Machine Learning CS 726, Fall 2011 *Christoph Schulze, Michael Whidby, and Kotaro Hara*

Project 2: Complex classification

1 WU1

We ran our experiments with 100 iterations. Based on our analysis the convergence threshold is close to a *stepSize* of 6.5. For values *stepSize* ≤ 6.5 it converges. For values above this threshold it diverges.

StepSize	Result
0.4	2.87E-07
1.0	0
6.3	3.58E-05
6.4	0.001761
6.5	0.073298

Table 1: Examples of convergence

StepSize	Result
6.7	81.61
6.8	2237.36
7	1192445
8	2.94242797e+17
10	5.09612299e+33

Table 2: Examples of divergence

2 WU2

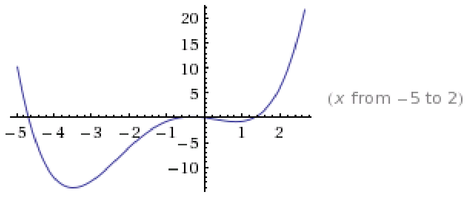


Figure 1: Plot

$f(x) = 0.25x^4 + x^3 - x^2 - x$

$f'(x) = x^3 + 3x^2 - 2x - 1$

The global minimum of the function is at $x \approx -3.49086$ and the local minimum at $x \approx 0.83424$. For both runs we use a *stepSize* of 0.2 and 100 iterations.

1. Local minimum with start at $x = 1$:

```
x, trajectory = gd.gd(
   lambda x: ((1/4)*pow(x,4) + pow(x,3) - pow(x,2) - x),
   lambda x: (pow(x,3)+3*pow(x,2)-2*x-1), 1, 100, 0.2
)
```

2. Global minimum with start at $x = 2$:

```
x, trajectory = gd.gd(
   lambda x: ((1/4)*pow(x,4) + pow(x,3) - pow(x,2) - x),
   lambda x: (pow(x,3)+3*pow(x,2)-2*x-1), 2, 100, 0.2
)
```

3 WU3

N/A

4 WU4

Highest weights

- "graphics": 1.09266018867492675781
- "images": 0.72071808576583862305
- "image": 0.72011238336563110352
- "card": 0.71161371469497680664
- "xx": 0.69300860166549682617

Lowest weights

- "motif": -1.21472561359405517578
- "window": -1.15353131294250488281
- "server": -0.95007282495498657227
- "list": -0.88857728242874145508
- "x": 0.86317312717437744141

These seem pretty "right". We see that some graphics-related words (graphics, images, image) are highly weighted, and words you might see in the windows list (window, server, list) are lowly weighted. It is interesting, however, that "x" and "xx" are on the opposite sides of the weights - maybe this is some type of notation that the newsgroups use for some purpose. Also strange is the appearance of "motif" as the lowest weight.

5 WU5

1

N x

N graphics

N usr

L 0 9

L 45 29

N vga

L 13 0

L 67 328

N window

N be

L 1 44

L 8 20

N motif

L 1 29

L 449 134

Here we see some of the same features as compared to the linear regression model, including “graphics”, “window”, “x”, and “motif”. There are some new features however, “vga”, “usr”, and “be”. I can see “vga” and “usr” being useful, but “be” is confusing - perhaps it is something that is used an abbreviation in one of the newsgroups.

With a depth 10 tree, there is a test error of 20.5%, a slight improvement over the tree of depth 3.

6 WU6

With FastDT, we expected overfitting as the maxdepth increased. With megam, we expected that very small lambda values would perform worse than larger ones. To be honest, we were unsure what to expect with the C values.

Figure 2 shows the megam error rate, which basically agrees with our hypothesis - the small values perform worse than the larger lambda values, with an increase in error rate if the lambda values get too big. Figure 3 shows the FastDT error rate. As expected, FastDT overfits as the maxdepth parameter increases. Figure 4 shows the libsvm error rate. There was actually hardly any variation in the error rates with different lambda values.

7 WU7

All three of the algorithms gets an error rate of 7% on the digit recognition using the default hyperparameter values provided in the project description. For the text categorization task, megam performs the best using

Figure 2: Plot of megam error rate with various lambda values

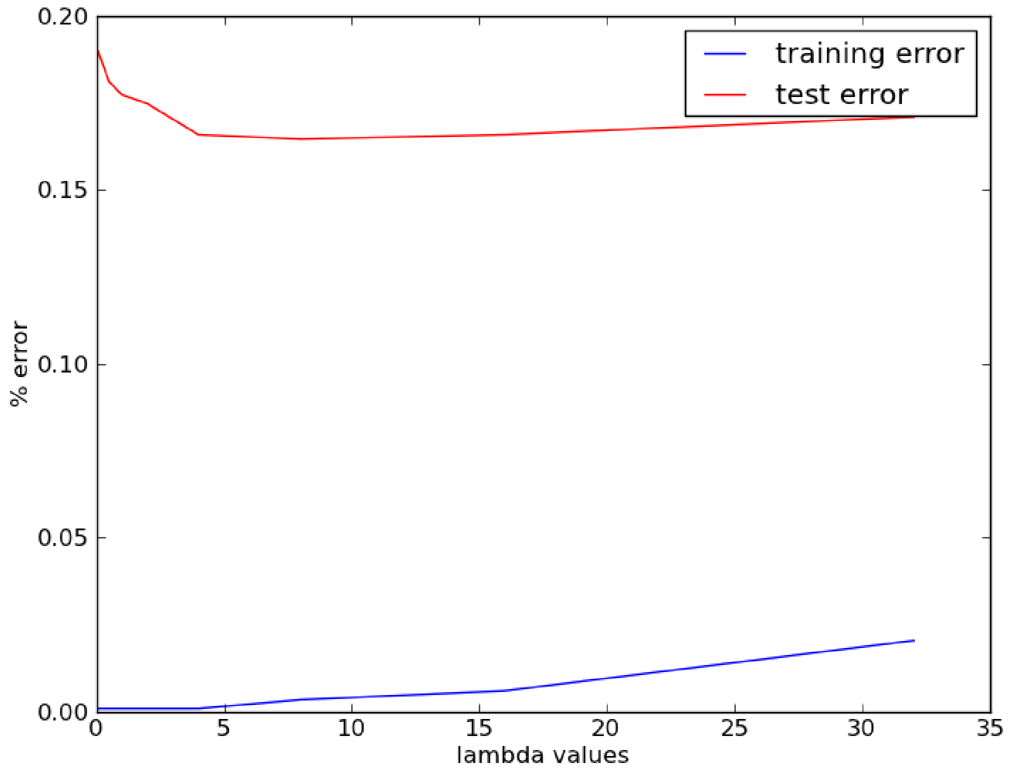


Figure 3: Plot of FastDT error rate with various maxdepths

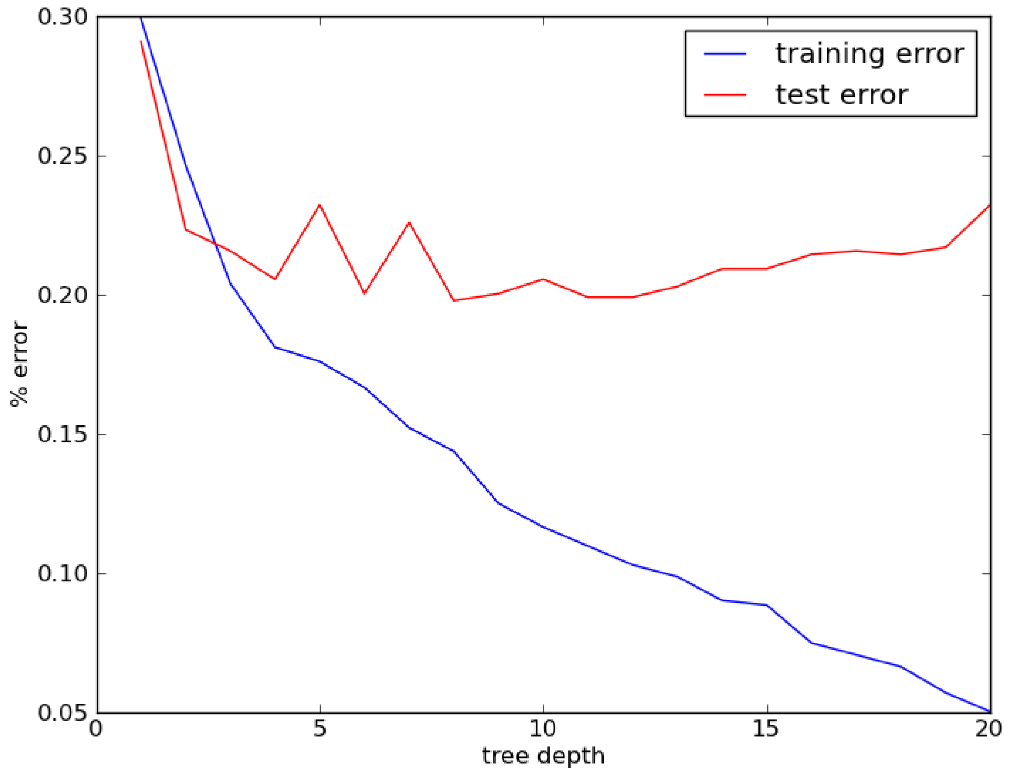
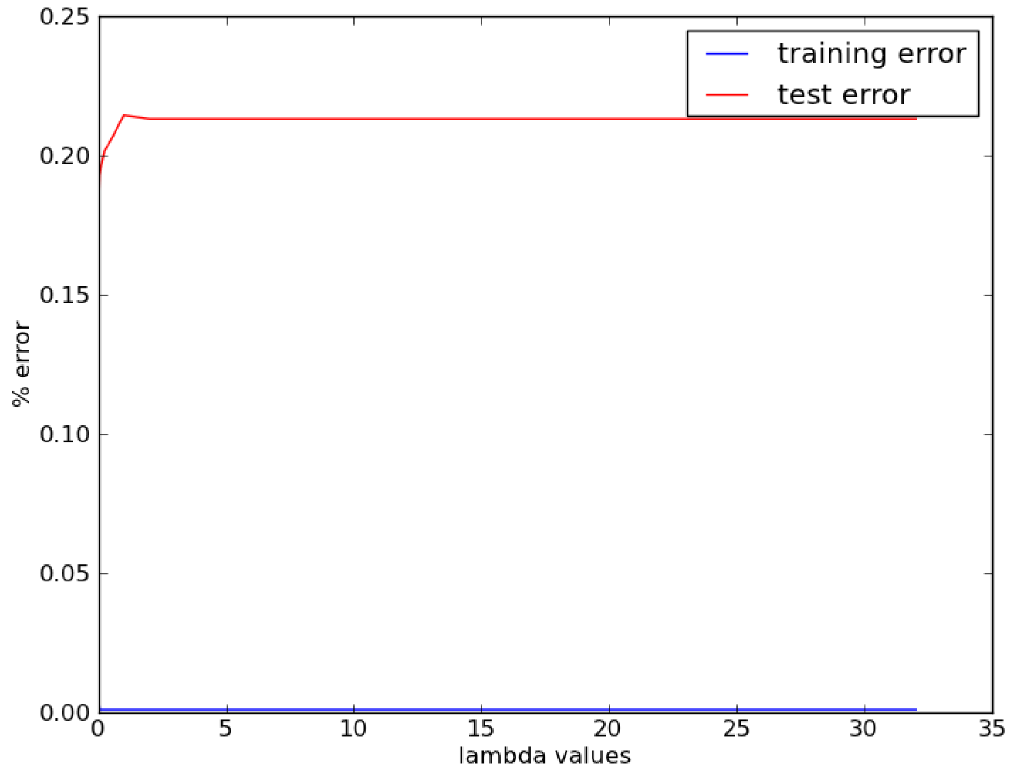


Figure 4: Plot of libsvm error rate with various C values



the default hyperparameter values, with an error rate of 17.7%

Megam and libsvm perform better than FastDT. This is because with a DT of maxdepth = 3, we are only looking at a small number of features (in this case, words), when the text categorization task involves looking at the word counts. On the other hand, megam and libsvm are able to predict by looking at all the input data.

8 WU8

OVA performed best with lambda = 2, with 85.6% accuracy. AVA performed best with lambda = 4, with 84.5% accuracy. Tree performed best with lambda = 4, with 84.9% accuracy.

9 WU9

"Tree-alt" performed best with lambda = 2, with 83.9% accuracy. It performed slightly worse than the original tree, which can be expected since it is making the harder distinction first.

10 WU10b

We decided to work on collective classification. We use a CiteSeer dataset shared on the Statistical Relational Learning Group's webpage. (http://www.cs.umd.edu/projects/linqs/projects/lbc/index.html). The following description of the dataset is excerpted from the webpage. *The CiteSeer dataset consists of 3312 scientific publications classified into one of six classes. The citation network consists of 4732 links. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary. The dictionary consists of 3703 unique words.*

We implement a stacking algorithm that is described at the end of the chapter 5 of the textbook (Algorithms 20 and 21.) We implement two flavors of the stacking algorithm (to see which one performs better):

- Simple. This implementation will only use predictions on neighbors from a previous layer. (Neighbors are examples that cites/are cited by the example that we are focusing on)
- Cumulative. This implementation will take all the predictions on neighbors until the Kth layer $(\boldsymbol{y}_1, ..., \boldsymbol{y}_{k-1})$ to train the Kth layer's classifier.

On the first layer, we simply use Megam in our script to train a multi-class classifier. Output (predictions) from Megam is parsed and stored inside prediction data. On each of succeeding layers, we first generate training data by combining an original training data (training data from the first layer) and predictions from a previous layer. I.e. we append predicted labels of neighbors into the training example as features. Then we train a classifier for the layer.

We split the dataset into five parts for cross-validation and calculate training errors and test errors for each layer. For each validation, we train classifiers with 2650 examples and test with 662 examples.

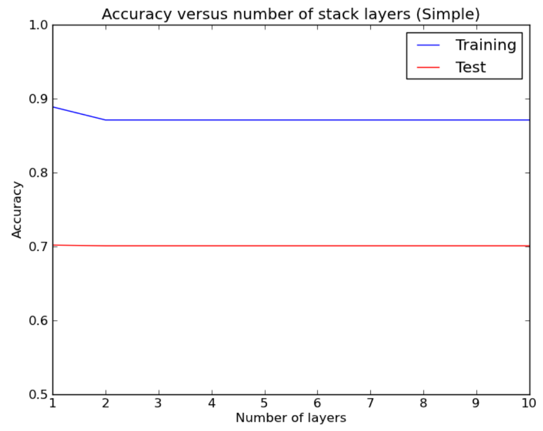


Figure 5: Simple

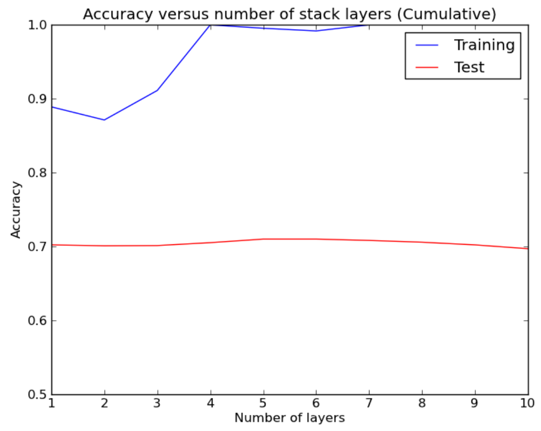


Figure 6: Cumulative

Figures show training errors and testing errors against layer indexes. On the first layer of the simple implementation, training error was about 11%. The classifier is trained only with word features. On the second layer, the error went up to 13% as we append predictions from the first layer into the training dataset. For the rest of the layers, training error did not change from 13%.

The testing error was 30% on the first layer. Testing error did not change more than 0.1% for any of following layers.

The training error on the cumulative implementation was 11% on the first layer. On the second layer, it went up to 13%. After the third layer, training error started to decrease, and on the fourth layer, the training error became almost 0. Then the training error stayed close to 0.

The testing error was 30% at the beginning, but it showed a slight improvement around layer 5 and 6. Errors went down to 29%.

Both simple and cumulative implementations of the stacking algorithm did not show a significant improvement in test accuracy. We believe there are several reasons:

- Dataset's network is sparse. There are more word features compared to neighbor features. Therefore neighbors' labels are less influential for classification.
- Neighbors' labels do not contain much information. Collective classification assumes neighbors' labels are informative because ML papers are more likely cited by other ML papers. However, in this dataset, AI paper can certainly cite ML papers, and so for others.

Another problem. Scalability (related to overfitting from the cumulative version)

Conclusion. Stacking did not really help.