSVM* to do our learning task, so every setup parameters (ex: scope of grid search) are the same with easy.py; and every learning parameters (ex: cost *C*) are fully determined by *easy.py*'s procedure, we also provide those parameter we use in reference page.

We use the strategies above to do learning task on Track400 and Track4000, the reason why we didn't try the LARGE set because the *efficiency* issue. We will use other algorithm to hit that set. Here is the result of testing error (using the half of TEST on the website as test data):

Training set	Preprocessing strategy				
	(a)	(b)	(c)	(d)	(e)
SMALL (Track400)	32.32%	32.26%	33.9%	33.3%	33.34%
MEDIUM (Track4000)	**30.06%**	30.28%	30.48%	30.28%	30.34%

The argument of those results is:

Arg. *C*/*g*	Preprocessing strategy				
	(a)	(b)	(c)	(d)	(e)
SMALL (Track400)	8/ 0.0078125	8/ 0.0078125	8/ 0.03125	8/ 0.03125	8/ 0.03125
MEDIUM (Track4000)	32/ 0.001953125	32/ 0.001953125	8/ 0.0078125	8/ 0.0078125	8/ 0.0078125

4. Algorithm 2: Logistic Regression

The tool we use is *LIBLINEAR*. All the parameters are set in default value except for cost parameter *C*, we simply do cross-validation in *C* = {1, 2, 4, 8, 16, 32} and choose the best *C* as our learning parameter. Here is the result of testing error, *value is the best result in that set and boldface value is best result in this algorithm:

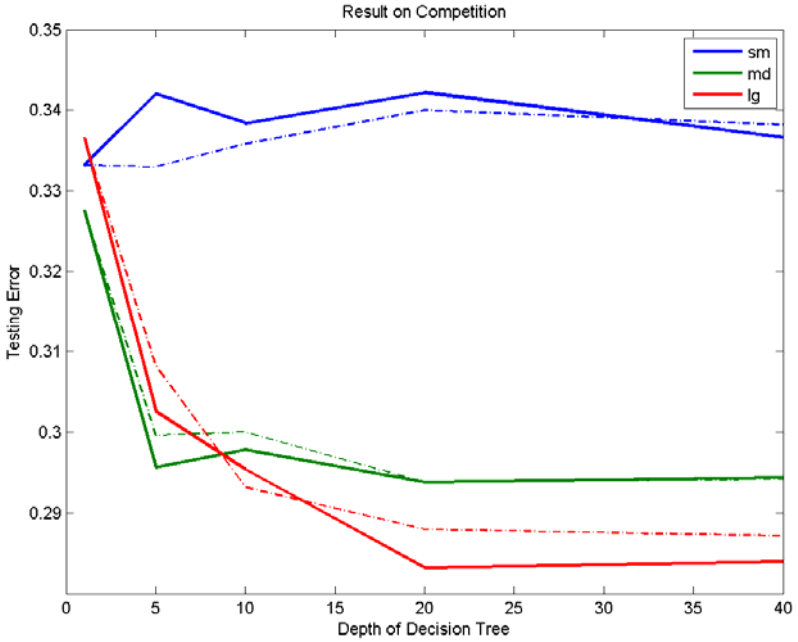
Training set	Preprocessing strategy				
	(a)	(b)	(c)	(d)	(e)
SMALL (Track400)	*31.84%	31.94%	34.42%	32.24%	32.18%
MEDIUM (Track4000)	29.64%	29.64%	30.38%	**29.64%**	29.70%
LARGE (Track40000)	29.80%	29.76%	29.84%	29.76%	29.76%

5. Algorithm 3: Adaptive Boosting

We use *GML-Matlab-Toolpackage* to implement the algorithm. The algorithm we chose is 'Modest AdaBoost'[1] to prevent the overfitting behavior. First we use 5-fold cross validation to choose the depth of tree node in *N* = {1, 5, 10, 20, 40}. The *N* we chose is 20. We didn't validate the iteration time because we believe this algorithm won't overfitting the training data. However, in our experiment, we find that it would overfitting at some case. Here is the result of testing error with iteration *T* = 100:

Training set	Preprocessing strategy				
	(a)	(b)	(c)	(d)	(e)
SMALL (Track400)	34.00%	34.00%	33.48%	33.7%	33.64%
MEDIUM (Track4000)	29.38%	29.38%	29.64%	*29.58%	29.99%
LARGE (Track40000)	28.80%	28.80%	***28.64%**	29.66%	29.58%

In the other hand, we also curious about teacher say that ``AdaBoost won't have overfitting behavior". So we did a test with *T*=100 (dash line) and *T*=1000 with preprocessing strategy (a), and find that AdaBoost did overfit in some cases, especially in SMALL set; see the picture follow:



6. Conclusions

We try five data preprocessing methods; three different scale of training data and use three algorithms to predict the testing set with 10000 data. We have some conclusions:

(1) Consider the different training set, we have lowest testing error in LARGE set.

(2) Consider the different preprocessing strategies we have lowest testing error in (c).

(3) Consider the different algorithm; we have lowest testing result in AdaBoost.

Therefore, the best strategy of the testing data is that ignoring all the missing-data and zero-data features and use AdaBoost algorithm.

7. Workload Balance

Chun-Wei Liu: AdaBoost tuning; report Integration.

Che-Han Chang: SVM, Logistic Regression tuning; preprocessing strategies designed.

8. Reference

[1] Alexander Vezhnevets, Vladimir Vezhnevets 'Modest AdaBoost' - Teaching AdaBoost to Generalize Better. Graphicon-2005, Novosibirsk Akademgorodok, Russia, 2005.