

This page is a work in progress describing Ensemble Selection, a relatively new and powerful approach to supervised machine learning.

The original paper can be found here:

<http://www.cs.cornell.edu/~caruana/caruana.icml04.revised.rev2.ps>

## **Ensemble Selection in a Nutshell (informal description)**

Over the years people in the field of Machine Learning have come up with all sorts of algorithms and improvements to existing approaches. There are so many unique supervised learning algorithms it's really hard to keep track of them all. Furthermore, no one method is “best” because it really depends on the characteristics of the data you are working with. In practice, *different* algorithms are able to take advantage of *different* characteristics and relationships of a given dataset.

Intuitively, the goal of ensemble selection then is to automatically detect and combine the strengths of these unique algorithms to create a sum that is greater than the parts.

This is accomplished by creating a library that is intended to be as diverse as possible to capitalize on a large number of unique learning approaches. This paradigm of overproducing a huge number of models is very different from more traditional ensemble approaches. Thus far, our results have been very encouraging.

## **Preliminary Results of our WEKA Implementation**

Thus far, we've seen considerable performance increases with our WEKA implementation of Ensemble Selection. The following table shows performance on 15 learning problems from the UC-Irvine dataset repository (7 binary and 8 multi class). These results were obtained by using half as a train set and half as test set. The Ensemble Selection default settings were used without any parameter tuning. Unfortunately, we do not have any error bars as training model libraries is quite expensive and we only had the computing time to try one of each. However, as you can see, it did quite well across the board and performance of the ensembles clearly improved over the “best” library models on all but a few of the datasets. Although the computing time necessary was large, we ran our classifier “out of the box” and thus the human interaction time was really low.

It is definitely also worth noting that even in cases where performance of the ensemble does not improve over the model library – you still get statistics on all the models in your ensemble library. This is useful and interesting in and of itself. We foresee many situations where people would want to see a ranked performance summary of how a default set of Ensemble Selection classifiers did on their dataset.

(For a description of metric loss reduction that is listed in the “reduction” column in the table below, please see section 5.2 of the Ensemble Selection paper.)

Dataset	Error			RMSE		
	Ens. Sel.	Best	Reduction	Ens. Sel.	Best	Reduction
breast-w	0.02	0.017143	-0.166657	0.1108	0.1241	0.107172
credit-a	0.101156	0.101156	0	0.2976	0.3177	0.0632672
credit-g	0.266	0.278	0.0431655	0.4192	0.4253	0.0143428
diabetes	0.197917	0.197917	0	0.3731	0.3797	0.0173821
kr-vs-kp	0.020025	0.020651	0.0303133	0.2435	0.197	-0.236041
mushroom	0.073363	0.082964	0.115725	0.2392	0.2701	0.114402
sick	0.012725	0.012195	-0.0434604	0.0992	0.1163	0.147034

Table 1: Performance of Ensemble Selection for binary-class classification problems. Ens. Sel. is Ensemble Selection. Best is the best model from the ensemble library as chosen based on Validation Set performance.

Dataset	Error			RMSE		
	Ens. Sel.	Best	Reduction	Ens. Sel.	Best	Reduction
ameal	0.002227	0.006682	0.666717	0.0339	0.0449	0.244989
hypothyroid	0.008484	0.008484	0	0.0602	0.0635	0.0519685
letter	0.059667	0.089333	0.332083	0.0658	0.0726	0.0936639
segment	0.025974	0.038095	0.318178	0.0771	0.0918	0.160131
soybean	0.069767	0.119186	0.414638	0.1119	0.1143	0.0209974
vehicle	0.205189	0.21934	0.0645163	0.2563	0.2825	0.0927434
vowel	0.412121	0.456566	0.0973463	0.2217	0.2373	0.0657396
waveform-5000	0.132694	0.131894	-0.00606548	0.2469	0.2512	0.0171178

Table 2: Performance of Ensemble Selection for multi-class classification problems. Ens. Sel. is Ensemble Selection. Best is the best model from the ensemble library as chosen based on Validation Set performance.

## The Ensemble Selection Algorithm

The basic algorithm can be briefly explained in two parts:

The first step is to create a “model library”. This library should be a large and diverse set of classifiers with different parameters. To the best of our ability we throw the kitchen sink at your data – the more the merrier. Keep in mind that it should not really hurt performance to train “bad” models as they will simply not be chosen by the Ensemble Selection algorithm if they hurt performance.

The second step is to combine models from your library with the Ensemble Selection algorithm to. Although there are a lot of very complex refinements to prevent over fitting, the basic idea behind the algorithm is simple. Basically, we start with the best model from our whole library that did the best job on our validation set (held aside data). At this point we have an ensemble with only one model in it. Then, we add models one at a time

to our ensemble. To figure out which model to add, each time: separately average the predictions of each model from the library currently being considered with the current ensemble. Then choose the model that provided the most performance improvement.

### **The Over fitting Problem**

This is a greedy hill climbing approach, and as previously mentioned has a lot of over fitting problems (while performance on the validation set improves performance on the test set degrades). To prevent the problems with over fitting, as described in the original paper, we take advantage of three proven strategies: “model bagging”, “replacement”, and “sort initialization”. For a more in depth description of these three methods, please refer to the original paper.

### **Ensemble Selection: A Slightly More Technical Explanation ([link to this on a separate WIKI page](#))**

Ensemble Selection is an ensemble learning method. The focus of the Ensemble Selection learning algorithm is combining a large set of diverse models in to a high-performance ensemble by determining appropriate weights for the models. The algorithm was inspired by forward-stepwise feature selection, wherein features are added one at a time to greedily improve performance of the model being trained. Here, we greedily add models to our ensemble to optimize for some metric, such as accuracy. Initially, we add the model which has the best performance on some validation data. Then, for some number of iterations, we add the model which, when combined with the current set of models, helps performance the most on the validation data. When we are done, we can consider the number of times a model was added as its “weight”, and our ensemble makes predictions by taking a weighted average over the chosen models.

Some other techniques are used to help performance and avoid over-fitting. One is sort-initialization. The idea is to sort models by their performance, and add the best  $N$  models, where  $N$  is the number of top models which optimize performance versus the validation set. Another important strategy is model bagging. The idea here is to only consider some random subset of the entire library of available models, for example half (default), and run ensemble selection for that subset. This is done some number of times, similarly to Breiman’s Bagging method, and the weights determined by Ensemble Selection for each bag are added up to create the final Ensemble.

Ensemble Selection can be run using a set-aside validation set, or using cross-validation. The cross-validation implemented in the EnsembleSelection classifier is a method proposed by Caruana et al. called “embedded” cross-validation. Embedded cross-validation is slightly different than typical cross-validation. In embedded cross validation, as with standard cross validation, we divide up the training data in to  $n$  folds,

with separate training data and validation data for each fold. We then train each base classifier configuration  $n$  times, once for each set of training data. We are then left with  $n$  versions of each base classifier, and we combine them in EnsembleSelection in to what we call a “model” (represented by the EnsembleSelectionLibraryModel class). Thus, a model in EnsembleSelection is actually made up of  $n$  classifiers, one for each fold.

The concept of embedded cross-validation is to choose *models* for the ensemble. That is, rather than being interested in the performance of a single trained *classifier*, we are concerned with how well the *model*, or “base-classifier configuration” performed (based on the performance of its constituent classifiers). Notice that for every instance in the training set, there is one classifier for each model/configuration which was not trained on that instance. Thus, to evaluate the performance of the model on that instance, we can simply use the single classifier from that model which was not trained on the instance. In this way, we can evaluate each model using the *entire* training set, since for every instance in the training set and every model, we have a prediction which is not based on a classifier that was trained with that instance. Since we can evaluate all the models in our library on the entire training set, we can perform ensemble selection using the entire training set for hill-climbing.

## User Guide

As with all WEKA classifiers there are two ways to run Ensemble Selection: either from the command line or from the GUI. However, this classifier is slightly unique in a couple of ways. First, it requires the additional first step of creating a model list file. This file basically tells the algorithm which models you would like to train for your model library. Second, the EnsembleSelection classifier requires you to specify a “working directory” on a writable file system where it will be able to save models that it has trained for use in building ensembles.

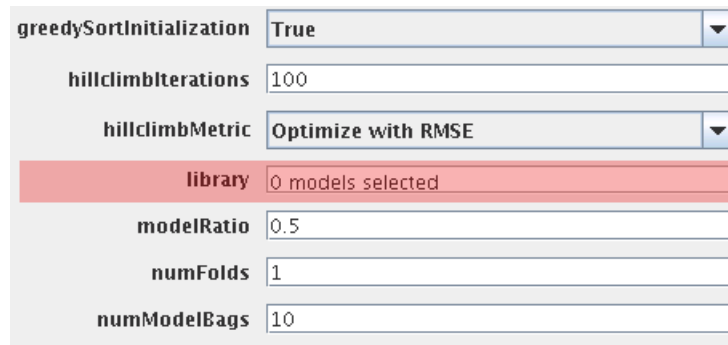
### Building the model list

Whether you are going to run the Ensemble Selection algorithm from the command line or from the GUI (e.g. Explorer), you are going to first need to create a model list. To do this, you will need to use the Library Editor GUI that was made for this purpose, build a model list and then save it to a .model.xml file.

We created this user interface to let us quickly build lists, aka libraries, of classifiers to be trained. The design goal of this widget was to let us quickly throw together really long lists of models that we will train for Ensemble Selection libraries. We want to be able to take shortcuts like “build me every neural net with every possible value in this range of learning rate values, and this range of momentum values, and both with and without the

nominalToBinary filter turned on” and then BAM – you have a list of classifiers consisting of every possible combinatoric combination of the parameter ranges specified.

You can bring up the Ensemble Library Editor by simply clicking on the EnsembleSelection “library” attribute in the Explorer (highlighted below).



The screenshot shows a GUI for the Ensemble Library Editor. It contains several parameters with input fields or dropdown menus:

- greedySortInitialization**: A dropdown menu set to **True**.
- hillclimbIterations**: A text input field containing **100**.
- hillclimbMetric**: A dropdown menu set to **Optimize with RMSE**.
- library**: A red-highlighted text input field containing **0 models selected**.
- modelRatio**: A text input field containing **0.5**.
- numFolds**: A text input field containing **1**.
- numModelBags**: A text input field containing **10**.

Clicking the attribute will bring up the Model List editor which is a separate GUI Window that consists of 4 tabbed panels. While the first panel displays all the models in the current model list, the other three allow you to add models to this list.



**Model List Panel** – This is the first panel that is responsible for displaying all models that are currently in the model library. It also lets you save/load model lists in either xml format (.model.xml) or as a simple flat file (.mlf – recommended for logging purposes only!).

**Add Models Panel** – lets you generate sets of models to add by specifying Classifiers and ranges of parameters for them.

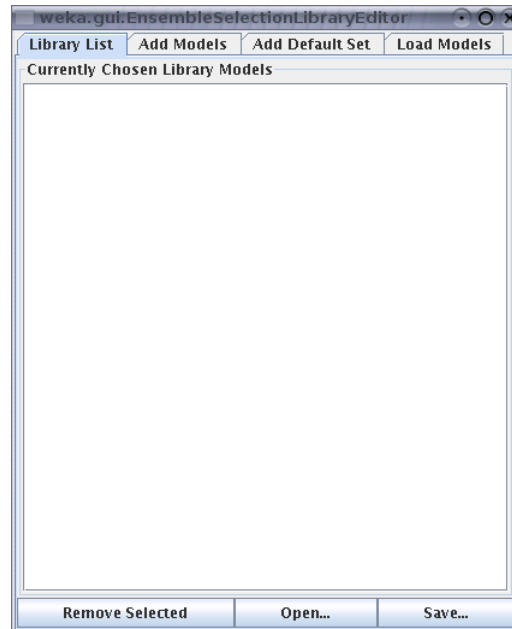
**Add Default Set Panel**– lets you add models from one of the Ensemble Selection default lists

**Load Models** – Detects all of the .elm files in the working directory currently specified for your Ensemble Selection classifier and builds a model containing all the models found.

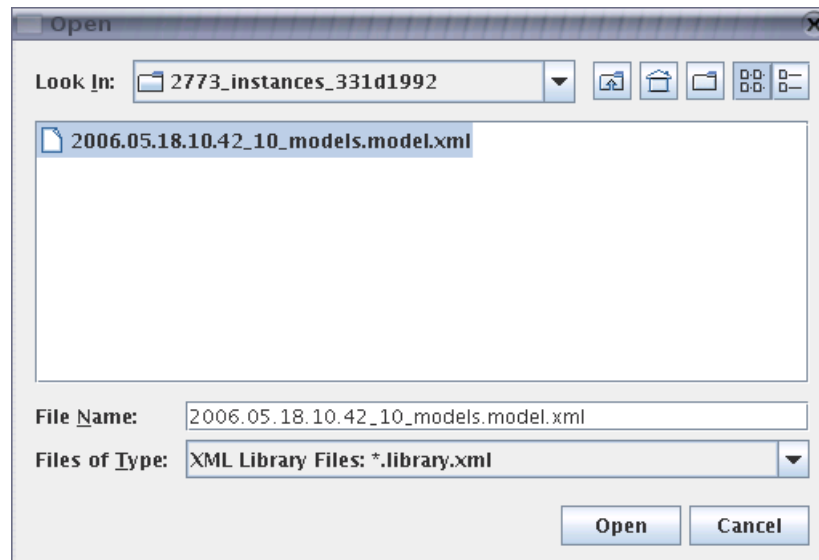
We tried to make these user interfaces as intuitive and simple as we could, so we think you could probably just sort of jump in and figure them out by playing around. If you get stuck, the following four sections describe these panels in more detail.

## The Model List Panel

The first panel you will see is the “Library List” Panel (shown below). As the name suggests, this tab simply shows you all the models that have currently been chosen to go into your library list. Since you shouldn't have any models in your list at the beginning, the list should be empty when it first comes up.



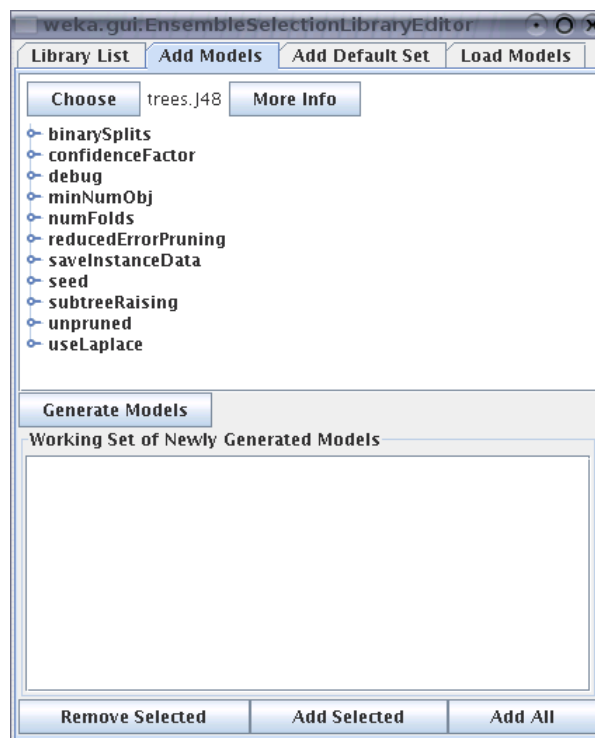
From this panel you can also save/load model list files with the respective buttons at the bottom which bring up standard file choosers.



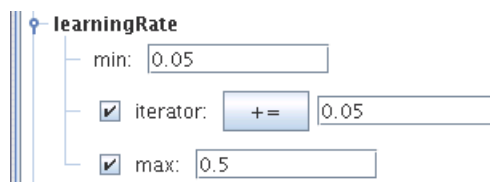
While loading, the file chooser will only let you choose files with either the .model.xml extension or the .mlf extension depending on which “files of type” is currently selected.

## The Add Models Panel

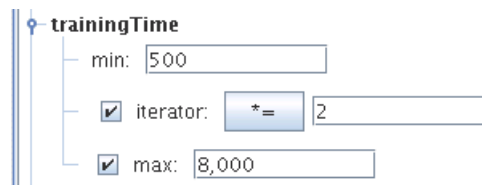
OK, so lets add some models to our now empty library. First hit the “add models” tab to make a small working set. There are two parts of the add models panel. The top half shows you all of the parameters for the current selected Classifier type. This is where you specify value ranges and combinations. Try playing around with collapsing and expanding the tree nodes. Also note that we created tool tips for all the parameters so if you hold your mouse still over one for a few seconds, a comment describing it will pop up. The bottom half shown you a current temporary working set from which you can add models to the main “Library List” panel (above).



For numeric attributes, you can specify ranges. For example, with the neural net classifier, the following says to try all learning rates from 0.05 to 0.5 in increments of 0.05, which will give us 0.05, 0.10, 0.15, etc...



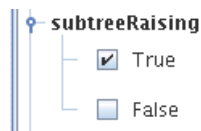
Note that you can also define exponential ranges – that is, multiply by the iterator instead of divide by the iterator to get each value in the range. You can toggle this mode by hitting the “+=” button. When you hit it will show “\*=” instead. As an example, for neural nets its often useful to plot exponentially increasing ranges of training epochs. The following would give the learning rate values of 500, 1000, 2000, 4000, and 8000.



The screenshot shows a configuration panel for the **trainingTime** parameter. It includes a 'min' field set to 500, an 'iterator' field set to 2 with a dropdown menu currently showing '\*=' (which was previously '+='), and a 'max' field set to 8,000. All three fields have checkboxes to their left, all of which are checked.

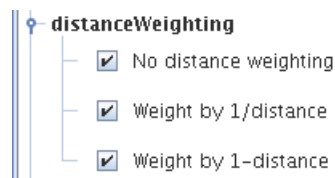
For nominal values (either an enumeration of values or binary true/false booleans) You simply check the boxes of the values you wish to try.

Example of Binary Attribute (subtreeRaising in J48):



The screenshot shows a configuration panel for the **subtreeRaising** parameter. It has two options: 'True' and 'False', each with a checkbox to its left. The 'True' checkbox is checked, while the 'False' checkbox is unchecked.

Example of an Enumeration Attribute (distanceWeighting in IBk):



The screenshot shows a configuration panel for the **distanceWeighting** parameter. It has three options: 'No distance weighting', 'Weight by 1/distance', and 'Weight by 1-distance'. Each option has a checkbox to its left, and all three checkboxes are checked.

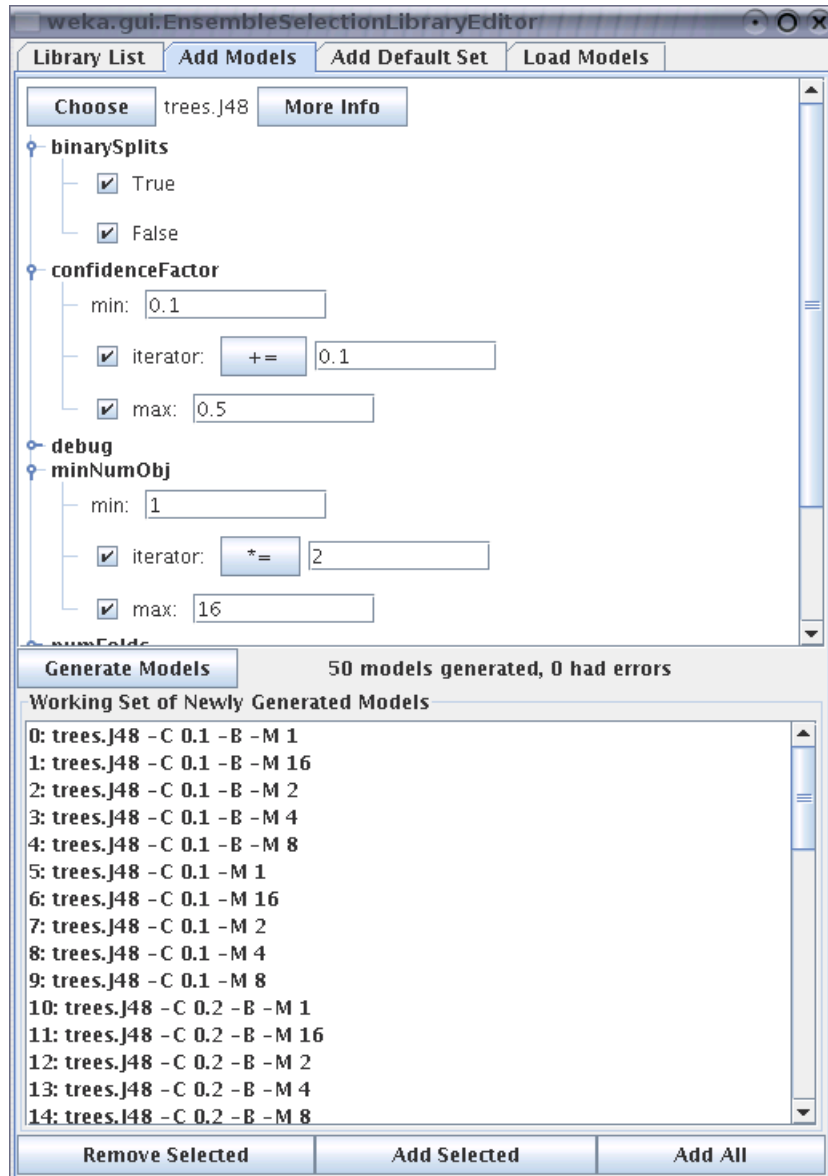
Now lets see what happens when we generate models from our specified parameter ranges. Consider the example below using the J48 classifier. We've specified 2 values for binarySplits (true and false), 5 values for confidence factor (0.1, 0.2, 0.3, 0.4, and 0.5), and 5 values for minNumObj (1, 2, 4, 8, and 16). So we should get a total of 50 models generated (2 x 5 x 5).

When you have selected the appropriate parameter space up top, you hit the “generate models” button in the middle to generate and add all of the models into the temporary working list on the bottom. This is a scrollable list in which each row represents a model (see below). To get more information about a model leave your mouse pointer over it to see a tool tip showing all the respective parameter values. Furthermore, “list like”



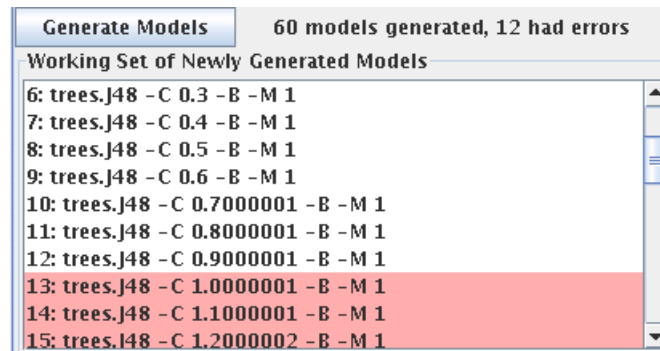
keyboard shortcuts also work: <ctrl>+a will select all models in the list and <delete> will delete the models currently selected.

You can prune this model list to your liking by selecting models from the list that you don't want and then hitting the "remove selected". Once you are satisfied with the working list and are ready to add them to the model library, you just hit the "Add all" button and every model in the list will be added to the main panel. Also, you can prune this list by selecting a set of one or more models (<ctrl>+click or <shift>+click) and then hitting the remove selected button.

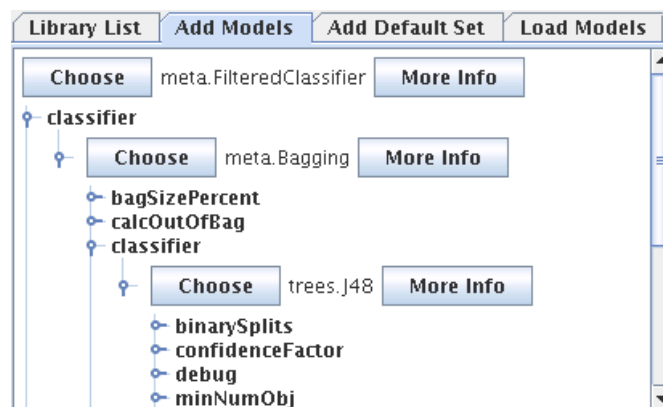


It should be mentioned that it is possible to use the GUI to specify invalid values for classifier parameters – in this case these models will appear highlighted. You can use the mouse tooltip to find out what the problem is (specifically what the exception text was

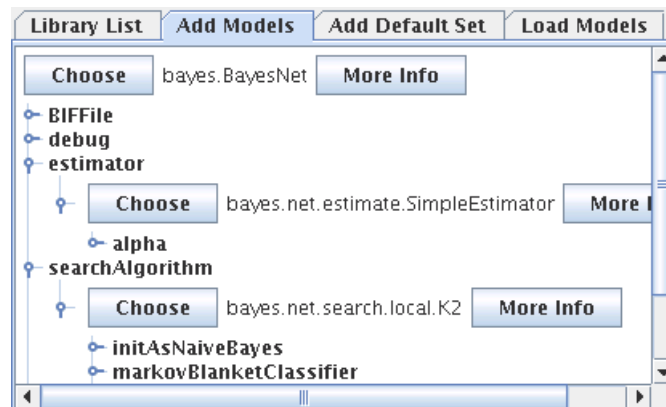
when trying to set the options for that particular model). For example, trying to specify a value greater than 1 for the “confidence value” parameter to J48 trees causes an exception. Therefore, these models will be highlighted and holding the mouse pointer over them will display the tooltip explaining why. If you want to remove all models that had errors, just hit the “remove invalid” button. Also note that these won’t get added to the main panel – they’ll just be ignored. Oh, and one other strange thing we’ve encountered is that some classifiers will not throw an exception when given invalid input but instead will just replace it with a default value. So in these cases the invalid models won’t be highlighted - they simply won’t appear in the list and a model with a valid value will instead.



Finally, it should be mentioned that the tree list is recursively defined. That is, a classifier with a classifier as a parameter (i.e. some of the meta classifiers) will result in that parameter having its own subtree of parameters just like the root node. Try it. Select the meta.Bagging classifier and experiment. You can even create bags of bags of bags. Although that is a really dumb idea its nice to know the original Weka flexibility remains. This allows powerful combinations. For example, consider the following: We are wrapping a J48 classifier inside of two layers of meta classifiers. We can actually specify parameter ranges on all three levels and it will generate all the possible combinations as expected.



This subtree behavior applies to any objects that are rendered by the GenericObjectEditor – not just classifiers. As an example, consider the BayesNet Classifier which has two parameters, the estimator and search algorithm, which are custom objects specific to that algorithm which have their own sets of parameters. So as another example, consider the two arguments estimator and searchAlgorithm to a BayesNet. These are not classifiers but since they use the GenericObjectEditor as a GUI, the LibraryEditor knows how to grab their relevant information and display their sub-arguments within the classifier tree.



## Add Default Set Panel

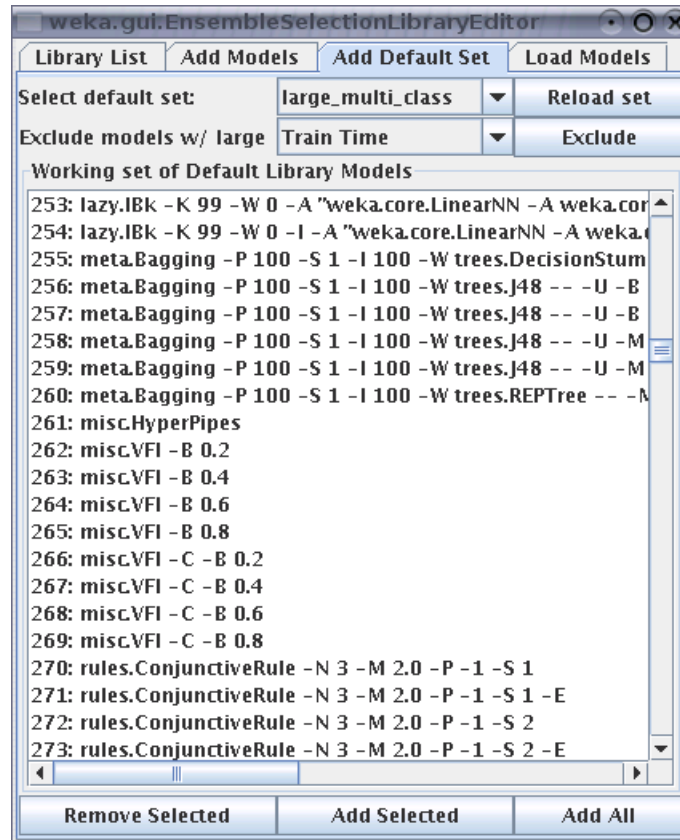
This panel is probably the right place to start if you just want to try out ensemble selection with too much effort. It lets you choose models from an already existing model list. All you have to do is select the list you want from the drop down list at the top. We currently have two notable default lists: one targeting multi class and one targeting binary class datasets. Note that it is normal for the GUI to freeze up for a little bit after selecting one of these lists because they are quite large (>1000 models), and the GUI needs to process and test each one to make sure that all the options are valid.

For a list to be selectable within this GUI it needs to

- 1) be listed in the `weka/classifiers/meta/ensembleSelection/DefaultModels.props` file
- 2) actually be located in the `weka/classifiers/meta/ensembleSelection/` directory in the classpath

You will also see near the top a drop down list labeled “Exclude models with large: ” with the selectable options of “train time”, “test time”, and “file size”. This lets you prune the default list by removing models that will take too much resources to train. Once you select the resource you care about, you can hit the exclude button and the list should shrink. The logic that determines which models get removed for each type of resource (train time, test time, and file size) is based on a list of regular expressions that can be found in the `weka/classifiers/meta/ensembleSelection/DefaultModels.props` file.

Currently, these lists of regular expressions are fairly basic e.g. IBk classifiers will get removed for test time, MultilayerPerceptron will get removed for train time, etc...



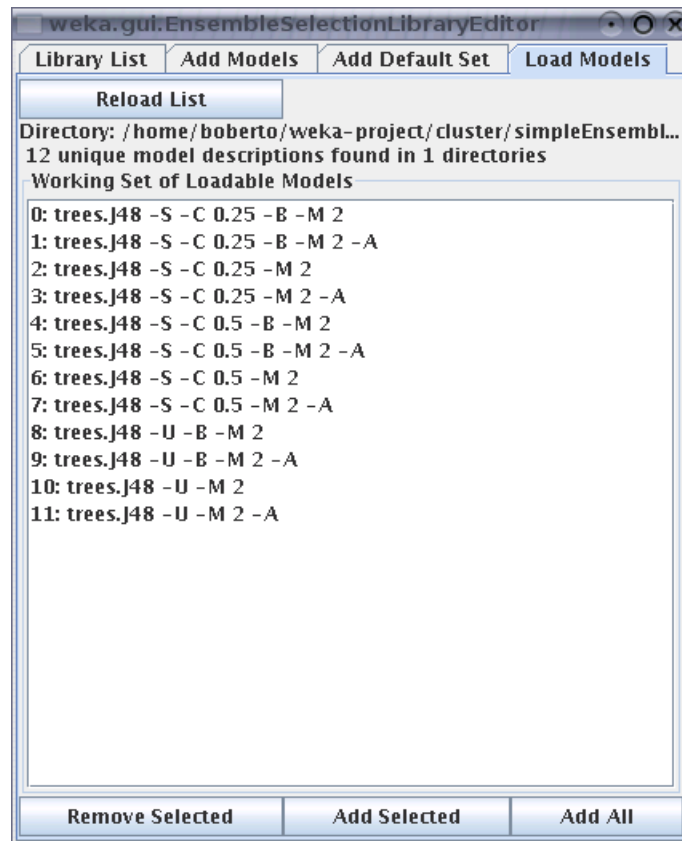
Similar to the Add Models Panel you simply hit either “add all” or “add selected” to move these models to the model library shown in the “list models” panel.

## Load Models Panel

This GUI will automatically search the file system for .elm files in the currently selected working directory and load them into a list. This is really just meant for convenience assuming that you've already trained a library and would now like to build ensembles with it.

It is important to note that the default behavior is to search ALL directories in your current working directory. So if you've trained two different libraries with different classifiers in your working directory the list that will appear in the load models panel will be the union of both these model sets.

The “reload” set button will simply reload the list in case 1) you've changed your working directory parameter or 2) the set of models in your directory has somehow changed.



## Choosing a Working Directory

The working directory is just a String argument to Ensemble Selection representing the file path to your ensemble. While that seems pretty straightforward, there's a lot of details about the “anatomy” of the working directory that aren't. This section will explain the default behaviors surrounding this parameter, the file structure of a working directory, and the naming conventions used.

As previously mentioned, in a typical run of Ensemble Selection you are going to want to train a lot of models. So we assumed from the beginning that there was no way all of them would fit in to system memory. To solve this problem, we assume that the user will specify a “working directory” for the library models to be saved in. This gives our classifier a place where it can serialize all of its library models to the file system along with other information it needs to track which models were trained what sets of data. Alternately, if you have already trained a library of models, then by specifying a working directory you are telling EnsembleSelection where to look for the models it needs.

The following subsections explain the internals of the working directory.

## **Establishing the working directory**

The first thing that the classifier does is establish the working directory by finding the directory specified as the working directory.

If you specify a working directory that does not exist on the file system, it will be created (assuming that its a well form file path, otherwise you get an exception).

If you do not specify a working directory, then EnsembleSelection will create a default working directory in your home directory named “Ensemble-X”, where X is the first unused number between 1 and 999 that doesn't exist in a in your home directory. If you already have directories named Ensemble-1 all the way to Ensemble-999 in your home directory then you will get an exception, not to mention a lot of working directories!

## **Dataset Directories and Naming Conventions**

The next thing that the classifier does at train time is to establish a directory in your working directory that is specific to the set of instances it was given to train. For each set of instances that you train in a working directory – a directory with a name unique to that set of instances will be created.

To get the unique directory name for the train set, the checksum of the Instances.toString() method is used to get a sequence of 8 alphanumeric characters. Note that the Instances.toString() method returns the dataset in .arff format. The checksum is then appended to the number of instances in the dataset.

For example, a typical directory name will be something like:  
2391\_instances\_46778c6d

If it doesn't already exist, then the dataset directory gets created and the classifier begins training all the classifiers that were specified in the model list argument. All of these models are stored in the directory associated with the training dataset.

If the dataset directory already exists, then each model in the model list is created only if it doesn't already exist in the dataset directory. Otherwise, each model is created and stored in the directory.

At first, this checksum naming convention might seem strange, but it fulfills a very important property. It makes sure that the models used to create an ensemble were trained on the vary same data that was given to train its underlying library models. Since we are allowing model libraries to be created separately from ensembles, we decided that we needed some mechanism to enforce this. This is an important property!

## Logging Files

It is also worth mentioning that at train time, the output of the `Instances.toString()` is automatically saved in a file with the same name as the dataset directory but with the `.arff` extension indicating it is the dataset file.

For example, in the `2391_instances_46778c6d` directory, we would find:  
`2391_instances_46778c6d.arff`

That would store the specific set of instances used to train all models in that directory. We thought this would be useful for logging purposes.

Also, the set of models that you attempted to train for the library will also be saved in this directory in both the xml (`.model.xml`) and human readable flat file (`.mlf`) formats. These lists will be saved with a file name reflecting the time training started along with the number of models that were going to be trained.

For example, you would find model list files with names similar to this:  
`2006.05.18.16.34_1041_models.mlf`  
`2006.05.18.16.34_1041_models.model.xml`

## Ensemble Library Model files (.elm extension)

The next thing to explain is the files used to store our ensemble library models. What we do is take the command line that would be used to normally train the library model and turn it into a file name with four transformations. First, we turn all space and quote (") characters into underscore(\_) characters to make things more readable. Second, for filenames that would be greater than 128 characters, we trim the end off so they will satisfy the 128 character limit of most operating systems. Third, we append a checksum of the original model command line to guarantee that the file names are unique for each model. Fourth, we add the `.elm` file extension indicating the file is an Ensemble Library Model file.

To summarize, the model represented by this command line string:

```
weka.classifiers.trees.J48 -C 0.25 -B -M
```

will be saved in a file with the name:

```
weka.classifiers.trees.J48_-C_0.25_-B_-M_23bae0c94.elm
```

Note that the .elm files hold all models (one for each data fold) associated with a particular classifier and associated set of parameters along with other information that Ensemble Selection will need later on to build ensembles.

## **Lock Files**

We created what we call “informal” lock files that are created before each library model is trained and deleted after the model is finished training. Since these lock files correspond to a specific model indicating that it is currently being worked on, they have the same naming convention except they carry the .LCK file extension.

The .elm file in the previous example will have the lock file name:

```
weka.classifiers.trees.J48_-C_0.25_-B_-M_23bae0c94.LCK
```

These are “informal” lock files we use for processes to say “hey, I’m working on that model right now”. When training a list of models, EnsembleSelection will simply skip any models in its list for which it detected a lock file. We do this for two reasons:

1 - This is nice when training libraries in parallel on clusters. Each node knows not to train a model that has a .LCK file. These get deleted when the process is done training the respective model.

2 - These files are also useful when training models on a single computer because they also help you figure out which (if any) models didn't train. When EnsembleSelection detects these .LCK it assumes that it shouldn't deal with the classifier and skips to the next one in the list.

You may be wondering why we don't use real lock files (which java does support). Our main reason is that we felt this would be overkill. The fact is that when training models in parallel, we don't really need to enforce any guarantees that the .LCK files truly reflect the current state of training. The worst case (which should be extremely rare) is that two nodes would train the same model which isn't really all that bad. Furthermore, implementing true file locks would have taken more time.

## **Ensemble Selection from the command line**

We recommend training the library models as a separate step because it will make things easier for debugging errors should something go wrong.

### **Step 1: Creating a model list file**



Even though you are running Ensemble Selection from the command line, you still need to create the model list file (.model.xml) file from the GUI. Follow the instructions in the previous section about building model list files to do this. Once you've saved the model list file, you will be specifying this file on the command line with the “-L path/to/your/model/list/file.model.xml” option described below.

## **Step 2: Training the library**

To build a library from the command line you need to use the “-A library” option to tell Ensemble Selection that it shouldn't build any ensemble models and that you only want to train the base classifiers.

**WARNING!** You should basically NEVER specify cross validation (CV) from the command line with the -x option that is read by Evaluation.java. Ensemble Selection fully supports cross validation but it is implemented very carefully to use validation data to build ensembles. Instead, you should use the -X option to specify the number of folds.

Also, this may seem counter intuitive, but the “-no-cv -v” options are extremely important since they prevent the Evaluation class from defaulting to 10 fold CV to try to create a performance estimate. This is a lot of wasted CPU time since we are only training a library and don't care about performance estimates. With Ensemble Selection, you should always tell the classifier evaluation code not to perform cross validation regardless of whether you are doing it for Ensemble Selection.

Finally, the “-V <validation ratio>” option is only meaningful when you do not specify a number of cross validation folds greater than 1. If you are using CV, then validation sets are generated automatically, otherwise if you have only one fold then the percentage specified with the -V option will be held aside for validation.

The following is an example command line to build an ensemble library:

```
java weka.classifiers.meta.EnsembleSelection -no-cv -v -L
path/to/your/model/list/file.model.xml -W /
path/to/your/working/directory -A library -X 5 -S 1 -O -D -t
yourTrainingInstances.arff
```

## **Step 3: Building the Ensemble**

Just like any WEKA classifier, you specify the file to save your model to with the -d flag and the train instances with the -t flag. Note that these train instances should be the same ones used in the previous step to train the library. In addition, the following options given reflect most of the default values:

-10 model bags with a model ratio of 50% for each.

- using RMSE as the hill climbing metric while adding models
- using 100 iterations of the hill climbing algorithm to add models.
- using greedy sort initialization with 100% of the models.
- five fold cross validation is being used.

The following is an example command line to build and save an Ensemble Selection model from an ensemble library:

```
java weka.classifiers.meta.EnsembleSelection -no-cv -v -L
path/to/your/model/list/file.model.xml -W /
path/to/your/working/directory -B 10 -P rmse -A forward -E 0.5 -H
100 -I 1.0 -X 5 -S 1 -G -O -R -D -d /path/to/your/model.model -t
yourTrainingInstances.arff
```

### Step 4: Testing the Ensemble

Just like any WEKA classifier, you specify the file to load your model from with the -l flag and the test instances with the -T flag. There's really nothing special to note here.

The following is an example command line to test an ensemble library:

```
java weka.classifiers.meta.EnsembleSelection -no-cv -W /
path/to/your/working/directory -O -D -l /path/to/your/model.model
-T yourTestingInstances.arff -i -k
```

## Ensemble Selection from the GUI

### A Note on Memory Usage from the Explorer

Running Ensemble Selection from the explorer is only recommended for either building model lists or playing around with small datasets. You will find that with a decent sized model list on anything but the smallest datasets you will quickly run out of memory.

The main problem is the fact that the Evaluation code in WEKA asks classifiers to make predictions one at a time instead of passing them all at once as a collection. We are not criticizing this design choice. It's just that in our case it means that we had to implement a small work around to make things acceptably efficient.

For ensemble Selection to work we need to average predictions for all our ensemble models together. The original implementation did something like this:

```
For each model
    Deserialize the model from the file system
    For each Instance
```

```
        Get a prediction for the given instances
    Garbage collect the model
```

This wasn't too expensive. We could just load models from the file system one at a time and get all their test predictions, and then throw the model away. There's no need to keep all the models in memory.

However, since `Evaluation.java` asks our classifier to make predictions on test instances one at a time, we would be stuck doing something more like this:

```
For each Instance
    For each model
        Deserialize the model from the file system
        Get a prediction for the given instances
        Garbage collect the model
```

This means that if you have 1000 test data points, you are going to have to deserialize every ensemble model 1000 times! We felt this was unacceptable and decided to force the classifier to keep all ensemble models in memory to prevent all the deserialization.

Furthermore, we implemented prediction caching in our main method where test predictions for all ensemble models are cached before handing control over to `Evaluation.java`. Unfortunately, this work around only works when invoking `EnsembleSelection` from the command line. This is because in the Explorer, unlike our main method there is no time when control is handed to our classifier with all the test instances.

So to summarize, what all this means is that when training Ensemble Selection for reasonably sized model lists on reasonably sized datasets, you should do it from the command line (as described in the previous section) or you might get an out of memory exception.

Otherwise, we think that the GUI is fine for sort of playing around to get a feel for how ensemble selection works with small model lists (such as the “`toylist.model.xml`” in the defaults panel).

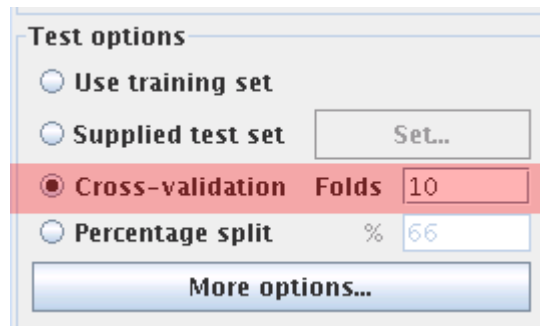
### **Step 1: Specify the models to use**

This is slightly different from the command line in that you don't specify the file. Simply click on the library attribute in the Ensemble Selection property panel to bring up the library editor. As mentioned in the previous section on the library Editor GUI, you can either load an already existing list or just add some models to your library with one of the panels.

### **Step 2: Training the library**

After typing in the name of your working directory and specifying the models for your library, you just need to tell the classifier it is only training models. In the drop down menu for algorithm on the Ensemble Selection Panel select “library” to indicate that you

**WARNING!** As described in the command line section, you should basically NEVER specify cross validation (CV) from the Explorer Classifier testing GUI (Shown below). Instead choose one of the other options as CV is prohibitively expensive from Ensemble Selection just to get performance estimates.



### Step 3: Building and Testing the Ensemble

These steps can be done just like they can with any other WEKA classifier. Just keep in mind our warning about the out of memory errors and don't be afraid to try things from the command line.

## Overview of Classes used by EnsembleSelection

The implementation of EnsembleSelection and its library editor was a significant undertaking. The following is a brief overview of the packages created to support the classifier with a list of the class files stored in each. For more information, please see the associated javadocs and code comments.

Package: `weka.classifiers.meta`

The actual classifier

`weka.classifiers.meta.EnsembleSelection.java`

Package: `weka.classifiers`

These are base classes we created so that others could use the basic “Ensemble Library” functionality of the LibraryEditor GUI.

`weka.classifiers.EnsembleLibraryModelComparator.java`

```
weka.classifiers.EnsembleLibraryModel.java
weka.classifiers.EnsembleLibrary.java
```

Package: weka.classifiers.meta.ensembleSelection

These classes support the main EnsembleSelection algorithm. There is also a properties file and three model lists that are used to populate the default models panel in the library editor GUI.

```
weka.classifiers.meta.ensembleSelection.EnsembleModelMismatchException.java
weka.classifiers.meta.ensembleSelection.ModelBag.java
weka.classifiers.meta.ensembleSelection.EnsembleMetricHelper.java
weka.classifiers.meta.ensembleSelection.EnsembleSelectionLibrary.java
weka.classifiers.meta.ensembleSelection.EnsembleSelectionLibraryModel.java
weka.classifiers.meta.ensembleSelection.DefaultModels.props
weka.classifiers.meta.ensembleSelection.large_binary_class.model.xml
weka.classifiers.meta.ensembleSelection.large_multi_class.model.xml
weka.classifiers.meta.ensembleSelection.toylist.model.xml
```

Package: weka.gui

EnsembleLibraryEditor is a base class we created so that others could use the basic LibraryEditor GUI. We extend this class with EnsembleLibraryEditor to do more specific EnsembleSelection things.

```
weka.gui.EnsembleLibraryEditor.java
weka.gui.EnsembleSelectionLibraryEditor.java
```

Package: weka.gui.ensembleLibraryEditor

These are classes implementing and supporting the panels in the Library Editor GUI.

```
weka.gui.ensembleLibraryEditor.ModelList.java
weka.gui.ensembleLibraryEditor.AddModelsPanel.java
weka.gui.ensembleLibraryEditor.ListModelsPanel.java
weka.gui.ensembleLibraryEditor.DefaultModelsPanel.java
weka.gui.ensembleLibraryEditor.LoadModelsPanel.java
weka.gui.ensembleLibraryEditor.LibrarySerialization.java
```

Package: weka.gui.ensembleLibraryEditor.tree

This entire package supports just the add models panel. Getting the neat Jtree user interface to work for building lists of classifiers was by far one of the greatest challenges we faced. Most of these classes implement the functionality needed for a single node in the tree.

```
weka.gui.ensembleLibraryEditor.tree.GenericObjectNode.java
weka.gui.ensembleLibraryEditor.tree.CheckBoxNodeEditor.java
weka.gui.ensembleLibraryEditor.tree.ModelTreeNodeRenderer.java
weka.gui.ensembleLibraryEditor.tree.ModelTreeNodeEditor.java
weka.gui.ensembleLibraryEditor.tree.DefaultNode.java
weka.gui.ensembleLibraryEditor.tree.CheckBoxNode.java
```

```
weka.gui.ensembleLibraryEditor.tree.NumberClassNotFoundException.java  
weka.gui.ensembleLibraryEditor.tree.GenericObjectNodeEditor.java  
weka.gui.ensembleLibraryEditor.tree.InvalidInputException.java  
weka.gui.ensembleLibraryEditor.tree.NumberNodeEditor.java  
weka.gui.ensembleLibraryEditor.tree.NumberNode.java  
weka.gui.ensembleLibraryEditor.tree.PropertyNode.java
```

## User FAQ

*When is Ensemble Selection a good idea?*

The interface allows for easy creation of a large set of classifiers with minimal (human) effort usually providing state-of-the-art performance. Other competing methods such as Bayesian Model Averaging and Stacking are known to over-fit with large libraries of models. This has the capability of not just training many models, but evaluating their performance on a test set (using EnsembleSelection.main() and the -V option).

*When is ensemble selection a bad idea?*

It's *very* time consuming, and takes a lot of memory. To train an ensemble library for a reasonably sized dataset it will take days to weeks of compute time. Also, at the end of training your ensemble it is difficult or perhaps even impossible to intuitively understand its mechanism – by that we mean the underlying logic your model uses to make predictions. Whereas with something like a single tree classifier you can look at the branches and validate/understand why it was built the way it was. With ensemble selection, you have predictions averaged across hundreds of models which makes this process difficult if not impossible.

*Expensive eh? So is it possible to parallelize this?*

If you have a cluster, we have successfully made the library training easily parallelizable, (assuming your cluster nodes have a shared file system, e.g. NFS). All you have to do is invoke the same command line argument to train your ensemble library to all nodes you wish to use to train your model library. Make sure that you specify the same path for your working directory (which again should be remotely reachable by all of them) and

make sure to use the “-A library” option to tell them all to only train the library and not to do anything else. Note that the step of using the library to then build an ensemble with the Ensemble Selection algorithm is not currently parallelizable.

*What if I just want to train a bunch of models and maybe find the best one without having to deal with all this fancy Ensemble Selection model stuff?*

We can do that too. Just use the “-A best” option. The ensemble will be made up of the single model which performed best on the validation data. Furthermore, if you want to get the performance of all the models with respect to the validation data, you can do this as well using the “-V” (verboseOutput) option.

*I got an out-of-memory error when using EnsembleSelection. What can I do?*

Two things: 1) make sure you use the java -xm.... Option to increase the amount of memory available to the JVM and 2) If you got the error from the GUI, try it again from the command-line. For an explanation of the Memory Usage problems in the explorer please see the User Guide section about this.

*Why do I need a working directory? Can't I keep them in memory?*

No, that takes too much memory. Ensemble Selection is designed for very large libraries of models (e.g. > 1,000), and so in most real-world situations, the model library could not possibly be held in memory.

*I don't know which models to use. Are there default lists of models I can use?*

Yes. This was high on our list when we first got started building this classifier, we wanted people to be able to choose from reasonable default model lists. Just fire up the LibraryEditor and click on the “default models” tab to see a few default lists.

*I want to try different models than in the default list. Can I do that?*

Yes, go check out the Add Models Panel section in the user guide.

*I noticed two types of model list files .mlf and .model.xml - what's the deal?*

Originally, we thought it would be easiest to make the model list files be a simple flat file with a different Classifier + set of options on each line. It turns out this has many problems (getOptions and setOptions do not necessarily interact properly or even fully represent the associated classifiers). These are the model list files with the “.mlf” extension. Later we adopted an xml file schema that works great but doesn't have the nice simplicity/human readableness of the old .mlf files. These xml model files have the .

model.xml extension. Anyway, we've had great success with the new format and we recommend you use this extension instead of the .mlf's.

*Can I use the same workspace directory to train models for two different problems? Two different partitions of the data for the same problem? Two different model lists?*

Yes, yes, yes. See the section describing the dataset/checksum naming convention.

*I got an EnsembleModelMismatchException, what does that mean?*

This is a tough one. Basically, we track all of the dataset information that was used to create each model. This is because we want to protect users from doing foreseeably bad things. e.g., trying to build an ensemble for a dataset with models that were trained on the wrong partitioning of the dataset. This could lead to artificially high performance due to the fact that instances used for the test set to gauge performance could have accidentally been used to train the base classifiers. So in a nutshell, we are preventing people from unintentionally "cheating" by enforcing that the seed, #folds, validation ration, and the checksum of the Instances.toString() method ALL match exactly. If you try to build an ensemble and one of its models was not trained with all of these same parameter values, then we throw that specific exception.

*Why did the load models tab in the LibraryEditor show me all the models I trained for a bunch of different datasets – that were in different directories?*

Yes, it can be confusing, but the load models tab is populated by *all* models in subdirectories of the working directory. (This is out of necessity, because at the time we populate it, we don't know what dataset you're using).

*Is it possible to modify mylist.model.xml by hand or with a script?*

Theoretically yes, in practice, no – do so at your own risk. We highly recommend using the list editor GUI instead. The .mlf files may be edited by hand, but as noted elsewhere, not all classifiers can be properly configured using command-line options.

*How do I specify the models for EnsembleSelection to use from the command-line?*

Use the -L option, and provide a ".model.xml" file. You can create a ".model.xml" file using the Library Editor (see the user's guide for more detail). It is also possible to use a file in ".mlf" format, which is simply a list of classifiers and options for them.

*So I made a bunch of models. How do I know specifically what data they were trained on?*



Every time you train a library, we write out a .arff file in the respective instances directory containing all of the instances used to train that set of models. It's named based on the data and number of instances. Convenient, no?

*So I made a bunch of models. How do I know what models were in the library at train time?*

For logging purposes, we also write out both a .model.xml and .mlf file at train time of all the models you are attempting to train.

*What happens if not all of my models train... i.e., throw an exception for whatever reason?*

This is actually not a matter of if, it's a matter of when. Not all models can use all data sets. lately though, we trap most exceptions and throw models that didn't train out of the ensemble. Most of the time this works and we're okay. However, if we ran out of memory, this could still kill the process no matter how much errors we trap. Sometimes, you just have to figure out which model(s) are gumming up the works and manually remove them from the model list. Although in our experience this isn't too often.

*Hey, I have a bunch of existing Weka models, and I'd like to make an Ensemble out of them. Can I do that?*

Sorry, currently not supported. Would be a great future addition though.

*Hey, I just ran EnsembleSelection to find the "Best Model", and now I want to save that model for later use.*

Sorry, can't do that either. It's kind of a hack, but what you can do is build an Ensemble of only the best model (-A best). This will give you an EnsembleSelection classifier to classify future instances. Ultimately though, we think that there should be a mechanism to extract the actual models from our .elm files.

*Why is there a separate list for binary vs multi class problems in the default list panel in the LibraryEditor? I thought all Weka classifiers worked for either?*

There are some models that don't scale well for multi class problems. Models such as SVM's (functions.SMO), have train + test times that increase exponentially with the number of classes for a problem. We've observed classifiers which took 15 minutes to train on a binary problem took over 8 hours to train on a multi class problem, while other classifiers in the library did not have the same increase. So we felt the best approach would be to maintain two separate default model lists. One for multi class and one for binary.

*Is it okay for me to combine training the library of models and the EnsembleSelection in one step?*

Yes, that's fine – but doesn't allow for parallelization. Also – separating the steps can make things easier for debugging.

*What's with the crazy letters and numbers in my .elm files and the subdirectories of my working directory?*

See the section on file naming and checksums in the “Working Directory”.

*Can I use a trained a EnsembleSelection classifier for new data after the ensemble is built?*

Yes, just use the .model file like you would any saved WEKA classifier. However, it must have access to the models it selected, which are saved in separate .elm files. The directory where they can be found can be specified using the `-W` option.

*How does cross validation work in EnsembleSelection?*

That's a doozy...

(note: this is just copied and pasted from above) Ensemble Selection can be run using a set-aside validation set, or using cross-validation. The cross-validation implemented in the EnsembleSelection classifier is a method proposed by Caruana et al. called “embedded” cross-validation. Embedded cross-validation is slightly different than typical cross-validation. In embedded cross validation, as with standard cross validation, we divide up the training data in to  $n$  folds, with separate training data and validation data for each fold. We then train each base classifier configuration  $n$  times, once for each set of training data. We are then left with  $n$  versions of each base classifier, and we combine them in EnsembleSelection in to what we call a “model” (represented by the EnsembleSelectionLibraryModel class). Thus, a model in EnsembleSelection is actually made up of  $n$  classifiers, one for each fold.

The concept of embedded cross-validation is to choose *models* for the ensemble. That is, rather than being interested in the performance of a single trained *classifier*, we are concerned with how well the *model*, or “base-classifier configuration” performed (based on the performance of its constituent classifiers). Notice that for every instance in the training set, there is one classifier for each model/configuration which was not trained on that instance. Thus, to evaluate the performance of the model on that instance, we

can simply use the single classifier from that model which was not trained on the instance. In this way, we can evaluate each model using the *entire* training set, since for every instance in the training set and every model, we have a prediction which is not based on a classifier that was trained with that instance. Since we can evaluate all the models in our library on the entire training set, we can perform ensemble selection using the entire training set for hill-climbing.

*I noticed that some of my models appeared red in the Library Editor. What does that mean?*

When our LibraryEditor dynamically generates a bunch of models from the parameter ranges you specify in the “addModels” panel, it tries to instantiate each classifier to see if the given set of parameters is valid. If it traps an error for a set of parameters then it flags that classifier as Red. Note that you don't have to manually remove these invalid models from the temp list. When you add models to the main library list, the invalid ones will be automatically removed.

*Can I use the Library Editor to define parameter ranges for both meta-classifiers and their base classifiers, simultaneously at the same time?*

YES! The Classifier tree in the AddModels panel is recursive which lets you do all sorts of crazy powerful things. You can use multiple layers of meta classifiers and set parameter ranges across each layer and it will generate all the possible combinations. But watch out or you'll end up with extremely huge lists. This was tough to implement but we think it was worth it.

*I have two slightly different model lists that are REALLY long. Is there any easy way to know the difference between the two without staring at them both forever?*

Save both lists as flat files from the Library Editor (the .mlf format), and then just use the diff command line tool.

*Can I modify the default model lists that appear in the Default List Panel? Alternately, can I just add a default list of my own?*

Yes and yes. The model lists are found in the weka/classifiers/meta/ensembleSelection directory of the WEKA classpath. You can modify the model lists there. To add a list, just add it to the appropriate line in the DefaultModels.props file found in that directory and then just drop your list into that directory. Note: this requires that you unjar your weka.jar file if you are using a .jar file.

*Can I modify the regular expressions used to prune the model list in the default model list panel?*

Yes, these regular expressions are found in the DefaultModels.props. Also, if you make changes to these that seem reasonable enough to be rolled back in as defaults, please share them with us as refining these properties is on our list of improvements to make.

## **Developer - FAQ**

*Why the long prediction caching main method in the EnsembleSelection? are you guys crazy?*

Yes, we are crazy! But not because of the prediction caching thing. We cache predictions in the main method so that we can achieve reasonable speed and memory even with a large ensemble.

Our main problem is that Evaluation.evaluateModel() only gives us one instance at a time. The cost of deserializing every one of our models from the file system for each and EVERY testing instance means that 1) your hard drive will be given a thorough workout and 2) test time will take thousands and millions of years.

It's a little hackish but what we do is cache all the test predictions from all of our models first before handing control to Evaluation. So when evaluation hands us instances one at a time, we already have the answers. When this is not run from the command line and EnsembleSelection is run from the GUI, we get around this by keeping ALL of our library models in memory – which is why we run out of memory from the command line so often.

The only way we could get away without doing this is if Evaluation passed all the instances to us at once instead of one at a time. However, this does not seem realistic as it would require significantly changing the rest of WEKA. So while this does seem slightly hackish, it does work well.

*So why didn't you just make the Model List a string argument to a file and the LibraryEditor a separate GUI?*

The closest thing we could find to what we wanted to do with the Library Editor was the Cost Matrix Editor used for classifiers like MetaCost. As much as possible, we wanted to do things the “WEKA” way so we followed this classifier/editor pair as a design pattern and it seemed to work well. We were just trying to follow the precedent.

*Why did you declare the LibraryEditor class a part of weka.gui package? Shouldn't it be somewhere in a subpackage associated with the EnsembleSelection algorithm?*

We did initially, and then it occurred to us that pieces of our LibraryEditor interface could be useful elsewhere. If someone else would like to use the neat classifier-parameter tree

GUI, they can. So what we did was actually make a base class `LibraryEditor` that has all the functionality we thought other people might want, and then extended it with the `EnsembleLibraryEditor` that has all the specific `EnsembleSelection` stuff we need.

*What is going on with the `EnsembleLibrary` and `EnsembleLibraryModel` classes. These seem sort of pointless. Why not just use an `Array of Classifier[]` instead of messing around with a bunch of extra wrapper classes?*

Two reasons. First, we have to be careful about memory. As previously discussed, when creating ensemble models we could be dealing with thousands or possibly tens of thousands of models. Each instantiation of these models could potentially take up a lot of memory. So we created a wrapper class that contains only the information necessary and useful for building lists of classifiers while making sure that the actual instantiations of the classifiers are garbage collected when they need to be.

Second, we are actually extending all of these base classes (respectively called `EnsembleSelectionLibrary` and `EnsembleSelectionLibraryModel`) to do much more work for us with our implementation of Ensemble Selection. So while these two classes might seem a tad simple and unnecessary – they are actually very important base classes that are laying the foundation for the `EnsembleLibrary` and `EnsembleLibraryModel` classes.

*Why do you have those strange static methods at the end of the `LibraryEditor` class?*

Basically, this is a hack. Our problem is that we need to access some `weka.gui` classes such as `PropertyPanel`, `PropertyText`, `CostMatrixEditor`, etc... and all of these classes in `weka.gui` are not declared public. The only alternative I could think of to having these seemingly out of place static methods was to throw all of our classes into the `weka.gui` package as well – it seemed like that would clutter things up too much so I went the static method route.

So I just implemented a bunch of methods that check for the class types of the numbers, cast them to whatever actual class they are, performs the desired arithmetic operation, and then returns the result. I'm sure there's got to be a much better way to do this – please let me know if you can think of one.

## **Known issues**

There are a few outstanding bugs/issues

**Using the GUI (e.g. Explorer) you will run out of memory on reasonably large datasets or model lists.** Please see the User guide section on memory usage problems when running from the GUI. We currently don't know if our workaround that is only

available from the command line is sufficient or of there's some other solution we could implement. For now, we are going to list this as an open issue.

**Currently don't do bounds checking on all values.** Sorry! Stick with defaults or use reasonable values somewhere around the defaults and you'll be fine.

**Phantom default models.** When you don't specify any models in the GUI LibraryEditor, our classifier just defaults to 10 REPTrees with different Seeds. The problem is that these 10 RepTrees get added outside of the GUI interface and won't show up in the LibraryEditor. So the problem is that if you train an Ensemble in the Explorer with no specified models 10 REPTrees will be added to the library. After training, these 10 models will be in the library but won't show up in the Model List GUI. This is a relatively smallish bug. Not sure of the best way to select a default list. Perhaps we should force the user to select some default set before training?

**Intermittent StateExceptions in the Library Editor when editing model lists.** This exception gets raised occasionally from the LibraryEditor GUI. We are not sure what the cause or the fix is. However, this exception seems to be benign and can safely be ignored. Perhaps it should just be trapped and thrown away?

**When using -V (verbose) option to get individual model performance on the validation set, the output is kind of ugly, and when RMSE is used, we actually display (1-RMSE).**

**The .mlf format for model lists is dangerous.** First of all, not all models support their command-line arguments properly. Secondly, the current implementation within EnsembleSelection for turning a classifier and its options in String format in to the actual classifier may have some problems handling things like nested options for meta classifiers wrapping meta classifiers wrapping base classifiers – somewhere in there things can become jumbled. For these reasons, we think it might make sense to force users to use only the .model.xml format.

## **Future Enhancements and other Desirable Improvements**

The following is a prioritized wish list of future enhancements to our implementation of Ensemble Selection in ranking order of desirability.

**Library Models should be saved separately:** One option would be to store every Model in  $n+1$  files, where we're using  $n$ -fold cross validation. One file would be the ".elm" file, somewhat like we have now, which would contain important information such as cached predictions for that model, the number of folds, training data hash, random seed, etc. But the classifiers themselves would be saved in separate ".model" files, with automatically-generated names (e.g. foo.fold1.model, foo.fold2.model, etc...). This would allow easy exporting of trained models, because they'd simply already exist

in the correct format in the working directory. Furthermore, in some cases this could speed up the performance of EnsembleSelection, because in many cases it would not have to load the Classifiers themselves in order to train the ensemble, where currently it does.

**Our multi-class and binary-class lists need to be refined.** Currently, the approximately 1000 models in the multi class and 1400 models in the binary class default model lists were sort of thrown together as a first attempt. To our knowledge, no one has tried to do anything like this before – creating a comprehensive list of Classifiers + parameter ranges that will give you a half-way decent coverage of the model diversity available in WEKA. These two lists that you will find in the Default models panel are based on parameter ranges used in the original paper on Ensemble Selection for most of the same base classifier types. In addition we also added a fair number of WEKA classifiers that seemed to “behave” well on a large number of UCI problems. We think this is a good start, but only that: a start. These lists need to evolve and improve in order to provide the best coverage we can of the model space available from WEKA.

**Refine DefaultModels.props regular expressions.** The regular expressions specifying models with large train times, model sizes, and test times in the Default.props file is currently a sort of a place-holder. The regular expressions currently used to define which models should be removed for large train times, etc... are based on some ad-hoc observations. However, it would be nice to be a little more specific and perhaps base these values on something a little more concrete. Originally we planned on plotting the train times, test times, and file sizes for models across a large number of problems in the UCI datasets and then adding.

**Metric calculation is slow.** We could implement the calculation of metrics separately from Evaluation, and probably see significant speedup. (Evaluation updates everything every time a model is evaluated on an instance)

**Could handle prediction caching more efficiently** - Currently, cache prediction for each model/base-classifier. Could just cache the current prediction of EnsembleSelection itself.

**Make prediction caching an option.** Currently no choice in the matter, might get too big.

**MismatchExceptions can be confusing.** If you use the same dataset for two runs and specify the same working directory, but specify a *different* #folds or validationRatio you'll get a Model mismatch exception, and rightly so. However, people might get confused about this. Perhaps, a solution to this would be to

**Integrate Calibration and other filters.** Possible to do filters now, but we could make it easier in the GUI. Also, some preliminary work has shown that calibrating ensemble

library models (with something like Platt's method) has been shown to clearly improve performance. This was actually on our original list of desired features but unfortunately we weren't able to add this due to time constraints.

**Custom serialization is currently not implemented for Ensemble Library Models.** for EnsembleSelectionLibraryModels so that trained models could be forward/backward compatible across different versions with minimal headaches.

**Making a self-contained EnsembleSelection model.** Currently, you must have a directory containing all the EnsembleLibraryModels that an EnsembleSelection classifier uses available to it for it to work.

**Allow users to import existing WEKA models as Library Models.** We're on the fence as to whether this would be a desirable feature as it would circumvent a lot of our error checking in training library models models “safely” on the same data intended for the Ensemble. However, surely some would find it convenient.