Gallop Documentation

Release 0.1, LT3 Technical Report LT3 13-03

Bart Desmet
Véronique Hoste
LT3 – Language and Translation Technology Team
Faculty of Translation Studies
University College Ghent

David Verstraeten
Jan Verhasselt
Yazzoom BVBA, Erpe Mere

August 14, 2013

Gallop (Genetic Algorithms for Linguistic Learner Optimization) is a python package for optimization of the hyper-parameters of NLP learners. Currently three learner algorithms are supported, namely Timbl, SVMLight and CRF++. Gallop assumes that these learner packages are installed.

CHAPTER

ONE

INSTALLATION AND PREREQUISITES

Gallop can be installed using pip. If the gallop tarball (gallop.tgz) is in the current directory, you can install it with the following command:

```
$ sudo pip install gallop.tgz
```

This installs gallop and fetches and installs its required packages, listed below:

- DEAP, the underlying GA library.
- decorator, utility functions for creating decorators.
- liac-arff, a library for reading ARFF files.
- Paramiko for communication with the submit host over ssh.

All of these are available through the Ubuntu package manager and will be installed along with Gallop.

1.1 Optional dependencies

- If you want to use Gallop on a cluster, you will also need a Torque client for submitting jobs to the cluster. The client is in the standard ubuntu repositories, and can be installed with:

  ```
  $ sudo apt-get install torque-client
  ```

  Then, edit the file */var/spool/torque/server_name* with your favorite editor and add the hostname or IP address of the Torque server for your local cluster. Check that you can access the cluster using e.g.:

  ```
  $ qnodes -a
  ```

  which should return a list of all nodes in the cluster with some additional information.

- If you want to build the documentation, you will need Sphinx. Then execute:

  ```
  $ tar xvzf gallop.tgz
  $ cd gallop/doc
  $ make apidoc
  $ make html
  $ xdg-open build/html/index.html
  ```

  This will open the documentation in a browser window.

1.2 What's next?

A good place to get started is the section on *Getting started*. Once you've read that, you can continue browsing the rest of the documentation:

1.2.1 Getting started

This page describes the different steps involved in doing a simple optimization run of the hyperparameters of a Timbl learner.

Note: The code given below is also available as *getting_started.py* in the *examples/* subdirectory of the gallop.tgz file.

First, we will initialize the logging for Gallop with the `init_logging()` function. This function optionally takes a directory where to write the output, and/or an optional filename for the log file. Here, we will construct an output directory based on the current time (to make sure we don't overwrite any files) and using the default filename gallop.log:

```
>>> output_dir = path.abspath("results_" + timestamp())
>>> log = init_logging(output_dir=output_dir)
```

To run a Gallop optimization, you will need a dataset. Gallop supports several dataset formats for every reader. The preferred way to construct datasets is using the `dataset_factory()` function (see the page on *datasets* for more detail). For instance, the following generates a dataset object in the 'columns' format:

```
>>> file_format = "columns"
>>> data = dataset_factory(file_format,"../coref1000.txt").load()
```

When possible, the dataset object automatically detects the number of features and instances, which we will need later when constructing the learner object:

```
>>> num_instances = data.n_instances()
>>> num_features = data.n_features()
```

Next, we define the evaluation metric to score individuals. It consists of a performance or *fitness* function and a crossvalidation method. This `CVMeasure` will take care of correctly splitting the dataset into folds, evaluating the results for each fold and returning a global fitness value to the optimizer. Here we will use 5-fold cross-validation. The Measure also accepts a parameter nprocs, which will distribute the evaluation of every fold locally over several processes. You shouldn't set this higher than the number of cores in your processor, and if you run into memory problems, try lowering this number.

```
>>> pfun = accuracy
>>> measure = CVMeasure(data,nfolds=5,perfun=pfun,output_dir=output_dir,nprocs=2)
```

Next, we will ask the measure to split the dataset beforehand (this ensures that all individuals are evaluated using the same train- and validation sets).

```
>>> measure.split()
>>> ni = num_instances // measure.nfolds
```

We create the learner object, using default settings. In the case of the TimblLearner, we need to explicitly provide the number of features and instances:

```
>>> tl=TimblLearner(num_features,ni,output_dir=output_dir,file_format=file_format)
```

The final element we need is the optimizer itself, to which we pass the learner and measure:

```
>>> ga =GAOptimizer(tl,measure,output_dir=output_dir,maximize=False)
```

Then, we can start the optimization, here we will do 10 generations with 10 individuals each.

```
>>> res = ga.run(popsize=10,generations=10)
```

After the optimization we can query the best individual and its fitness.

```
>>> log.info("The best individual is: %s with fitness %s" % (res.best_ind,res.best_fitness))
```

You can find additional sample scripts in the examples/ directory.

1.2.2 Learners

Gallop optimizes the hyperparameters of different learning algorithms. It does this through wrapper classes for every learner, derived from a base class `Learner`.

The Learner talks to the underlying executable, passing it the necessary command line arguments and afterwards collecting the predicted classes for the validation set. Learners also know about the hyperparameters that can be optimized as well as their types and ranges (discrete or continuous).

Gallop provides the following learner wrapper classes:

- *TimblLearner*
- *SVMLightLearner*
- *CRFppLearner*

In addition, you can *write your own*.

Supported learners

TimblLearner

The `TimblLearner` wraps the Tilburg Memory-Based Learner binary. It communicates using the python subprocess module and files. Here's an example how to create a TimblLearner instance:

```
>>> learner = TimblLearner(num_features, num_instances, timbl_binary='/usr/bin/timbl',
                           progress="100000", verbosity="+vf", dist_metrics="",
                           file_format="columns")
```

The arguments are:

- *timbl_binary*: the location of the binary
- *num_features*: the total number of features in the dataset to begin with (prior to possibly feature selection)
- *num_instances*: the total number of instances in the dataset
- *progress*: a string directly passed to the timbl binary, indicating how often progress should be reported (see the Timbl manual for more info)
- *verbosity*: a string directly passed to the timbl binary, indicating the level and type of verbosity (see the Timbl manual for more info)
- *dist_metrics*: a string that indicates the distance metrics for every feature that does not follow the global metric, according to the format for the -m switch (see the Timbl manual for more info). This argument is useful in case some features are incompatible with certain distance measures (e.g. Cosine distance for non-numeric features) or in case you want to force the use of a specific distance measure for a certain

feature. For instance, if you want to force the features 3, 4 and 5-10 to use the overlap distance metric, you can use:

```
>>> tl = TimblLearner(num_features, ni, output_dir=output_dir,file_format=file_format, dist_
```

Note that in case of feature selection, this string is automatically adjusted in accordance to the (de-)selected features.

- *file_format*: a string directly passed to the timbl binary indicating the format of the input files.
- *erase_full_output*: a boolean (default True) indicating whether the output files generated by Timbl should be erased after evaluating the individual. This is especially useful for larger datasets, where the disk usage can grow quickly. Note that the TimblLearner always writes a file per fold containing the actual and predicted class values for the validation set, regardless of the value of this keyword.
- *write_idb_file*: a boolean (default False) indicating whether the instance database (idb file) should be saved or not.

The hyperparameters that can be optimized using the GAOptimizer are:

- *-a*: the algorithm. Default search space: 0-3.
- *-w*: the feature weighting. Default search space: 0-3.
- *-k*: the neighbourhood size. Default search space: 1-10.
- *-q*: TRIBL offset. Default search space: 1-num_features.
- *-b*: the number of bootstrap samples to use. Default search space: 2-num_instances.
- *-m*: the distance metrics to use for each feature. Default search space: *['O', 'M', 'J', 'D', 'C', 'N', 'L', 'DC'].*
- *-d*: the voting type. Default search space: *['Z', 'ID', 'IL', 'ED'].* In case this hyperparameter gets the value *ED*, two additional values are randomly generated for the alpha and beta parameters (these determine the shape of the weighting curve, see Fig. 5.2 in the Timbl manual). The default search space for alpha is [0.01, 5] and for beta [0.01, 3]. If you want to change the search space for these alpha and beta parameters, you can do this as follows:

```
>>> tl.get_hyperparam('d').alpha_range = [0.5,3.5]
```

If you would like to set a fixed value for alpha or beta, set the limits of the range to the same value, for example:

```
>>> tl.get_hyperparam('d').beta_range = [0.5,0.5]
```

Note: Note that the default settings for the TimblLearner includes all possible distance metrics for the global metric hyperparameter ('-m'), some of which may be incompatible with your dataset (some distance metrics don't work with non-numeric features for instance). There are three ways to deal with this:

- Force the distance metric for certain features to a fixed value using the *dist_metrics* keyword when instantiating the learner (see above).
- Exclude the incompatible distance metrics from the search options, using, e.g.:

```
>>> tl.get_hyperparam('m').choices = ['O', 'L', 'M', 'J']
```

- Simply ignore these individuals: they receive a very low fitness and quickly disappear from the population anyway.

For any of these hyperparameters, you can specify that the Timbl default setting should be used instead of including it in the optimization, by setting the choices for that hyperparameter to *[None]*:

```
>>> learner.get_hyperparam('k').choices = [None]
```

An example script showing this in more detail is included in the gallop distribution tarball, as *examples/default_parameters.py*.

SVMLightLearner

CRFppLearner

1.2.3 Datasets

Gallop wraps several learners, each of which has its own dataset input format. Gallop manipulates these raw datasets (for instance, to do feature selection), and as such it needs to know what type of dataset is being used during an optimization. This information and manipulation is handled by one of the subclasses of the `Dataset` class.

Instead of instantiating a Dataset object directly, the preferred way to do this is by using the `dataset_factory()` function. This function used as follows:

```
>>> dataset = dataset_factory(format, filename, nfeats)
```

and will return a dataset object of the correct type based on the arguments. The *format* argument is a string denoting the dataset type, e.g., *'C4.5'* (see below for all supported formats). The filename points to the original text file containing the dataset, and the optional argument *nfeats* is an integer denoting the number of features present in the dataset, which is required for certain sparse datasets - the formats for which this is mandatory are indicated below.

The dataset object created as above cannot be used yet. In order to actually load the data and set the properties of the dataset (such as the number of instances), you call the load() function:

```
>>> dataset.load()
```

Once this is done, you can query the number of features, instances and columns:

```
>>> num_instances = data.n_instances()
>>> num_features = data.n_features()
```

and pass it to a GAOptimizer.

Supported formats

Gallop can handle the following dataset types:

- *sparse*: the Timbl sparse format. *nfeats* is required.
- *binary*: the Timbl sparse binary format. *nfeats* is required.
- *columns*: space separated values
- *C4.5*: comma separated values
- *arff*: Weka ARFF format
- *sentence*: CRF++/Yamcha sentence dataset. 'Instances' are sentences, separated by newlines in the textfile.
- *docmarker*: CRF++/Yamcha document dataset. 'Instances' are documents, separated by the string -DOCSTART- on a separate line. Documents consist of sentences, separated by newlines in the textfile.
- *sparsesvmlight*: Sparse SVMLight dataset, where instances are represented as index:value pairs separated by spaces. Lines starting with # are ignored. *nfeats* is required.

TODO: add dataset formats for other learners once they are implemented.

1.2.4 Optimization using Genetic Algorithms

Gallop optimizes the hyperparameters of the learners and does feature selection using a Genetic Algorithm (GA). The combined 'configuration' (selected hyperparameter values and features) is called a *genotype*. The GA works in two stages:

- a population of individuals is created, initially at random. The *fitness* of every *individual* is evaluated.
- **based on the fitness of the individuals, a new population is generated by**
  - creating new individuals from the previous population using genetic operators such as *mutation* and *cross-over*.
  - carrying over the best individuals of the previous population to the current population (elitist approach)

The optimization of the learner is handled by a `GAOptimizer`. A GAOptimizer is instantiated as follows:

```
>>> ga = GAOptimizer(tl,measure,output_dir=output_dir,maximize=True)
```

where the arguments are:

- *tl*: a `Learner` instance.
- *measure*: a `Measure` instance.
- *output_dir*: a string denoting the directory where the output should be written (log files, results, ...). See the page on *I/O* for more detail.
- *maximize*: a boolean value indicating whether the measure should be maximized (i.e., performance) or minimized (i.e., loss).

The evaluation of the fitness consists of training and validating the learner using the current genotype values. Gallop supports both hyperparameter optimization and feature selection. We will discuss both features separately.

Hyperparameter optimization

By default, every supported *hyperparameter* for a given learner will be optimized using default values for the sequence of values (for discrete hyperparameters), or the allowed range (for continuous hyperparameters). Some learners (such as the `TimblLearner`) have custom hyperparameter classes which allow more complicated usage - normally you shouldn't use these directly as a user.

You can adjust the set or range of values for every hyperparameter using `get_hyperparam()`. Read the section on *hyperparameters* for more detail on how to use this functionality.

Feature selection

The optimizer can also perform feature selection, optionally simultaneously with the hyperparameter optimization. This can be done in two ways:

- on individual features
- on feature sets

Feature sets can be specified by the user as 'belonging together' in some way, and consequently every feature set will be included or excluded completely by the feature selection. Feature sets can be specified using a `FeatureSets` object as follows:

```
>>> feats = FeatureSets(1,2,5,[6,9,10],range(11,15),label_col=35)
```

This defines a FeatureSets object with three sets containing only a single feature ([1], [2], [5]), a set containing the features with index 6, 9 and 10; and a set containing the features from 11 to 14 (note that when specifying a Python range, the last element is always excluded). Finally, the label_col keyword argument specifies which column index corresponds to the instance labels.

If you want to do feature selection, you can pass the FeatureSets object to the optimizer using the keyword argument `features`.

Measures

The way the fitness of a single individual is determined, is handled by a `Measure` object. This object takes care of splitting the dataset into train and test sets in case of cross-validation and aggregating the scores. Currently there is only one type of measure which covers most use cases called the `CVMeasure`, but if you want to write your own you can subclass the `Measure` base class yourself.

A *CVMeasure* is created as follows:

```
>>> measure = CVMeasure(data,nfolds=5,perfun=pfun,output_dir=output_dir,nprocs=2)
```

with these arguments:

- *data*: a *dataset* object
- *nfolds*: the number of folds for *cross-validation*
- *perfun*: the scoring function (see the documentation for `gallop.performance` for a list of possible functions)
- *output_dir*: the output directory
- *nprocs*: optional. This will run the evaluation of every fold using a different subprocess. If you have a multicore machine, you can set this to the number of cores in your processor to run the optimization maximally in parallel - in case you run out of memory you should lower this number.

Once the measure object is created, the `split` method can be used to create the individual train and test dataset files in the output_directory. This method takes an optional boolean argument *shuffle_data* that indicates if the instances should be shuffled before creating the dataset files (default is *True*:

```
>>> measure.split(shuffle_data=False)
```

In most datasets, the instances are words. However, sequence taggers work on sequences of words. These dataset formats are supported through the `SentenceDataset` and `DocMarkerDataset`. For these datasets, the Measure interprets the instances as sentences and documents respectively, meaning that the Measure will only split the dataset between sentences and documents, respectively.

The `SentenceDataset` and `DocMarkerDataset` can be used with a `CVMeasure` without problems, but this can sometimes result in folds where the train and validation sets differ substantially in size (esp. in the case of DocMarkerDatasets, where the amount of words in a document can vary considerably). To counter this, Gallop offers a `BalancedCVMeasure`, which will distribute the instances (sentences and documents) over the folds, such that the total amount of words in every fold is as equal as possible, while still respecting the sentence and document boundaries.

Running the optimization

The `run()` method takes three arguments: *popsize* setting the number of individuals in a single generation, *generations* indicating the maximum number of generations to be evaluated, provided no other termination criterium is satisfied before, and optionally an initial population *initial_pop* to start from - see also the section below on checkpointing. It returns a variable with the following members:

- *pop*: the last population that was evaluated
- *best_ind*: the best individual of the optimization run
- *best_fitness*: the fitness of the best individual
- *trace_dir*: the directory containing the output
- *per_fun*: the measure (validation method + performance function) used for the optimization
- *stats*: a DEAP Statistics object, containing several statistical properties of the optimization run (see this page)
- *history*: a DEAP History object, containing the full genealogy tree of the optimization run

Checkpointing

After the evaluation of every generation, the optimizer writes trace files to the output directory, both in human-readable and binary form. (see also the documentation on *Output*). The binary trace file stores the population after every generation and the whole history of the optimization, and can be used to restart or continue optimizations from that population, e.g. if an error has occurred or if you want to fine-tune the performance further.

The checkpoint files can be used as follows (the entire code is available as *examples/checkpointing.py*)

```
# It's assumed that datasets, measures etc. have all been defined
# We run the optimization for the first time for ten generations

# Do the first optimization
print 'Running the GA for ten generations...'
res = ga.run(popsize=10,generations=10)
print 'Done...'

all_trace_files = glob(path.join(output_dir, 'trace','*.pickle'))
latest_trace = max(all_trace_files, key=path.getmtime)

# Load the checkpoint file containing the last population
with open(latest_trace) as tracefile:
    pop, history = pickle.load(tracefile)

# Start a new optimization run from the last population
print 'Continuing the GA for an additional ten generations from pickle file %s...'%latest_trace
res = ga.run(popsize=10,generations=10, initial_pop = pop, history = history)
print 'Done...'
```

Initializing the optimization

If you have some idea of good parameter values to start the optimization from, you can give an initial population of individuals to the optimizer.

This can be done as follows (only the relevant part is shown, the entire code is available as *examples/initial_population.py*)

```
# Create a set of 3 random individuals
initial_pop = [Individual.create_individual(tl, None, FitnessMax()) for _ in range(3)]

# Set some hyperparameter values to chosen values, others remain random
initial_pop[0].hyperparams['w'] = 2
```

```
initial_pop[0].hyperparams['a'] = 0
initial_pop[1].hyperparams['w'] = 1
initial_pop[2].hyperparams['k'] = 4

# the optimizer
ga = GAOptimizer(tl,measure,output_dir=output_dir,maximize=True)

# perform the optimization, feeding in the initial population created earlier
res = ga.run(popsize=10,generations=100, initial_pop = initial_pop)
```

Convergence

The optimization run halts if either of the following conditions is fullfilled:

- the maximum number of generations has been evaluated (specified by the `generations` keyword argument)
- the mean fitness of the populations has not changed much over the last window of generations (specified by the `fit_tol` and `fit_window` keyword arguments)
- the standard deviation of the current population is small (indicating the population has converged - specified by the keyword `std_tol`)

Distributed optimization

You can also run the evaluation of a population in parallel on a computer cluster. The cluster should have a working installation of Torque/OpenPBS, and all worker nodes should have access to a shared file system.

Distributed optimization is handled by the `DistGAOptimizer` class. A DistGAOptimizer is instantiated in the same way as a local GAOptimizer, but with some additional arguments:

```
>>> ga = DistGAOptimizer(tl,measure,output_dir=output_dir,maximize=True,
                  username="user",password="password",submit_node="submit_host",
                  remote_dir="/shared/gallop")
```

The additional arguments are:

- *username*, *password*: the login credentials for SSH'ing to the submit node. TODO: check passwordless login.
- *submit_node*: the hostname or IP address of the node from which the jobs will be submitted. If your local computer can submit jobs to the cluster, you can leave this on the default setting of *localhost*.
- *remote_dir*: a shared remote directory that all compute nodes have access to, which will be used to exchange information between the compute nodes (data files etc.)
- *walltime*: a string (hh:mm:ss) indicating an estimate of how long the evaluation of a single individual takes. This will be added to the job submit script and is required for some clusters where different queues exist for jobs of different runtimes. See the documentation for the qsub command for more information on this option.
- *mem*: a string (for example, '500kb' or '4Gb') indicating an estimate of the maximal memory requirements for a single job. This will be added to the job submit script and is required for some clusters where the allocation of jobs to compute nodes is decided based on the memory requirements. See the documentation for the qsub command for more information on this option. The default is None, which will cause the job to get the default setting used on the cluster (usually this is 2Gb).
- *poll_interval*: an integer indicating the interval in seconds for polling the job status. Default is 30s.

Once you have instantiated your DistGAOptimizer as above, you can use it in the same way as you would use a regular GAOptimizer, except all evaluations of an individual will be submitted as jobs on the cluster.

Once a job is submitted the optimizer will poll its status periodically (set by the poll_interval argument, see above), and print the status of the jobs to the screen and the log. Note that you can also monitor the status of your job (and get more information such as the nodes it is running on) using the qstat command.

1.2.5 Hyperparameters

A *hyperparameter* is a value that controls the behaviour of the learning algorithm itself, such as the number of neighbours K in a KNN algorithm. Generally speaking, hyperparameters can be of two types:

- `DiscreteHyperparam`: defined by a finite sequence of possible values. This can also be defined by, e.g., using range(n).
- `ContinuousHyperparam`: characterised by a minimal and maximal value [min, max]. These can both be set to +- infinity if the hyperparameter is unbounded.

If you should encounter a case where the hyperparameter of a learner does not fit into either of the above cases, you can write your own by subclassing `Hyperparam` and implementing the method `random_value()` that returns a random value for that hyperparameter.

Every learner supported by gallop has a set of hyperparameters that can be optimized. See the *documentation pages for the other learners* for more info on their supported hyperparameters. If a learner is constructed with the default parameters, it contains a pre-defined dictionary of hyperparameters with default ranges/choices. The dictionary of hyperparameters the learner will optimize along with their allowed ranges/choices can be queried using the method `get_hyperparams()`:

```
>>> tl= TimblLearner(num_features = 20, num_instances=1000)
>>> tl.get_hyperparams()
[('a',
  Discrete hyperparameter. Description: 'Algorithm'. Possible values: [0, 1, 2, 3]),
 ('w',
  Discrete hyperparameter. Description: 'Feature weighting'. Possible values: [0, 1, 2, 3]),
 ('k',
  Discrete hyperparameter. Description: 'neighbourhood size'. Possible values: [1, 2, 3, 4, 5, 6, 7,
 ('b',
  Discrete hyperparameter. Description: 'number of bootstrap samples'. Possible values: [2, 3, 4, 5,
 ('q',
  Discrete hyperparameter. Description: 'TRIBL offset'. Possible values: [1, 2, 3, 4, 5, ..., 15, 16,
 ('m',
  Distance metric hyperparameter. Description: 'Feature distance metric'. Possible values: ['O', 'M',
 ('d',
  Voting type hyperparameter. Description: 'Class voting weight type'. Possible values: ['Z', 'ID', '
```

Note that the method `get_hyperparams()` in an interactive session gives the output above, but it actually returns a list of tuples (name, hyperparam), where hyperparam is the actual object. This allows you to change the valid ranges for that hyperparameter. There is also a method `get_hyperparam ()` to retrieve a single hyperparameter using its name. For example:

```
>>> neighbour_size = tl.get_hyperparam('k')
>>> neighbour_size.choices = range(1,20)
>>> tl.get_hyperparam('k')
Discrete hyperparameter. Description: 'neighbourhood size'. Possible values: [1, 2, 3, 4, 5, ..., 15,
```

1.2.6 Output

During an optimization run, gallop writes several files. This page explains their contents.

All output for an optimization run is stored in an output directory given by the user. It is common usage to rerun the same script several times, which may result in overwriting certain files. For this reason, it's advised to include a timestamp somewhere in the output directory name, for instance:

```
output_dir = path.abspath("results_" + timestamp())
```

The output directory contains a logfile and three subdirectories:

- `gallop.log`: this file contains the complete log output of the optimizer
- `data`: this subdirectory contains generated train and validation files for every fold, which can either be used directly during the optimization or which will be further manipulated in case the user requested
- `trace`: this subdirectory contains the trace and checkpoint files, written after the complete evaluation of every generation of individuals.
  - The trace files (extension `.txt`) are human-readable. The top line shows the different parameter names, followed by the values for those parameters and the fitness of the corresponding *individual*. An example trace file for a TimblLearner is shown below

    ```
    "a" "b" "d" "k" "m" "q" "w" fitness_0
    1 8 ID 8 C 19 0 0.982
    0 139 ED:1.43890388705:2.7317819108 2 C 11 0 0.982
    2 121 Z 1 N 7 2 0.982
    1 156 ID 5 M 43 3 0.982
    0 158 IL 3 J 37 2 0.982
    1 191 ED:3.04682468196:0.393761676126 9 N 19 0 0.982
    0 165 Z 7 J 45 0 0.982
    2 144 ED:2.93505822539:1 7 DC 33 3 0.982
    3 144 ID 8 D 46 3 0.982
    1 106 ED:4.64991014743:1 6 N 35 3 0.982
    ```

  - The checkpoint files (extension `.pickle`) are binary and contain the state of the population at that point. These files can be used to continue a previous optimization run in case of failure, or to evaluate more generations from a certain starting population. See the section on *Checkpointing* for more information.
- `learner`: this subdirectory contains individual subdirectories for every training/evaluation of a single individual. The per-individual subdirectories are named with a timestamp + a random suffix to avoid collisions in the case of multiprocessing. Every individual subdirectory contains:
  - the learner output for the validation set for every fold, named `DATASETNAME_fold_N_test.txt.out`
  - any intermediate output files generated by the learner (e.g., for the TimblLearner the training `*.idb` files).
  - any intermediate output files generated by the learner (e.g., for the TimblLearner the training `*.idb` files).
  - for every fold, a log file containing the command line used to invoke the learner and the corresponding screen output.

1.2.7 Writing your own Learner subclass

If you want to write a wrapper class for your own learner, you can do this by subclassing the Learner base class and inheriting some of its functionality.

Logging

Gallop logs its activity thoroughly to file and screen, using the standard Python logging module. If you write your own Learner, it's wise to do the same. If you inherit from the Learner base class, your instance will have a member object log, which you can use to log messages of varying importance levels (debug up to exception), e.g.:

```
class myLearner(Learner):
    def train(self, filename):
        self.log.debug('Starting to train myLearner...')
        if not path.exists(filename):
            e = Exception('Training file %s not found!'%filename)
            self.log.exception(e)
            raise e
```

Hyperparameters

Normally the Hyperparam class and its children are only used internally in the Learner. However, if you want to implement you

- name: the command line switch
- value: the current value of the hyperparameter
- description: a short description of what the hyperparameter controls

Read the section on *hyperparameters* for more detail on how they work and how to implement your own.

Subclassing Learner

The Learner base class already has some functionality that is suitable for most cases. If you want to write a new Learner, you pr

- The Learner instances can be pickled
- Set and get methods for one or all hyperparameters

Some of the methods of the Learner *can* be implemented/overridden by the subclass but in some cases the default implementatio

- Constructor `__init__()`: the base implementation simply checks if the output directory exists if it is given, but you will probably want to add some additional checks here (e.g. if the learner binary exists). This is also the best place to define the hyperparameters and their acceptable values, using, e.g.:

  ```
  class myLearner(Learner):
      def __init__(self):
          self.add_hyperparam(DiscreteHyperparam("k",range(50), "Number of neighbours"))
          self.add_hyperparam(ContinuousHyperparam("b",(-1,1), "Regularization parameter")
  ```

- `make_feature_compatible(self, hp, enabled_features)`: this method filters the hyperparameters such that they are compatible with the selected features. The hyperparameters hp are passed as a {name:value} dictionary, enabled_features is a sequence of feature indices. It returns the updated dictionary of hyperparameters.
- `train_and_predict(self, train_file, validation_file)`: in the base class this simply calls train() and predict() consecutively, but for some learners this can be implemented more efficiently. If that is the case, you can override this method, and the train() and predict() methods below do not need to be overridden. This function returns a tuple of two sequences, one containing the true class for every instance and one containing the predicted class for every instance in the validation set. This is the only function that is called by the optimizer, so you will likely only need to override this method.

In some cases it may be useful to have separate train and predict methods. If that is the case, you can override the following two

- `train(self, filename)`: trains the learner with the current hyperparameter values using the *train set* given by filename. This function doesn't return anything.
- `predict(self, filename)`: applies a trained learner to the *validation set* given by filename. This function returns a tuple of two sequences, one containing the true class for every instance and one containing the predicted class for every instance in the validation set.

These last three methods communicate with the underlying learner, either through a Python API or directly via the command line. In the latter case, you can use the convenience function `run_process`. This function has the following arguments:

- a path to the binary
- a sequence of command line arguments (as strings)
- an output directory
- a logger

The function will run the binary (using python's subprocess module) with the given command line arguments and take care of the details such as logging and making sure the process exited properly.

1.2.8 Glossary

cross-over A genetic operator that splits and swaps genotypes between individuals at a cross-over point. See also the Wikipedia entry on cross-over.

cross-validation The practice of splitting a dataset into K parts (called folds) whereby K-1 parts are used for training and 1 part for validation, repeating the process K times such that every part is used for validation exactly once.

individual An instance of a learner with certain hyperparameters and/or selected features that is represented by a genotype. An individual can be evaluated to assess its fitness.

feature A distinguising characteristic of an instance. Features can be discrete (numerical or taken from a list) or continuous (numerical).

fitness A scalar value that represents the value of an individual. This is typically the validation score according to some evaluation metric such as accuracy.

genotype A vector representing a combination of hyperparameter values and (if applicable) selected features. A genotype determines the properties of an individual.

hyperparameter A setting that governs the behaviour of the training of the model, can be set by the user. This is different from the actual model parameters, which are set by the training algorithm (e.g., the weights of an SVM).

instance A single train or validation element. An instance is characterised by a number of features and a class.

label An indication of which category (out of a finite set of possibilities) the instance belongs to.

mutation Modifying a gene to a random value. For binary genes, this amounts to flipping the bit, for continuous-valued genes this amounts to drawing a new random value in the allowed range for that gene. See also the Wikipedia entry on mutation.

train set A set of instances that is used for training the model.

validation set A set of unseen instances that is used to evaluate the performance of a trained model.

1.2.9 gallop Package

`data` Module

Provides a set of wrappers around various types of data sources. The classes below should rarely be instantiated directly. The provided factory function `dataset_factory()` should be used instead.

class `gallop.data.`**`ARFFDataset`**(*filename*)

Bases: `gallop.data.Dataset`

Wrapper for datasets in ARFF format.

Parameters **filename** – filename to load

`load`(*predict_mode=False*)

Load the dataset into memory.

Parameters **predict_mode** – if True, two labels should present in each instance, one for the true value and one for the predicted value. In that case the last item in each instance will be a list.

`n_cols`()

The number of columns (features + labels) in each instance

`n_instances`()

The number of instances in the dataset

`save`(*filename*, *instance_filter=None*, *feature_filter=None*)

Save the dataset to disk.

Parameters

- **filename** – the filename to write to
- **instance_filter** – a list of (0 based) indices into the instances, only these will get written. If None, all instances will be saved.
- **feature_filter** – a list of feature indices (1 based)

`uri`()

The location of the data as a string (e.g., URL, absolute path, ..)

class `gallop.data.`**`Dataset`**

Bases: `object`

Abstract base class for datasets.

`instance_lengths`()

The length of all instances as a list. Base implementation assumes every instance has length 1.

`load`(*predict_mode=False*)

Load the dataset into memory.

Parameters **predict_mode** – if True, two labels should present in each instance, one for the true value and one for the predicted value. In that case the last item in each instance will be a list.

`n_cols`()

The number of columns (features + labels) in each instance

`n_features`()

The number of features in an instance. Base implementation assumes one less than the number of columns.

`n_instances`()

The number of instances in the dataset

save (*filename*, *instance_filter=None*, *feature_filter=None*)
Save the dataset to disk.

Parameters

- **filename** – the filename to write to
- **instance_filter** – a list of (0 based) indices into the instances, only these will get written. If None, all instances will be saved.
- **feature_filter** – a list of feature indices (1 based)

uri ()
The location of the data as a string (e.g., URL, absolute path, ..)

class gallop.data.**DocMarkerDataset** (*filename*, *colsep=' '*, *sentsep='n'*, *docsep='-DOCSTART-'*)
Bases: gallop.data.Dataset

Dataset where documents are delimited by certain docmarkers (strings). This dataset is meant for use with sequence taggers. In this class, instances are interpreted as documents (to avoid splitting documents between folds).

Also implements str,len,iter, and getitem

param filename dataset filename

keyword colsep column sepator character (default: ' ')

keyword sentsep sentence sepator character (default: '

')

keyword docsep document separator string (default: '-DOCSTART-')

instance_lengths ()
The length of all instances as a list. Base implementation assumes every instance has length 1.

load (*predict_mode=False*)
Load the dataset into memory.

Parameters **predict_mode** – if True, two labels should present in each instance, one for the true value and one for the predicted value. In that case the last item in each instance will be a list.

n_cols ()
The number of columns (features + labels) in each instance

n_instances ()
The number of instances in the dataset

save (*filename*, *instance_filter=None*, *feature_filter=None*)
Save the dataset to disk.

Parameters

- **filename** – the filename to write to
- **instance_filter** – a list of (0 based) indices into the instances, only these will get written. If None, all instances will be saved.
- **feature_filter** – a list of feature indices (1 based)

uri ()
The location of the data as a string (e.g., URL, absolute path, ..)

class `gallop.data.`**`SentenceDataset`**(*filename*, *colsep=' '*, *sentsep='n'*)

Bases: `gallop.data.Dataset`

Dataset where the instances are sentences. This dataset is meant for use with sequence taggers.

Also implements str,len,iter, and getitem

param filename dataset filename

keyword colsep column sepator character (default: ' ')

keyword sentsep sentence sepator character (default: '

')

`instance_lengths`()

The length of all instances as a list. Base implementation assumes every instance has length 1.

`load`(*predict_mode=False*)

Load the dataset into memory.

Parameters **predict_mode** – if True, two labels should present in each instance, one for the true value and one for the predicted value. In that case the last item in each instance will be a list.

`n_cols`()

The number of columns (features + labels) in each instance

`n_instances`()

The number of instances in the dataset

`save`(*filename*, *instance_filter=None*, *feature_filter=None*)

Save the dataset to disk.

Parameters

- **filename** – the filename to write to
- **instance_filter** – a list of (0 based) indices into the instances, only these will get written. If None, all instances will be saved.
- **feature_filter** – a list of feature indices (1 based)

`uri`()

The location of the data as a string (e.g., URL, absolute path, ..)

class `gallop.data.`**`SimpleDataset`**(*filename*, *colsep=' '*)

Bases: `gallop.data.Dataset`

Simple file based dataset, one line per instance and features/labels are separated by the given separator.

Also implements str,len,iter, and getitem

Parameters

- **filename** – dataset filename
- **colsep** – column separator character

`load`(*predict_mode=False*)

Load the dataset into memory.

Parameters **predict_mode** – if True, two labels should present in each instance, one for the true value and one for the predicted value. In that case the last item in each instance will be a list.

`n_cols`()

The number of columns (features + labels) in each instance

n_instances()
The number of instances in the dataset

save(*filename*, *instance_filter=None*, *feature_filter=None*)
Save the dataset to disk.

Parameters

- **filename** – the filename to write to
- **instance_filter** – a list of (0 based) indices into the instances, only these will get written. If None, all instances will be saved.
- **feature_filter** – a list of feature indices (1 based)

uri()
The location of the data as a string (e.g., URL, absolute path, ..)

class gallop.data.**SparseBinDataset**(*nfeats*, *filename*)
Bases: gallop.data.Dataset

Wrapper for datasets in sparse binary format, "binary" in Timbl.

Parameters

- **nfeats** – total number of features
- **filename** – filename to load

load(*predict_mode=False*)
Load the dataset into memory.

Parameters **predict_mode** – if True, two labels should present in each instance, one for the true value and one for the predicted value. In that case the last item in each instance will be a list.

n_cols()
The number of columns (features + labels) in each instance

n_instances()
The number of instances in the dataset

save(*filename*, *instance_filter=None*, *feature_filter=None*)
Save the dataset to disk.

Parameters

- **filename** – the filename to write to
- **instance_filter** – a list of (0 based) indices into the instances, only these will get written. If None, all instances will be saved.
- **feature_filter** – a list of feature indices (1 based)

uri()
The location of the data as a string (e.g., URL, absolute path, ..)

class gallop.data.**SparseDataset**(*nfeats*, *filename*)
Bases: gallop.data.Dataset

Wrapper for datasets in sparse format, "sparse" in Timbl.

Parameters

- **nfeats** – total number of features
- **filename** – filename to load

`load`(*predict_mode=False*)
Load the dataset into memory.

Parameters **predict_mode** – if True, two labels should present in each instance, one for the true value and one for the predicted value. In that case the last item in each instance will be a list.

`n_cols`()
The number of columns (features + labels) in each instance

`n_instances`()
The number of instances in the dataset

`save`(*filename*, *instance_filter=None*, *feature_filter=None*)
Save the dataset to disk.

Parameters

- **filename** – the filename to write to
- **instance_filter** – a list of (0 based) indices into the instances, only these will get written. If None, all instances will be saved.
- **feature_filter** – a list of feature indices (1 based)

`uri`()
The location of the data as a string (e.g., URL, absolute path, ..)

class `gallop.data.SparseSVMLightDataset`(*nfeats*, *filename*)
Bases: `gallop.data.Dataset`

Wrapper for datasets in SVMLight sparse format.

Parameters

- **nfeats** – total number of features
- **filename** – filename to load

`load`(*predict_mode=False*)
Load the dataset into memory.

Parameters **predict_mode** – if True, two labels should present in each instance, one for the true value and one for the predicted value. In that case the last item in each instance will be a list.

`n_cols`()
The number of columns (features + labels) in each instance

`n_instances`()
The number of instances in the dataset

`save`(*filename*, *instance_filter=None*, *feature_filter=None*)
Save the dataset to disk.

Parameters

- **filename** – the filename to write to
- **instance_filter** – a list of (0 based) indices into the instances, only these will get written. If None, all instances will be saved.
- **feature_filter** – a list of feature indices (1 based)

`uri`()
The location of the data as a string (e.g., URL, absolute path, ..)

`gallop.data.`**`dataset_factory`**(*format*, *filename*, *nfeats=None*)

Factory function for creating the correct `Dataset` subclass based on a format string. This should be preferred over creating instances directly. Supported formats are (one line per instance in all cases):

- columns: space separated values
- C4.5: comma separated values
- binary: timbl sparse binary format
- sparse: timbl sparse format
- arff: weka ARFF format
- sentence: CRF++/Yamcha sentence dataset (sentences separated by
- newlines)
- docmarker: CRF++/Yamcha document dataset (sentences separated by
- newlines, documents separated by the line -DOCSTART-)

`measure` Module

Provides a set of model selection algorithms to compare learners.

class `gallop.measure.`**`BalancedCVMeasure`**(*data*, *nfolds=5*, *output_dir=None*, *nprocs=1*, ***kwargs*)

Bases: `gallop.measure.CVMeasure`

k-fold Cross validation with balanced folds. This measure distributes the instances over the folds such that the total lengths of all instances is as equal as possible across folds. This measure should be used with datasets where instances are made up of subentitities (e.g., documents for the DocMarkerDataset).

Parameters

- **data** – The `Dataset` instance to use
- **nfolds** – number of folds
- **nprocs** – number of processors to use when calculating folds
- **output_dir** – the directory to write fold data to (a data subdirectory will be created)
- ****kwargs** – passed on to the base class

class `gallop.measure.`**`CVMeasure`**(*data*, *nfolds=5*, *output_dir=None*, *nprocs=1*, ***kwargs*)

Bases: `gallop.measure.Measure`

k-fold Cross validation

Parameters

- **data** – The `Dataset` instance to use (file based data assumed)
- **nfolds** – number of folds
- **nprocs** – number of processors to use when calculating folds
- **output_dir** – the directory to write fold data to (a data subdirectory will be created)
- ****kwargs** – passed on to the base class

`foldfiles`()

Return a list of tuples containing the training and testing filenames for each fold.

nfolds
The number of folds.

score(*learner*, *enabled_features=None*)
Apply the model selection metric to the given learner and then return the result of applying the performance function (passed to the class constructor).

Parameters

- **learner** – the learner to score
- **enabled_features** – a list of features that are selected, typically passed on to the learner

split(*shuffle_data=True*)
Split the dataset into 2*k files. For each fold the training set and test set are saved as two separate files.

Parameters **shuffle_data** – whether the instances should be shuffled

before splitting into folds

class `gallop.measure.`**`Measure`**(*perfun=None*, *display_perfuns=[]*)
Bases: `object`

Base class for a model selection algorithm, or measure. Given a model (learner) and performance function it returns a numeric score indicative of the model quality.

Parameters

- **perfun** – the performance function to use, see `gallop.performance`
- **display_perfuns** – extra performance functions, calculated and logged but not used in any functional way.

score(*learner*, *enabled_features=None*)
Apply the model selection metric to the given learner and then return the result of applying the performance function (passed to the class constructor).

Parameters

- **learner** – the learner to score
- **enabled_features** – a list of features that are selected, typically passed on to the learner

class `gallop.measure.`**`ValidationSetMeasure`**(*data_train*, *data_test*, ***kwargs*)
Bases: `gallop.measure.Measure`

Use a separate, fixed validation set to measure model performance.

Parameters

- **data_train** – A `Dataset` instance containing the training data
- **data_test** – A `Dataset` instance containing the validation data
- ****kwargs** – passed on to the base class

score(*learner*, *enabled_features=None*)
Apply the model selection metric to the given learner and then return the result of applying the performance function (passed to the class constructor).

Parameters

- **learner** – the learner to score
- **enabled_features** – a list of features that are selected, typically passed on to the learner

`performance` Module

A set of functions used to estimate the performance of a classifier.

`gallop.performance.`**`accuracy`**`(`*true*, *predicted*`)`

Calculates the accuray of a classifier.

Parameters

- **true** – iterable of the true class lables
- **predicted** – iterable of the predicted lables

Return type the accuracy as defined here

Note: requires the labels are converible to booleans according to `gallop.util.POS_VALS` and `gallop.util.NEG_VALS`

`gallop.performance.`**`accuracy_multiclass`**`(`*true*, *predicted*`)`

Calculates the accuray of a multiclass classifier, defined as the fraction of correct classifications. Note that this does not take into account any potential class imbalances.

Parameters

- **true** – iterable of the true class lables
- **predicted** – iterable of the predicted lables

Return type

the accuracy as defined here

`gallop.performance.`**`check_dimensions`**`(`*f*`)`

Return a version of the performance function where the dimensions of the vectors are checked beforehand.

`gallop.performance.`**`misclass`**`(`*true*, *predicted*`)`

Calculates the number of misclassified instances.

Parameters

- **true** – iterable of the true class lables
- **predicted** – iterable of the predicted lables

Return type number of wrongly predicted instances

`util` Module

A set of general helper functions and classes

class `gallop.util.`**`LoggerWriter`**`(`*logger*, *level*`)`

Wrapper to allow loggers to be used as file like objects

`write``(`*message*`)`

class `gallop.util.`**`SSHWrapper`**`(`*host='localhost'*, *username='bart'*, *password=None*, *working_dir='~'*`)`

Bases: `object`

Simple wrapper around the Paramiko SSH Client object. Handling of username/password arguments is the same as in Paramiko.

`close``()`

`connect``()`

download(*files*)

is_connected()

remote_exec(*command*)

upload(*files*)

gallop.util.**add_logger**(*f*)
Automatically add a log instance variable to a class. Correctly configures a python logger. To be applied to the __init__ method.

gallop.util.**append_to_fname**(*fname*, *s*)
Append a string to a filename while keeping the extension.

gallop.util.**clip**(*v*, *range*)
Clip value v to the given numeric range

gallop.util.**doc_inherit**(*cls*)
Simple decorator to apply to other decorator definitions to ensure the function signature is not lost to sphinx when generating api docs for a decorated function.

gallop.util.**enum**(**args*)
An Enumeration type

gallop.util.**filter_columns**(*src_fname*, *cols*, *dest_fname=None*, *sep=' '*)
From *src_fname* remove the columns not listed in *cols*, numbering starts at one. If *dest_fname* is None, a temporary file is created and its name returned. *sep* is the column separator to use.

gallop.util.**find_ge**(*a*, *x*)
Find the index of the leftmost item greater than or equal to x

gallop.util.**flatten**(*l*)
Flatten all nested lists

gallop.util.**get_logger**(*obj=None*, *name=None*)
Get the logger for the given object. If obj is specified the class name is used as the logger name, else the passed name is used. If neither are specified the default gallop name is used.

gallop.util.**get_username**()

gallop.util.**hostname**()
Return the hostname

gallop.util.**init_logging**(*output_dir=None*, *filename=None*)
Initialize the python logging framework

Parameters

- **output_dir** – where to store the log file
- **filename** – name of the log file

gallop.util.**label_to_bool**(*val*)
Convert a class label into a boolean value.

gallop.util.**long_list_repr**(*l*, *length=10*)
For a list longer than length, return a shortened string representation showing only length elements. :param l: the list :param length: max. number of elements to show

gallop.util.**mkdir_p**(*path*)
Emulate mkdir -p shell command

gallop.util.**multiproc_map**(*func*, *data*, *nprocs=2*, *log=None*)
Convenient wrapper around map_async from `multiprocessing`.

Parameters

- **func** – function to map
- **data** – the data the function is applied to
- **nprocs** – the number of processors to use
- **log** – optional logger object

`gallop.util.`**`pickle_to_file`**(*obj*, *fname=None*)
Pickle the given object to file. If no filename is given a temp file is used and its path returned.

`gallop.util.`**`randstring`**(*n=4*)
Return a random string of length n

`gallop.util.`**`round_to_element`**(*v*, *seq*)
Round number v to the nearest element of seq

`gallop.util.`**`run_process`**(*executable*, *args*, *output_dir=None*, *cwd=None*, *logger=None*)
Fork a new process, all output is saved to an output file.

Parameters

- **executable** – path to the executable
- **args** – list of arguments
- **output_dir** – where to write the console output to (defaults to the current working directory)
- **cwd** – working directory for the executable
- **logger** – logging object to use for reporting progress

`gallop.util.`**`svmlight_sparse_to_timble_sparse`**(*infile*, *outfile*)
Convert a dataset from svmlight sparse format to Timble sparse format :param infile: input file :param outfile: output file

`gallop.util.`**`timestamp`**(*n=0*)
Generate a timestamp string with optional random suffix of length n

`gallop.util.`**`write_to_file`**(*s*, *fname=None*)
Dump the given string to file. If no filename is given a temp file is used and its path returned.

Subpackages

learner Package

`base` Module Base class for all learners and other useful utility functions.

class `gallop.learner.base.`**`ContinuousHyperparam`**(*name*, *range*, ***kwargs*)
Bases: `gallop.learner.base.Hyperparam`

Represents a hyperparameter that can take on any value in a continuous numeric range.

Parameters

- **name** – hyperparameter name
- **range** – [min,max] tuple
- ****kwargs** – passed on to the base class

`random_value`()
Return a random valid hyperparameter value.

class `gallop.learner.base.`**`DiscreteHyperparam`**(*name*, *choices*, ***kwargs*)

Bases: `gallop.learner.base.Hyperparam`

Represents a hyperparameter that can only take on a small, finite number of discrete values (integers, strings, ...).

Parameters

- **name** – hyperparameter name
- **choices** – allowed values
- ****kwargs** – passed on to the base class

`random_value`()

Return a random valid hyperparameter value.

class `gallop.learner.base.`**`Hyperparam`**(*name*, *value=None*, *description=''*)

Bases: `object`

Encapsulates a hyperparameter of a learner.

Parameters

- **name** – name of the parameter
- **value** – value
- **description** – description

`random_value`()

Return a random valid hyperparameter value.

class `gallop.learner.base.`**`Learner`**(*output_dir=None*, *overrides={}*)

Bases: `object`

Abstract base class for a particular learning algorithm.

Parameters

- **output_dir** – directory where any output should be written to (a learner subdirectory will be created)
- **overrides** – a dict of parameter name -> value mappings in order to manually force a particular value of a hyperparameter

`add_hyperparam`(*p*)

Add a `Hyperparam` object to the list of hyperparameters for this learner.

Parameters **p** – the hyperparameter object to add

`apply_constraints`(*ind*)

Apply any constraints on the individual, e.g. conflicts between hyperparameters. Returns the updated individual. Default is to do nothing.

Parameters **ind** – a dict of (name:value)

`get_hyperparam`(*name*)

Return the `Hyperparam` object with name *name*

Parameters **name** – the name of the hyperparameter

`get_hyperparam_names`()

Return a list of hyperparameter names

`get_hyperparams`()

Return a dict mapping the hyperparameter name to its `Hyperparam` instance

`get_random_hyperparams`(*enabled_features=None*)
Return a (name,value) dict with a random (but valid) value for every hyperparameter.

`make_feature_compatible`(*hp*, *enabled_features*)
Ensure the passed hyperparameter dict is compatible with the given list of enabled features and any manual overrides.

Called by `get_random_hyperparams()`, `mutate_hyperparams()`, and `recombine_hyperparams()`.

Parameters

- **hp** – a dictionary mapping the hyperparameter name onto its value
- **enabled_features** – a list of integers in [1,#features]

`mutate_hyperparams`(*hp*, *enabled_features=None*)
Mutate the given hyperparameters randomly but ensuring they are still valid and respect the enabled feature list.

Parameters

- **hp** – a dictionary mapping the hyperparameter name onto its value
- **enabled_features** – a list of integers in [1,#features]

`predict`(*filename*)
Predict the classes for the data in *filename*.

Returns a tuple of two lists, the first with the true data (or None if not available), the second with the predicted data.

`recombine_hyperparams`(*hp1*, *hp2*, *enabled_features_child1=None*, *enabled_features_child2=None*)
Recombine the two hyperparameter sets into two new, valid, children. The base implementation uses single point crossover.

Parameters

- **hp1** – a dictionary mapping the hyperparameter name onto its value
- **hp2** – a dictionary mapping the hyperparameter name onto its value
- **enabled_features_child1** – a list of integers in [1,#features]
- **enabled_features_child2** – a list of integers in [1,#features]

`set_hyperparams`(*new_hp*)
Set the hyperparameter values of the learner.

Parameters **new_hp** – a dictionary mapping a hyperparameter name onto a value

`train`(*filename*, *num_features=None*)
Train the learner on the data *filename*.

Parameters

- **filename** – path to the training file
- **num_features** – how many features in the file, needed only for sparse dataformats

`train_and_predict`(*train_fname*, *predict_fname*, *num_features=None*)
Perform the training and prediction in one single step. This can be more efficient and this method will always be called where possible. The default implementation simply calls `train()` and `predict()`.

`crflearner` Module Functionality for interfacing with CRF++.

class `gallop.learner.crflearner.`**`Algorithm`**(*algorithms=['O', 'M', 'J', 'D', 'C', 'N', 'L', 'DC'], dist_metrics=[]*)

Bases: `gallop.learner.base.Hyperparam`

A custom hyperparameter, representing the -a CRF++ flag.

static **`make_feature_compatible`**(*val*, *enabled_features*)
Remove from the -m value any references to features that do not appear in the enabled_features list.

`random_value`()
Return a random valid hyperparameter value.

class `gallop.learner.crflearner.`**`CrfLearner`**(*num_features*, *num_instances*, *crf_learn_binary='/usr/local/bin/crf_learn'*, *crf_test_binary='/usr/local/bin/crf_test'*, *verbosity='-v0'*, *file_format='sentence'*, *templates=''*, *ncores='1'*, **args*, ***kwargs*)

Bases: `gallop.learner.base.Learner`

Wrapper class for the crf binary.

Parameters

- **crf_learn_binary** – path to the crf_learn executable
- **crf_test_binary** – path to the crf_test executable
- **num_features** – total number of features in the data
- **num_instances** – how many instances to expect (needed for -b)
- **verbosity** – passed directly to the binary (see timbl manual)
- ***args** – passed to base class
- ****kwargs** – passed to base class

`make_feature_compatible`(*hp*, *enabled_features*)
Ensure the given hyperparameter set does not refer to missing features. For timbl this means looking at -q,-m

`predict_old`(*filename*, *model*)
Predict the classes for the data in *filename* using CRF++.

`train_and_predict`(*train_fname*, *predict_fname*, *num_features=None*)

`train_and_predict_old`(*train_fname*, *predict_fname*, *num_features=None*)
Perform training and prediction over the data in the given filenames.

`train_old`(*filename*, *num_features=None*)
Train a CRF-model on the data in *filename*.

class `gallop.learner.crflearner.`**`VotingType`**(*types=['Z', 'ID', 'IL', 'ED'], alpha_range=(0.01, 5), beta_range=(0.01, 3)*)

Bases: `gallop.learner.base.Hyperparam`

A custom hyperparameter representing the -d timbl flag.

`random_value`()
Return a random valid hyperparameter value.

`gallop.learner.crflearner.`**`random`**() → x in the interval [0, 1).

`features` Module Utility classes and functions to handle feature selection.

class `gallop.learner.features.`**`FeatureSets`**(**args*, ***kwargs*)

Bases: `object`

Container for one or more feature sets, those features that the user wants to take into account in the optimization. Note that features are numbered starting from 1.

Example invocation:

```
>>> feats = FeatureSets(1,2,5,[6,9,10],range(11:15),32)
```

Parameters

- ***args** – an arbitrary long list of integers or lists of ingeters (representing feature groups)
- **num_features** – the total number of features

`deep_iter`()

Return an iterator over the flat list of features.

`flat_list`(*sorted=True*)

Return a flat list of all the features in the various groups.

`validate`()

Check the feature sets are valid (no overlap, etc).

`timbllearner` Module

optimizer Package

`distgaoptimizer` Module The same as `GAOptimizer` except that the fitness function is evaluated on a Torque managed cluster.

class `gallop.optimizer.distgaoptimizer.`**`DistGAOptimizer`**(**args*, ***kwargs*)

Bases: `gallop.optimizer.gaoptimizer.GAOptimizer`

Distributed version of `GAOptimizer`. Fitness evaluations are submitted to a Torque managed cluster, optionally connecting to a submit node over ssh first. Takes the same arguments as `GAOptimizer` and requires a number of extra keyword arguments.

Parameters

- **username** – SSH username
- **password** – SSH password
- **submit_node** – job submission hostname
- **remote_dir** – remote working directory (defaults to the output dir the submit_node is localhost
- **walltime** – wall clock time required for every job (e.g., 2:00:00)
- **mem** – maximum memory for each job (e.g., 4gb)
- **ppn** – number of processors required per node (e.g., 3)
- **poll_interval** – interval in seconds for polling the job status (e.g., 30)

Note that if the submit_node is the same node as gallop is running on, the ssh connection will simply loop back.

get_job_status (*jid*)
Return the status of the given job id as returned by qstat.

gallop.optimizer.distgaoptimizer.**gallop_job** (*datafile*, *popfile*, *idx*, *resdir*)
The actual function that is run as a cluster job. Reads in the pickled data files and individuals. Calls the fitness function on the individual identified by idx. Writes the updated individual to file.

gallop.optimizer.distgaoptimizer.**main** ()

gallop.optimizer.distgaoptimizer.**random** () → x in the interval [0, 1).

gaoptimizer Module The core gallop logic. An implementation of a genetic algorithm to perform learner hyperparameter optimization and feature selection.

class gallop.optimizer.gaoptimizer.**FitnessMax** (*values=()*)
Bases: deap.base.Fitness

weights = (1,)

class gallop.optimizer.gaoptimizer.**FitnessMin** (*values=()*)
Bases: deap.base.Fitness

weights = (-1,)

class gallop.optimizer.gaoptimizer.**GAOptimizer** (*learner*, *measure*, *feature_sets=None*, *maximize=True*, *nprocs=1*, *output_dir=None*, *std_tol=1.0000000000000001e-05*, *fit_tol=1.0000000000000001e-05*, *fit_window=5*)
Bases: object

Use an elitist GA to optimize the hyperparameters of the given learner and perform feature selection. All output will be written to the given output directory and a *trace* subdirectory will be created where the population at the end of each generation will be saved.

Parameters

- **learner** – an instance of a subclass of gallop.learner.base.Learner
- **measure** – an instance of a subclass of gallop.measure.Measure
- **feature_sets** – an instance of gallop.learner.features.FeatureSets. If set to None no feature selection is performed, only hyperparameter optimization.
- **maximize** – maximize the objective function if True, else minimize
- **nprocs** – numer of processes to use when evalutating the fitness function
- **output_dir** – where all output should be written to

Termination criteria:

Variables

- **std_tol** – minimum requried standard deviation of the population fitness
- **fit_tol** – minimum change in fitness needed over *fit_window* generations
- **fit_window** – generation interval for *fit_tol*

GA parameters:

Variables

- **xover_prob** – the crossover probability

- **mutation_prob** – the mutation probability
- **elite_perc** – percentage of the population to keep as elite

`run` (*popsize=50*, *generations=40*, *initial_pop=None*, *history=None*)
Run the GA.

Parameters

- **popsize** – population size to use
- **generations** – number of generations to run for
- **initial_pop** – starting population to use (can be taken from a previous run by loading in one of the pickle files in the trace directory). If less than popsize individuals are passed in, random individuals will be added to make up the difference.
- **history** – (optional) DEAP History object, from a previous optimization run. This serves as a cache for previously evaluated individuals, to avoid recomputing the fitness.

Returns a result object with the best individual, its fitness, the rest of the population and some statistics

class `gallop.optimizer.gaoptimizer.`**`Individual`** (*hyperparams*, *features*, *fitness*)
Bases: `object`

Represents a single individual in the GA: A combination of a set of hyperparameters and a set of feature groups.

Parameters

- **hyperparams** – a (name,value) dict of hyperparameters
- **features** – a (feature group tuple,enabled flag) dict of feature groups
- **fitness** – a subclass of `deap.base.Fitness`

static **`create_individual`** (*learner*, *feature_sets*, *fitness*)
Factory method for creating random individual instances.

`enabled_features` ()

`features`

`hyperparams`

`labels` ()

`gallop.optimizer.gaoptimizer.`**`fitfun_single`** (*learner*, *feature_sets*, *measure*, *maximize*, *ind*)
Evaluate the fitness of a single individual. Not a method since that prevents it from being easily pickled and used with multiprocessing.

Typically should not be called by any third party code.

`gallop.optimizer.gaoptimizer.`**`hyperp_feature_mutation`** (*ind*, *learner=None*)
Mutation function for individuals. Performs mutation on the selected features and delegates to the learner for mutating the hyperparameters.

`gallop.optimizer.gaoptimizer.`**`hyperp_feature_xover`** (*ind1*, *ind2*, *learner=None*)
Crossover function for individuals. Performs recombination on the selected features and delegates to the learner for recombining the hyperparameters.

`gallop.optimizer.gaoptimizer.`**`random`**`()` → x in the interval [0, 1).

PYTHON MODULE INDEX