Distributed Meta Classifiers

Ang Huey Ting , Li Guoliang

1 *Introduction*

We are entrusted with the task of developing an application that runs on Grapevine, an Internet Distributed Computing architecture. After much consideration, we settled on the idea of building a meta-classifier, a classifier built on top of a set of classifiers, for the following reasons.

- Meta-classifiers have long proven to give better performance over its component classifiers.
- Meta-classifiers are computationally intensive to build as it involves the building of a set of component classifiers. Thus a parallel version is desired to speed up the process.
- The component classifiers are built independently, requiring no communications between each other in the process. Thus it is suitable for the Distributed Computing environment of Grapevine.

We envisage that such an application (together with Grapevine) will appeal to researchers working in the machine-learning domain who are interested to experiment with meta-classifiers. As far as we know, there are machine learning toolkits freely downloadable from the web but none of them successfully implemented meta-classifiers in the parallel mode.

2 *Project Overview*

Our project can be divided into the following two portions:

- Building a Distributed Meta-Classifiers (DMC) on Grapevine.
- Experiments and Analysis of the DMC in the Text Categorization domain.

It can be further summarized into the following stages:

- Literature Survey on Meta-Classifiers
- Identify machine learning toolkits that can be ported to Grapevine
- Preprocessing of the Data in Reuters21578 which is a benchmark corpus used in Text Categorization
- Preliminary experiments on short listed machine learning toolkits.
- Study the selected machine learning toolkit for porting
- Study the Grapevine architecture.

- Building the DMC.
- Experiment on the DMC with the Reuters21578 corpus.

3 *Meta classifiers*

3.1 Classifiers

In the simplest form, a classifier is a function that maps an instance into a Boolean.

Classifier(instance) = {True, False}

For example, in Text Categorization, for a given category, the classifier determines if any document is relevant to the category. Such text categorizing classifiers have wide application in today's world where digital documents are abundant. For instance, one may build a web crawler with classifiers functionality that will craw documents from the web and automatically classified the documents into user-defined categories like "finance", "sports", "politics" etc.

A popular approach of building classifiers is that of using machine learning tools, such as neural network, Bayesian network and support vector machine. In this approach, the classifier will be trained by applying a machine-learning algorithm on a set of instances that have been tagged as *true* or *false*. This set of instances is known as the training set.

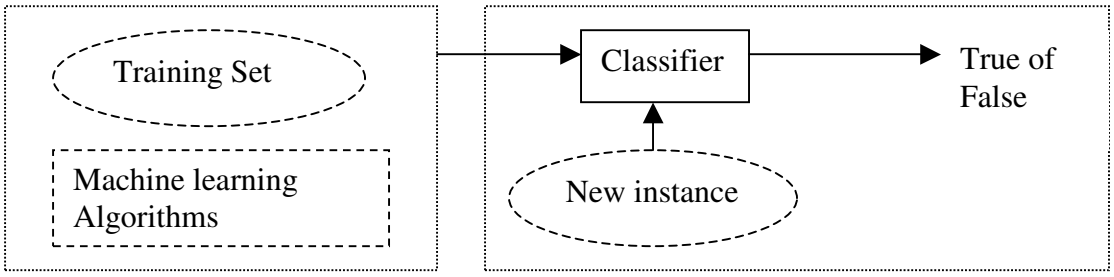

Figure 1: Structure of Classifier

After training, the classifier is ready to classify new unseen instances. Depending on the complexity of the machine learning algorithm, size and quality of the data set and parameter settings, the training process can take from a few seconds to a few days. In our experiment, it took 6 hours to build a classifier using the Neural Net in Weka for identifying documents belonging to the category of "copper".

There are machine-learning toolkits that are freely downloaded from the Web, which will form the basis of our DMC.

3.2 Meta-Classifiers

Meta-classifier, in our context refers to classifier built on top of a set of classifiers. There are several ways of building meta-classifiers. For example, "boosting" involves iterative retraining of the same classifiers through the same training data. At each iteration, the weight of each training instance is adjusted to improve the classification results.

We focused on the approach of using an ensemble of classifiers that are built independently and relied on a voting mechanism to compute the final output. This process is described below. Previous work using such techniques has reported good results. Furthermore, the component classifiers are built independently in this approach and thus require no communication between the nodes during the parallelization.

- Build a set of classifiers.
- Send new instance to each classifier.
- Meta-Classifier combine results from ensemble of classifiers.

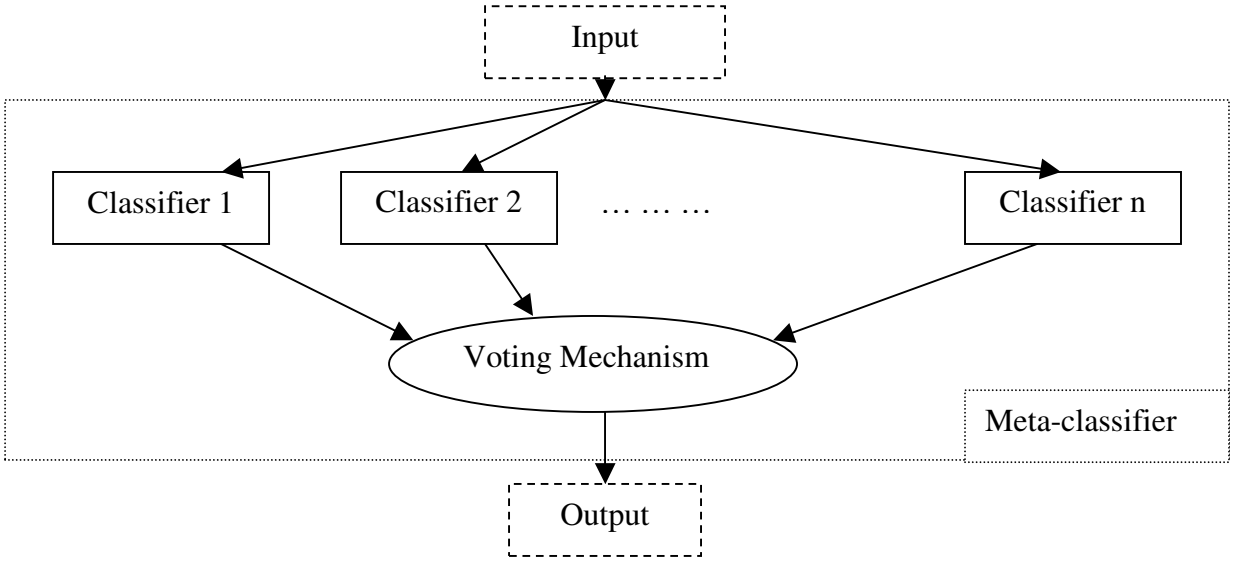

Figure 2: Structure of a Meta-Classifier

As can be seen from Figure 2, building a meta-classifier involves the training of a set of (say n) classifiers. Suppose each of these classifiers takes an hour to train, the meta-classifiers with n component classifiers will take n hours to train. In our experiment, it tool 6 hours to train a single classifier on around 1.5 M of data. Thus parallelising the process is essential!

3.3 Component Classifiers

There are several ways to generate the component classifiers but we will describe the two that we implemented, namely,

- Generate different data sets from the original train set by random sampling with replacement. This technique is commonly known as bagging.

- Use different machine learning algorithms to build the component classifiers.

Generally speaking, errors are random. By combining the results from several classifiers, the error pertaining to each component classifier can be singled out and removed. Thus we require the component classifiers to be different from each other so that the errors generated are varied too.

In bagging, it is desirable that the machine learning algorithm used is unstable. An unstable algorithm is sensitive to changes in its training set. In bagging, we generate different bags of data with different characteristics. As a classifier depends on the training set, the classifiers trained will be varied thus improving the performance of the meta-classifiers.

For the second way of generating component classifiers, we need to use different algorithms to generate classifiers of different favors.

3.4 Voting Mechanism

The voting mechanism we used is straightforward, that is majority wins.

4 Distributed Meta-Classifier

To build a Distributed Meta-Classifier (DMC), we require the following

1. A distributed computing environment
2. A machine learning toolkit

For the computing environment, we are using the Grapevine. As for the toolkit, we need to search and explore the packages online that are freely downloadable.

4.1 Toolkits

There are two popular comprehensive machine learning toolkits that are freely available:

1. WEKA - machine learning packages in Java, by Waikato University, New Zealand
2. MLC++ - machine learning library in C++, by Stanford University

Both toolkits are comprehensive as they include implementation of most popular machine learning algorithms. Using algorithms from the same package implies that we have a standardized input and output format. This convenience, however, comes at the expense of the algorithms sharing a large common of base that are non existence if the algorithms are built independently. It hinders the porting of the algorithms because the porting of single algorithm involves the porting of a huge number of classes.

We have downloaded both Weka and MLC++. Weka is suitable for our purpose as it is implemented in Java. We also explored MLC++ with the attempt of porting it to openMP or MPI. We managed to install MLC++ on linux but not without tweaking the makefile and bypassing some library installation. As we are unsure of the effect of the modification to the performance of the package, we finally gave up the idea of porting it to openMP. Nevertheless, we did successfully compile an independent C implementation of "QuickProp" and have run some preliminary experiments. "QuickProp" will be treated in more detail in Section 5.

Our experience with playing with these toolkits suggested that packages in C/C++ is much more difficult to port than those implemented in Java. We will give a brief description of our experiments on the toolkit QuickProp in section 5.

4.2 Framework

The two forms of Distributed Meta-Classifiers that we are building are as shown in figure 3 and 4. There are two parts to each meta-classifier, the server and the clients. As we use the Grapevine, the server and clients need to be set up on the machines that we are running on.

In the meta-classifier implementing bagging (figure 3), the server part of the meta-classifier will generate the different bags of training data, send the bags and classifiers to the clients, collect back the component classifiers from each client and build the meta-classifiers.

In the meta-classifier implementing different algorithms (figure 4), the server part of the meta-classifier will send the same training set and the different classifiers to the clients, collect back the component classifiers from each client and build the meta-classifiers.

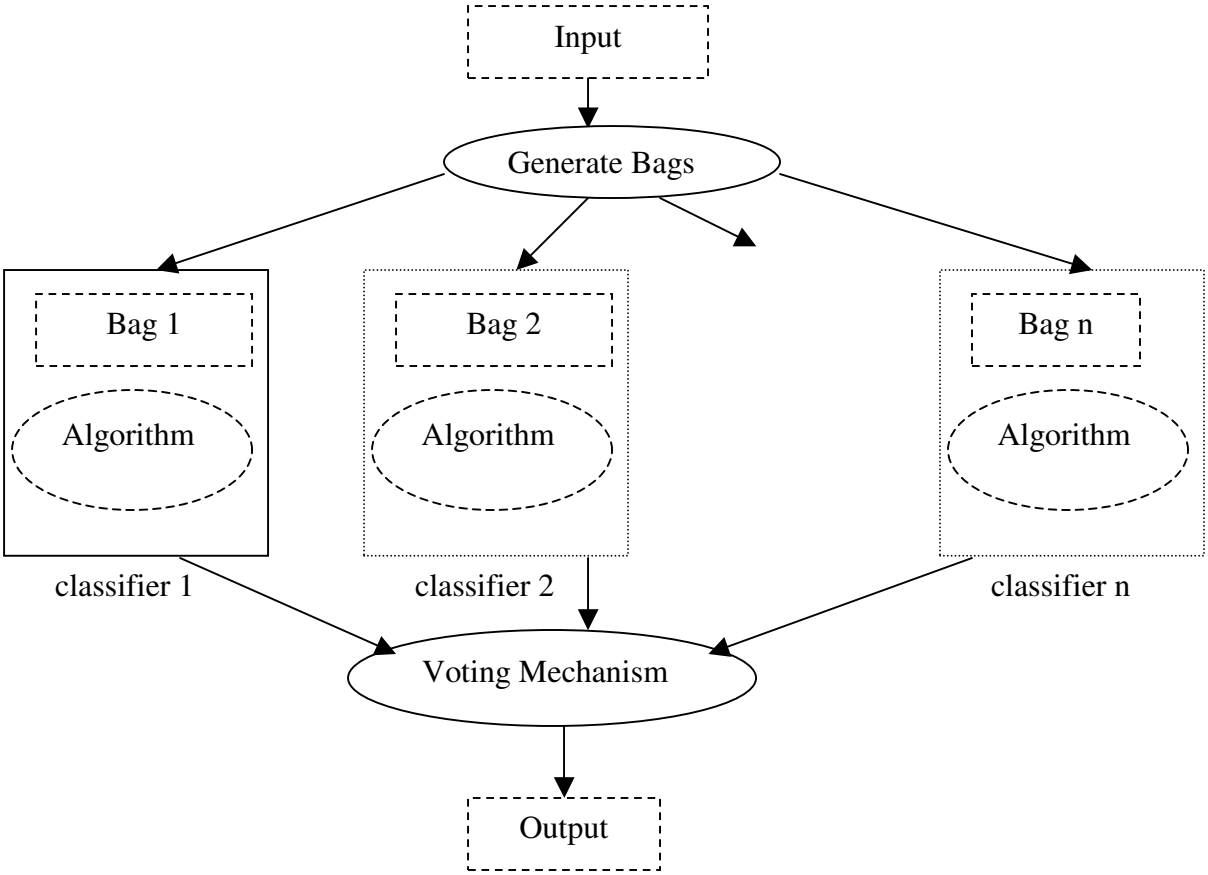

Figure 3: Structure of a Meta-Classifiers with Bagging

In our actual implementation in grapevine, the component classifiers are built on each client and test instances are sent to the client for classification and the classification results are return back to the server. This approach is adopted as the current version of Grapevine can't support the passing of an object of a user-defined class.

4.3 Issues

We envisage that if grapevine is deployed over the Internet, there is a possibility that different client require different length of time to complete the jobs assigned to them. There may also be cases of packet loss or disconnection etc. If a particular client takes particular long time to return the results or never returns it at all, it may actually slow down the entire process of building the meta-classifiers.

In view of this, we have developed the concept of a "time out". What it is essentially doing is to generate more jobs and send to different clients, it will then collect whatever is returned within the "time-out" period and use them to build a meta-classifier.

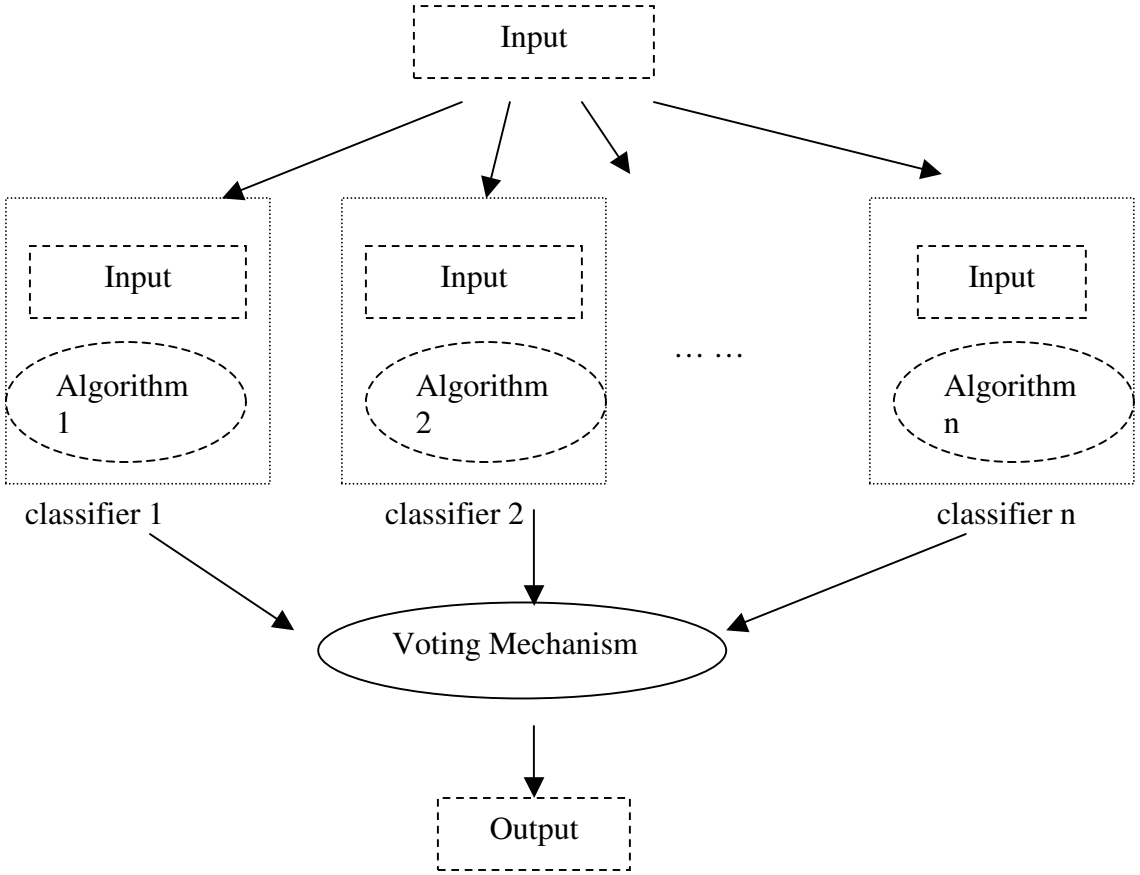

Figure 4 : Structure of an Meta-Classifiers with Different Algorithms

5 *Experiments*

We tested our distributed Meta-Classifiers in the domain of Text Categorization. We chose this domain as we see the potential and need for an automated text categorization in this digital document age. Moreover, for this task, a benchmark corpus (Reuters21578) is available for free download, and many machine learning algorithms have been used on this corpus (Yang and Liu, 1999).

5.1 Text Categorization

Imagine building a text categorization system (as in Yahoo, but involves not only websites but all documents residing on the machines in the environment). On each machine there is an individual classifier that will train on the documents residing on that machine. As the classifiers are trained on the documents residing on each machine, the classifiers can be sent over to other machine to classify the documents on that machine.

5.2 Dataset – Reuters21578

We are using the Reuters21578 corpus, which is a benchmark corpus for Text Categorization for our experiments. It can be downloaded from http://www.research.att.com/~lewis/reuters21578.html

We used the ModApte split of the data that consists of 9603 training texts and 3299 test texts spread across 90 over categories. It is the test corpus used in the ACM SIGIR 1999 paper "A re-examination of text categorization methods" –http://www-2.cs.cmu.edu/~yiming/publications.html

We have done the necessary preprocessing to the dataset, which consists of the following steps:

- Extract the train and test texts from the corpus.
- Remove SGML tagging from the files.
- Removal of stop word like is, are, this, that.
- Transform the words into their morphological root form. Note that we employ the porter's stemming algorithm to do the stem the words.
- Feature extraction, we implemented a simple feature selection algorithm that extracts the top n features for each category based on their correlation metric.

5.3 Preliminary Experiments

We did some experimentation on the feasibility of our concept with MPI before the Grapevine architecture is available for integration.

5.3.1 Experiment with MPI version of Quickprop

This version comes from the file quickprop.c, and is ported to MPI. Also bagging algorithm is added. The description of the algorithm is as follows:

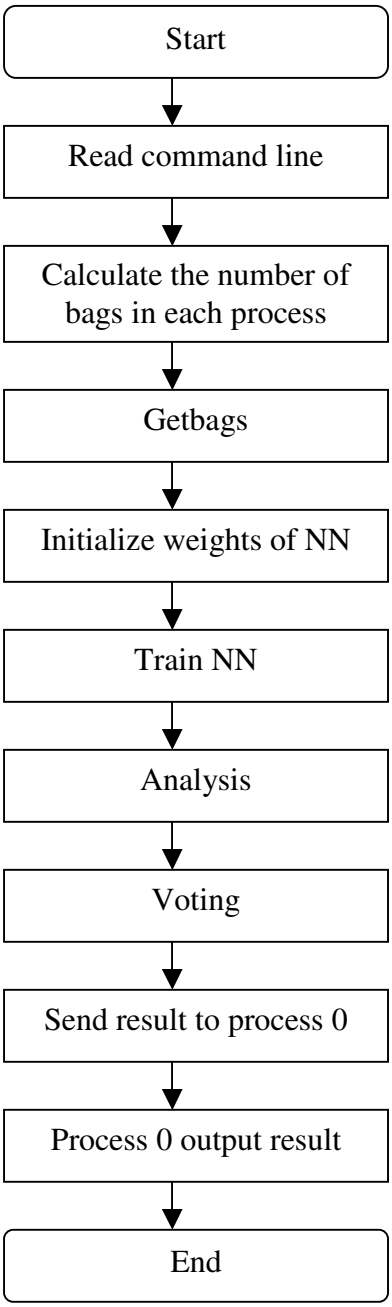

Figure 5 the flow of the algorithm

Here are some explanations of the algorithm. Function GetBags will generate training dataset from the original training set. The number of the patterns in the new training dataset is equal to the product of the variable ratio and the number of the patterns in the original training dataset. Function INIT_WEIGHTS will initialize the weights of the neural networks. The initial weights are defined in some range. Function TRAIN will train the neural network with quickprop algorithm. Function Analysis will analyze the training result and get the threshold, which is the best value for some criterion. Function

VOTING will apply the trained neural network and the threshold to the testing dataset, and will get predicted possibility and class for each testing patterns.

There is a special variable *flag* in the program. It is used to vote or not after one neural network is trained. Because some the neural network is over-trained or its parameters are not suitable, it will give strange results. If this happens, function analysis will find it and set flag to 0. It this does not happen, function analysis will set flag to 1. Then the function VOTING will be performed.

There are 6 options to control the program:

"-net" <filename>	the file to contain the network, the training data and the testing data (Required)
"-seed" <int>	the seed of random function (default: 1)
"-e" <int>	the epoch to train the neural network (default: 100)
"-bags" <int>	how many bags will be used (default: 10)
"-o" <outfilename>	the output file name (default: no output file)
"-r" <float>	the ratio of each bag to the whole training patterns (default: 1)

Speedup

If the number of the process is set to 1, we can get the result of the sequential version. If the number of the process is set to a number greater than 1, we can get the result of the parallelized version. The command lines are as follows.

command line for 1 process:

```
mpirun -np 1 n1 ./paim -net e6.net -bags 20 -e 500 > out
```

command line for 2 processes:

```
mpirun -np 2 n1-2 ./paim -net e6.net -bags 20 -e 500 > out
```

In this case, we use a simple data set to test the neural network, and have gotten the results as follows. The time is all the time spent by process 0. So, it includes the training time, communication time and the overhead. Figure 6 and 7 show the time elapsed and the speedup when different numbers of processes are used.

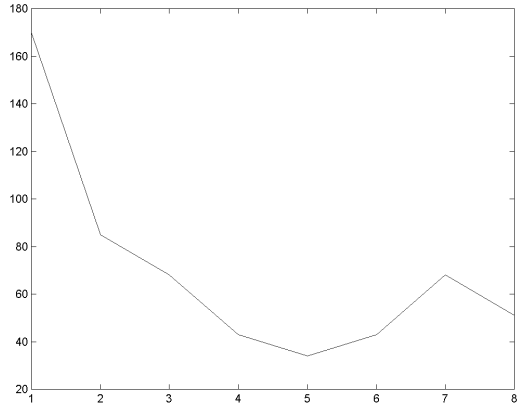

Figure 6 Time elapsed with different processes

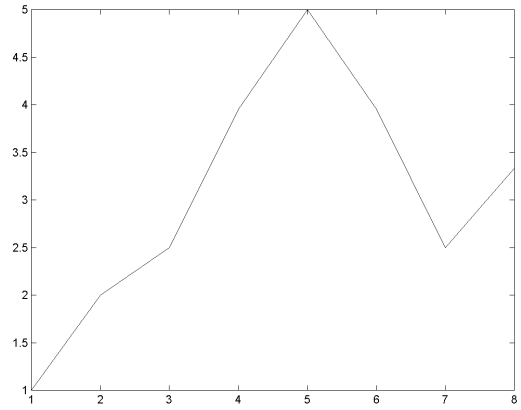

Figure 7 Speedup with different processes

From the graph, we know the speedup of the algorithm is very good when the number of the processes is small. When the number of the processes is big, the speedup is not so good. There are two possible reasons for it. One is the overhead will take more time when the processes increase. The other reason is that the load of the nodes is changed. The second reason can explain why the speedup with 8 processes is better than with 6 or 7 processes.

In this experiment, 20 bags are used. If there are more bags, each process will do more jobs and the calculation time will be more than the overhead. Then the speedup will be better when more processes are used.

5.4 Text Categorization Performance

We have run a series of experiments using the following machine learning algorithms:

- Support Vector Machine
- Naïve Bayes

We had wanted to make use of a few more machine learning algorithms (such as neural network, but due to time constraints, we had to stick to the above two).

Evaluation Criteria

Results are evaluated by using the notions of precision, recall and f-measure: (Rijsbergen 1979)

Precision = number of correct positive predictions / number of positive predictions
Recall = number of correct positive predictions / number of positive examples
F-measure = 2 * recall * precision / (recall + precision)

Bagging: Same classifier with different training data

We have run experiments for all the above two algorithms. For each experiment, we randomly generated n bags of 90% of the training data to train n independent classifiers. It would have been desirable to repeat each of this experiments m times and to take the mean of these m experiments. However, due to time constraints, we have only run one experiment for each n and each algorithm. The results are summarized in the tables below.

These experiments are conducted on the Reuters21578 corpus, using the category "gnp". We have selected this category because it is a small category, and yet not too small (the smallest categories in Reuters contain just a few positive examples!) Using a small category makes it difficult to evaluate the systems accurately (a difference of 1 classification can make a big difference to precision and recall). Using a larger category, however, would mean that almost all the classifiers perform at near 100% precision. This also makes it difficult to illustrate the advantage of the meta-classifier, which should improve the precision of its components.

Table 1 Results for Support Vector Machine

Number of Bags	Precision	Recall	F-measure
2	0.94	0.49	0.64
3	0.84	0.60	0.70
4	0.82	0.51	0.63
5	0.84	0.60	0.70
6	0.91	0.57	0.70
7	0.85	0.63	0.72
8	0.88	0.60	0.71
9	0.85	0.63	0.72
10	0.90	0.54	0.68

Table 2 Results for Naïve Bayes

Number of Bags	Precision	Recall	F-measure

2	1.00	0.23	0.44
3	0.82	0.26	0.37
4	0.88	0.20	0.39
5	0.83	0.29	0.33
6	0.82	0.26	0.43
7	0.83	0.29	0.39
8	0.89	0.23	0.43
9	0.90	0.26	0.36

Different classifiers with the same training data

Results of experiments run by using different algorithms with the same training data (Figure 4) are as follows:

Table 3 Results for the combination of SVM and Naïve bayes

Classifiers	Precision	Recall	F-measure
Support Vector Machine and Naïve Bayes	1.00	0.17	0.29

Due to the fact that these experiments are due to only one trial run for each row of the above tables, the results are not stable. However, in general, we believe that a meta-classifier should always produce higher precision than its component classifiers, although there might be some tradeoff in its recall.

Experiment with Grapevine

Two main programs will be explained here, JobLauncher.java and Bag.java. The first resides on the server while the second is sent from the server to selected nodes. The flow of the algorithm is similar to that of the algorithm in the MPI version of quickprop which can be summarized in the following steps

1. The nodes or clients will register itself with the server. It is up to the server to decide which node will be assigned the task of building the component classifier.
2. The client node should prepare the job first. It will split the job into some independent tasks and submit the job to the server. The submitted job includes the executable codes and the data.
3. The server will distribute the tasks to the volunteer nodes. Every volunteer node will return its result to the server. The server will send the result to the client.

In this architecture, the overhead includes the time to prepare the job, to submit the job, and to collect the result. If the overhead is very much less than the time required to build a classifier, the efficiency of the program might be close to 1.

The second program, Bag.java sent to the volunteer nodes will build the classifier with the data and the specified method: support vector machine or neural network.

In the experiment, we use support vector machine to build the classifier, because it takes less time than neural network, and gives good result.

TP = 4714
FP = 275
TN = 379
FN = 555
precision = 0.94487876
recall = 0.8946669

JobLauncher: 62.8 s
task 0: 26.5 s
task 1: 27.2 s
overhead: 9.1 s (= 62.8 - 26.5 - 27.2)

Figure 8 output of the Grapevine

There are only two volunteer nodes available to do the experiments. So, we set the number of tasks to two. After running the program, we choose one typical result as follows.

In the experiment, we use one node to emulate the volunteer nodes. So, the overhead is equal to the total time minus the time elapsed by the tasks. Assume we run the task on different nodes. The total time will be the sum of the overhead and the longest time the task takes. It will be 36.3 s (= 9.1 + 27.2). Compared with the time took by the sequential version of the program, 57 seconds, the speedup will be 1.57 (=57 / 36.3) and efficiency is 0.785 (=1.57/2). If the volunteer nodes are more enough, the speedup will be better than as the overhead of preparing, sending jobs and collection result will be amortized.

6 *Conclusions*

We have always heard the saying that little is done to bridge the gap between parallel computing and other domains of scientific research. Our application, in some way, is trying to fill the gap between meta-machine learning and parallel computing.

We admit that our version of the application is not a full fledged one due to development time constraint. Nevertheless, we have tested out the feasibility of our concept and demonstrated the possible speedup that the user of this application may achieve.

7 *Future Work*

Use of confidence of a classification by a classifier:

ML algos such as Bayesian outputs a probability with each classification. These probabilities can be used to help decide which classifier should be more reliable than others. A classifier that gives a classification with high confidence should mean that it is less likely to be wrong.

8 *References*

Popular Ensemble Methods: An Empirical Study
http://www-2.cs.cmu.edu/afs/cs/project/jair/pub/volume11/opitz99a-html/paper.html

J. Platt (1998). *Fast Training of Support Vector Machines using Sequential Minimal Optimization*. Advances in Kernel Methods - Support Vector Learning, B. Schölkopf, C. Burges, and A. Smola, eds., MIT Press.

George H. John and Pat Langley (1995). *Estimating Continuous Distributions in Bayesian Classifiers*. In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence. pp. 338-345. Morgan Kaufmann, San Mateo.

Yiming Yang and Xin Liu (1999). *A re-examination of text categorization methods*. In Proceedings of ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'99, pp 42--49), 1999.

C. van Rijsbergen (1979). *Information Retrival*. Butteroworths, London.

Scott E. Fahlman: ``An Empirical Study of Learning Speed in Back-Propagation Networks", Technical Report CMU-CS-88-162, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1988.

Quickprop.c http://www.cae.wisc.edu/~ece539/software/quickprop/quickprop.c

An introduction to support vector machines http://www.support-vector.net/