Using Distributed Machine Learning to Predict Arterial Blood Pressure

by

Ijeoma Emeagwali

B.S., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014

Author .... Signature redacted ....
Department of Electrical Engineering and Computer Science
January 31, 2014

Certified by .... Signature redacted
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Certified by .... Signature redacted ..
Erik Hemberg
PostDoctorial Associate
Thesis Supervisor

Accepted by .... Signature redacted ....
Albert R. Meyer
Chairman, Department Committee on Graduate Theses

Using Distributed Machine Learning to Predict Arterial Blood Pressure

by

Ijeoma Emeagwali

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2014, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science

Abstract

This thesis describes how to build a flow for machine learning on large volumes of data. The end result is EC-Flow, an end to end tool for using the EC-Star distributed machine learning system. The current problem is that analysing datasets on the order of hundreds of gigabytes requires overcoming many engineering challenges apart from the theory and algorithms used in performing the machine learning and analysing the results. EC-Star is a software package that can be used to perform such learning and analysis in a highly distributed fashion. However, there are many complexities to running very large datasets through such a system that increase its difficulty of use because the user is still exposed to the low level engineering challenges inherent to manipulating big data and configuring distributed systems. EC-Flow attempts to abstract a way these difficulties, providing users with a simple interface for each step in the machine learning pipepline.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

Thesis Supervisor: Erik Hemberg
Title: PostDoctorial Associate

Acknowledgments

I would like to acknowledge Erik Hemberg, Kalyan Veeramachaneni, and the other members of CSAIL's ALFA group that made who's work made this thesis possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

When attempting to perform it on large enough datasets, the process of using machine learning to analyse data can begin to present many difficulties apart from the theories and algorithms of the machine learning itself. Algorithms that may be easy to run on a small dataset that can be stored and processed on one machine, may not scale graciously to run datasets on a terabyte scale. Even if the machine learning technique is designed to scale well from an algorithmic perspective, the engineering challenges of storing, processing and analysing big data still present a challenge that will influence the implementation of such an algorithm in practice. This project focuses on the use of EC-Star, a software for running Evolutionary Algorithms, an area of machine learning that lends itself naturally to running in a highly distributed fashion since each candidate solution can be evaluated in parallel and asynchronously. With the EC-Star platform, users can perform highly distributed runs of evolutionary algorithms on large amounts of data. However, the platform does not abstract all of the details of manipulating the data and moving through the machine learning process away from the user. At large enough scale, configuring such a distributed system, and preparing a dataset to run through it becomes an engineering challenge itself. With this project, EC-Flow attempts to tackle this issue, presenting an end to end solution beginning with a raw data set, and moving through the process to analysing the end results.

1.2 Challenges with Big Data Machine Learning

Even with a platform like EC-Star to take care of the implementation of the machine learning algorithm in a distributed fashion, there are still many steps in the process that become tedious and error prone if the users are forced to perform them every time they use the system. For example, software like EC-Star accepts data in a standardized format meaning before a dataset can be run in the system it must first be manipulated and transformed into an acceptable format. At large scale this can be a timely and error prone process due to the fact that the usual scenarios don't involve manipulating millions of files totaling hundreds of gigabytes of data. Furthermore a system like EC-Star must be tweaked and configured to the specific experiment and dataset being analysed.

EC-Flow attempts to tackle these Big Data challenges and hide these details from the user making the entire pipeline from data formatting to starting the distributed algorithm appear as a black box. By inputting the high level parameters and experimental settings at the beginning of the pipeline, EC-Flow will automatically prepare the data and configure EC-Star to the user's specifications making the process much more seamless and abstracting away the last layer of big data engineering tasks away from the end user.

Chapter 2

The EC-Star System

2.1 Design

2.1.1 Design Motivations for EC-Star

One of the significant challenges in Evolutionary Algorithms as in any machine learning algorithm is the engineering challenge of being able to scale the actual running of the algorithm to larger and larger datasets. When datasets are too large and the field of solution possibilities to vast to fit on a single computer distributed systems must be utilized to make such expansive analysis practical. The EC-Star system [1]is a distributed software system that can allow genetic programming experiments to be run on the scale of a million globally distributed nodes known as "Evolutionary Engines" or clients. The EC-Star platform is a distributed Evolutionary Algorithms framework utilizing commercial volunteer resources. These nodes can be independently added and removed while the software is running with easy integration into a continuously running evolutionary algorithm. The EC-Star platform distributes the computations on pool servers using a hub and spoke topology(Figure 2-1). An Evolution Coordinator(also known as pool servers) serves as the hub with an Evolutionary Engine as each spoke. The coordinator sends the high performing partially evaluated candidate solutions for further fitness evaluations, mixing and evolution to the Evolutionary Engines.

2.1.2 Representation in EC-Star

Each Evolutionary Engine in EC-Star hosts an independent evolutionary algorithm with a fixed population size during the client's idle cycles. They request fitness cases in the form of data packages from the fitness servers, evaluate and breed them and eventually dispatch them as migrants to the Evolutionary Coordinator. The solutions are represented as a set of conjunctive rules(Figure 2-2). Each rule has a variable length conjunctive set of conditions and associated actions representing a class in the given classification problem. Each condition can also have a complement operator

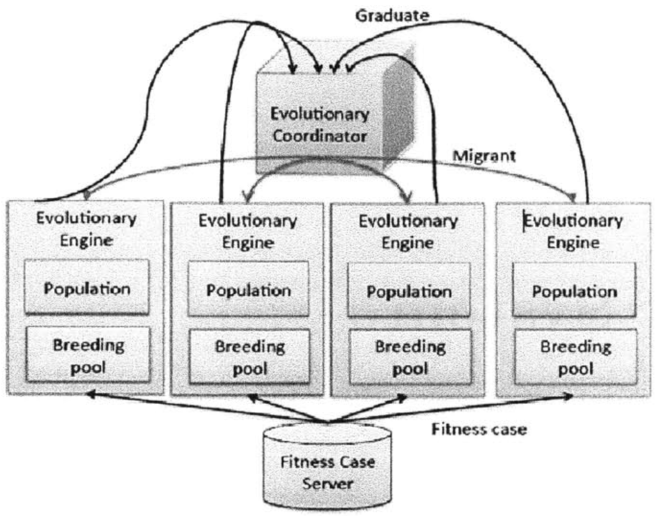


Figure 2-1: EC-Star Hub and Spoke Model[1]. Pool servers serve as the hubs in this model and communicate with the clients that act as the spokes.

```
<rules> ::= <rule> | <rule> <rules>
<rule> ::= <conditions> => <action>
<conditions> ::= <condition> | <condition> & <conditions>
<action> ::= prediction label
<condition> ::= <predicate> | !<condition>  | <condition> [lag]
<predicate> ::= truth value on a feature indicator
```

Figure 2-2: BNF grammer showing how a solution in EC-Star is represented as a set of conjunctive rules

which negates the truth value, and a lag which refers to past values of the attribute. The condition checks if an attribute value currently or in the past given the lag is greater then a threshold. The thresholds are discretized values for each feature.

2.2 Architecture

2.2.1 Clients in EC-Star

The clients in EC-Star are computers on a volunteer compute network[2] using their idle cycles to act as the "Evolutionary Engines". Given the nature of using the idle cycles of volunteer nodes, no guarantees can be assumed about how quickly work will be completed by the nodes or if it will be completed at all. Furthermore, to limit the

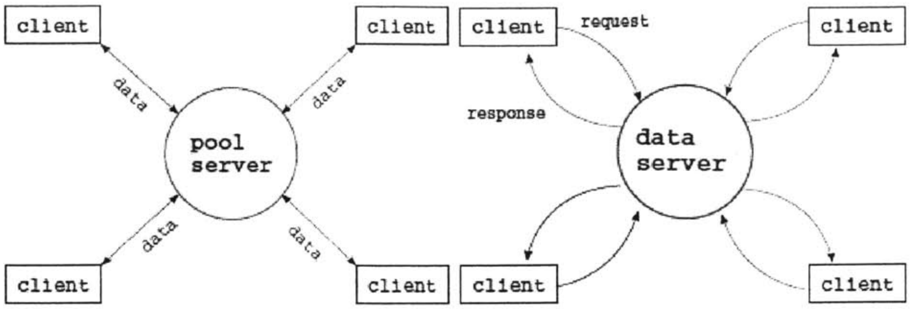


Figure 2-3: Pool servers act as the hubs in EC-Star and receiving solutions from the clients. The data server responds by sending data packages to the clients upon request.

footprint of running the platform the clients are restricted in the amount of memory they are allowed to use for their computations. They are however able to write state to disk such that after a program is shut down it can use its state file in order to resume where it left off. In addition clients do not communicate with each other to maintain privacy. Instead, the clients communication is the dedicated pool servers and data servers.

2.2.2 Servers in EC-Star

Dedicated resources that can communicate with the volunteer compute resources act as the servers, the hubs in the systems(Figure 2-3). Pool Servers handle communication with clients using a database for persistence and scalability. Data Servers serving the data requested by clients. The data servers return random packages to the clients that request them. The servers therefore act as the hubs in this hub and spoke model. The dedicated servers run continuously as clients come on and off line due to the nature of them being volunteer compute resources.

By taking advantage of the massive scale of the volunteer compute resources the EC-Star platform will allow the use of evolutionary algorithms to solve problems that would have previously been intractable. The use of idle cycles in volunteer nodes also makes the system more cost effective compared to owning the equivalent hardware or using similar cloud services.

Chapter 3

EC-Flow

Setting up and running EC-Star on a given data set involves many steps from formatting the data to configuring he clients and servers with the code to run the software and harvest data. The following section presents EC-Flow, an end to end system to set up runs of EC-Star beginning with the raw data set.

3.1 EC-Flow Overview

Running a machine learning algorithm with EC-Star can be broken into 4 phases all of which can be automated to some extent through the use of EC-Flow. First, data packages must be created transforming the raw data set the user is working with into a standard format EC-Star can handle. Next the code and configuration of EC-Star must be customized to the specific dataset and type of experiment being run. The computers playing the different roles (clients, servers, data hosts) must be configured so that the experiment can actually be run (Figure 3-1).

As soon as some initial solutions have been found the solutions can be tested on another subset of the data. Scripts for getting the accuracy of the solutions can be used to generate confusion matrices and the overall accuracy percentages.

3.1.1 Data Package Creation

A given dataset of raw csv files must first be transformed into a data package format that EC-Star expects. Each row is an examplar and each column is a feature. The last column is a label. At a high level, this includes limiting the size of the data packages to approximately 5MB each (compressed), changing any formatting or values in the data packages that EC-Star might not accept, as well as adding a header to each data package. The process of going from raw csv files to EC-Star data packages is potentially the longest step in the setup process. This process may take only minutes for datasets on the order of 1GB, to many hours to perform on the scale of 100GB if not done in parallel. In general not only does the total amount of raw data affect the data package creation time, but also the number of data packages one is creating.

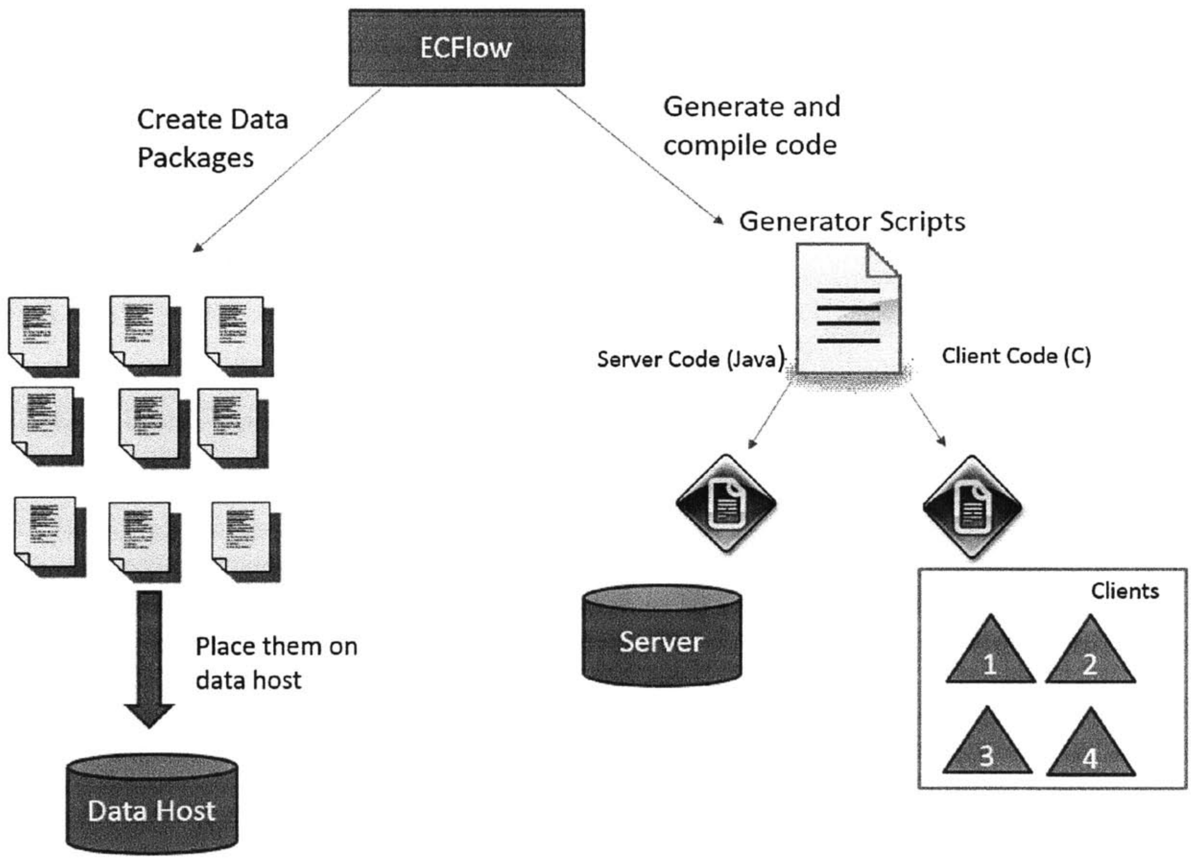


Figure 3-1: EC-Flow Overview. EC-Flow is used both to create the data packages from raw csv files, as well as set up and execute runs of EC-Star

The smaller the size of the data package, the more packages will have to be created a given amount of raw starting data. On many computer setups, the increase in the number of files can drastically slow down the time it takes to create and process all of them. In addition, the large number of files can slow down the process even if the packages are created in parallel, if in the end they are stored on the same disk causing i.o. contention.

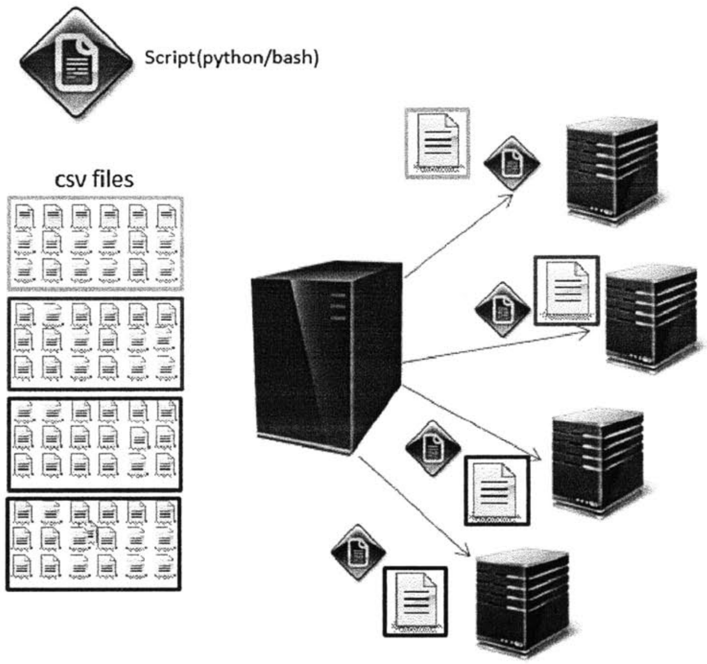
3.1.2 Parallel Data Processing on Open Stack

For creation of large amounts of data packages you may wish to utilize the multiple nodes on the Open Stack platform to simultaneously do processing on the initial csv files. The `sleipnir` package (Figure 3-2) will allow you to, given a source directory and a python or bash script, distribute the processing of data files across nodes on Open Stack. The processing of creating data packages from raw csv files can in general be done independently. If multiple compute nodes are available it is possible to split the work of creating data packages. A script designed to work with `sleipnir` must take in at least an input and output directory as arguments, additional arguments are optional. When `sleipnir` is called to run the script on the directory of data, it will automatically partition the data based on the amount of nodes available. It will then transfer the data and script to each node for processing. Since the only requirements are for the script to take an input and output directory so that `sleipnir` can point it at its partition of files to work with, converting most steps in the data processing pipeline to `sleipnir` scripts is relatively straightforward. See Appendix C for an example of performing the file chunking step (see section 3.2.1) in parallel on `sleipnir`.

3.1.3 EC-Star Configuration

Beyond the data packages to be used in the experiment, every run of EC-Star has a multitude of settings and configurations that must be adjusted depending on the type of data packages used and the specific parameters of the experiment. As a result, much of the client and server code must be generated and remade each time new data is used. To avoid the tedious and likely error prone method of manually copying and pasting numbers in the client and server code whenever changes are made, EC-Flow provides methods through which the code an automatically be re-generated and made whenever necessary. Settings passed in at a high level, either through the command line or the EC-Flow config file, will automatically be integrated into the client and server code during the code generation phase so that users can expect that the code will be consistent across the client and server and that values will be changed in all the proper places. In addition, after generating the code EC-Flow will also compile the code, creating executable and configuration files for the client and jar files for the server, as well as run unit tests.

Figure 3-2: Sleipnir Overview. Sleipnir is used to process large amounts of data files in parallel. The system will divide up the data among the available nodes and run the given script on each set of data on each node.



3.1.4 Running Experiments

Once the data packages have been created and the EC-Star code has been configured, the final step is to run the experiments. EC-Star can be configured to run in a number of ways from running the server and clients locally, to using multiple machines to serve as clients, data hosts and machines dedicated to analysing the results. While determining which machines play which roles in a given experiment is determined by the user, setting up a given machine for a its desired role is a process EC-Flow can automate. Depending on the role a machine is playing, certain files must be generated, data placed in certain directories, and network settings must be adjusted all of which can be done through various EC-Flow commands. Once an experiment has been started, EC-Flow and scripts it has access to can be used to analyse the results.

3.2 Data Package Creation

EC-Flow (`ecflow.py`) contains several useful commands to individually perform each step in the process of going from raw csv files to data packages to be used by EC-Star. Steps can be performed individually or all at once with parameters being passed in from the command line or through a config file.

Figure 3-3: Data package creation process.

3.2.1 File Chunking

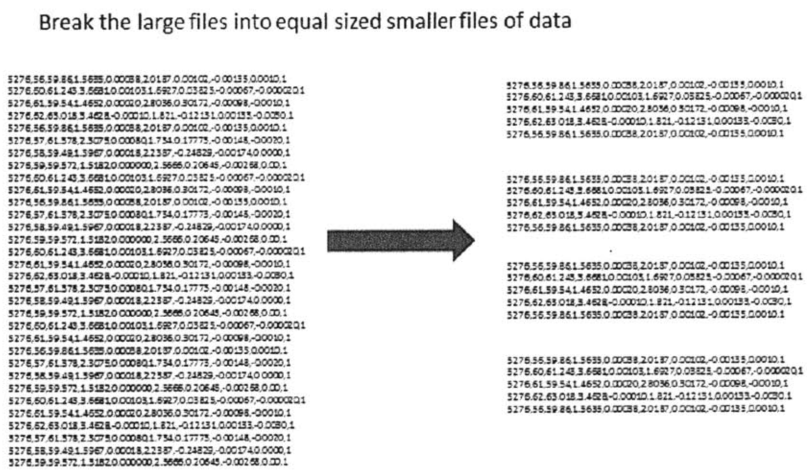


Figure 3-4: File chunking.

```
$ python ecflow.py -d src_directory --chunk --rows N
```

The above command chunks every file in `src_dir` into multiple files of `N` rows. Files leftover after chunking that have less than `N` rows will be deleted if the `exact` setting in EC-Flow is set to `true`. This step will roughly determine the number of data packages that will be produced as a final output(some data packages may be deleted during data cleaning). Choosing a smaller value of `N` will result in more data packages and cause the rest of this pipeline to run slower due to the amount of files even given the same total storage size. Note that the chunking scripts will specifically look for files ending in `.csv`.

3.2.2 Data Cleaning

In this step we discard packages that contain bad values (e.g. `NaN`). In some sources of data values such as `NaN` that EC-Star cannot process will be present. We therefore must throw out data packages containing such values.

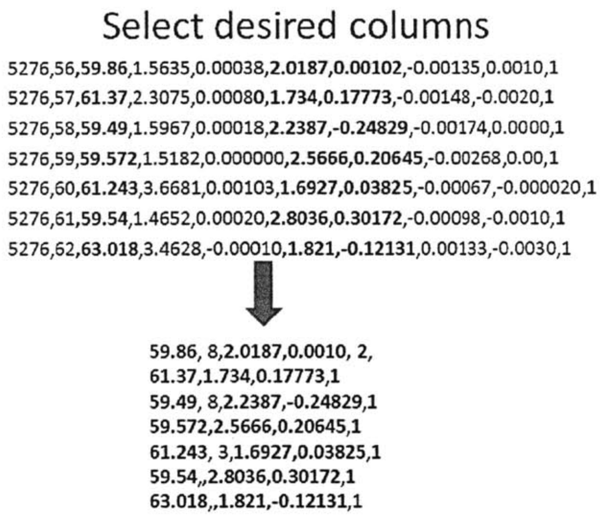


Figure 3-5: Feature selection. The highlighted features in the original file plus the label(the final column) are selected for the final data package.

```
$ python ecflow.py -d src_directory --clean
```

Running this command will clean the values of the `src_directory` of `NaN`, deleting any packages that contain it.

Scientific notation is another data format that EC-Star is unable to handle. If the generated data contains scientific notation, these entries will have to be converted to decimal to be used by EC-Star. This will be done automatically during the feature selection step (section 3.2.3).

1.23759e4 `becomes` 12375.9

3.2.3 Feature Selection

Rather than taking all features of the data, you may wish to use only certain features of the data by deleting certain columns in the data set. Both of these tasks are accomplished simultaneously by using –select.

```
$ python ecflow.py -d src_directory --select -f [0,1,2,7]
```

This command will alter all data packages in the directory to only contain features 0,1,2 and 7. In addition all scientific notation will be converted to decimal. If you wish to simply remove all scientific notation but retain all features, you may either pass in a list of all features, or simply not use the `-f` flag at all, ECFlow will take all

```
5276,56,59.86,1.5635,0.00038,2.0187,0.102,-0.00135,0.00,1
5276,60,61.243,3.6681,0.00103,1.6927,0.025,-0.00067,-0.020,2
5276,61,59.54,1.4652,0.00020,2.8036,0.372,-0.00098,-0.10,1
5276,62,63.018,3.4628,-0.00010,1.821,-0.131,0.00133,-0.30,2
5276,56,59.86,1.5635,0.00038,2.0187,0.002,-0.00135,0.0010,1
5276,57,61.378,2.3075,0.00080,1.734,0.173,-0.00148,-0.20,0
5276,58,59.49,1.5967,0.00018,2.2387,-0.229,-0.00174,0.00,2
5276,59,59.572,1.5182,0.000000,2.5666,0.245,-0.00268,0.00,2
```

↓

```
5276,56,59.86,1.5635,0.00038,2.0187,0.102,-0.00135,0.00,2
5276,60,61.243,3.6681,0.00103,1.6927,0.025,-0.00067,-0.020,1
5276,61,59.54,1.4652,0.00020,2.8036,0.372,-0.00098,-0.10,0
5276,62,63.018,3.4628,-0.00010,1.821,-0.131,0.00133,-0.30,2
5276,56,59.86,1.5635,0.00038,2.0187,0.002,-0.00135,0.0010,2
```

Figure 3-6: Data package before and after a lead time adjustment of 3.

features by default. Note that if your data contains scientific notation, this step must be performed even if you intend to keep all columns in the dataset, because this step will also convert any instances in the data of scientific notation into decimal.

3.2.4 Adjusting Lead Time

For time series data you may wish to adjust the lead time in the data packages. In time series data, the rows are ordered because each is features from a certain time point. Adjusting the lead time by `N` will move the label in each line `N` rows higher resulting in deletion of the final `N` rows of the original data package(Figure 3-6). To do this as a final step before adding the header, use the command

```
$ python ecflow.py -d src_directory --adjust --lead N
```

where `N` is the amount of lead time you wish to include.

3.2.5 Adding Headers

The final step in data package creation is to add a header to each file. The headers are required by EC-Star to process the data packages. This can be done with the following command

```
$ python ecflow.py -d src_directory --header
```

An example header is shown in figure 3-7. The header contains the file name, a unique id for the file, the number of data points in the file, and a list of column names.

```
example_file.gdp
samples 1
sampleIds 167
events 1000
fields 17
sampleId, eventId,v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12,v13,v14,label
```

Figure 3-7: Sample EC-Star data package header.

3.2.6 Data Package Creation Summary

If you wish to perform all of the above steps at once simply run

```
$ python ecflow.py -d src_directory --all
```

This will chunk, clean, select features and finally add headers to the data packages. Note that EC-Flow will use default values for all parameters not passed in via the command line or a configuration file. (See Appendix C Table A.2 for full details)

3.2.7 Creating Folds for Cross Validation

For the purpose of cross validation EC-Flow can, given a percentage of data to use for testing and training, generate multiple random folds of the data.

```
$ python ecflow.py -d src_directory/ --split
```

The above command will create a file called `splits.csv`. This file will for each fold list which data packages belong in the test set (the training data packages can be found by taking the compliment of this set). EC-Flow can than parse this file moving test and training files to desired directories with the command

Moving files

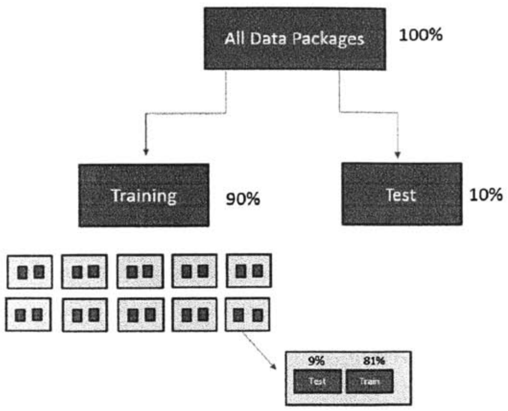
```
$ python ecflow.py -d src_directory/ --move --move_split N --train_dest
train/ --test_dest test/
```

In the above command `N` is the fold number, `train/` and `test/` will be the directories the data packages will be copied into.

10-Fold Cross Validation

The `--xfold` flag will allow you to in one step make two splits files. `90-10_splits.csv` will be a splits file splitting the data directory with 90% training. `splits.csv` will

Figure 3-8: 10-fold split. The original set of data is broken into a 90/10 split. The 90% split is then further divided into ten 90/10 splits.



be a splits file that takes 9 folds (90% of the first split) and re-splits them 10 ways (equivalent to splitting with 90% training, see Figure 3-8). To do this run

```
$ python ecflow.py -d src_directory/ --xfold
```

Note that this will only create the two splits files but will not actually move any data packages. To do this, first move 90% of the files into one directory and 10% into another using the `move` command as in the previous section. Use the flag `--split_file` to pass in `90-10_splits.csv`. Next, specifying the directory with 90% of the files as the target directory with `-d` call move again `move_split N` to move the N^{th} split into your specified train and test directories. It is not necessary to pass in `splits.csv` since it will be chosen by default.

3.3 Data Packages as a Service (DPaaS)

Even when making data packages in parallel, on the scale of hundreds of gigabytes of raw data, the creation of data packages can lead to hours of overhead in setting up EC-Star runs. Since it may be impractical to repackage hundreds of gigabytes of data every time a new type of data package is desired, generating the data packages on the fly as a service may be a more logical approach. Depending on the architecture for processing and storing the data packages, the i.o. contention on the disk where the packages are being stored can limit the rate at which data packages can be created, even if computers are available to perform some parts of the process in parallel. In general, the larger amount of data, and the more data packages you expect to end with after the data as been processed the less amount of parallelization can be achieved(see Table A.3 for empirical data on data package creation using OpenStack virtual machines). The program `dpaas.py` (data packages as a service) uses a subset of EC-Flow to create the requested amount of data packages from a directory of csv

Figure 3-9: Sample tomcat log on data server

```
Nov 17, 2013 10:16:27 AM com.gf.eacore.clientserver.DataServer fetchDataPackageFromFolderRandomly
INFO: Loaded data package webapps/dataPackages/syn_cardiotocography_17_101_482.gdp
Nov 17, 2013 10:16:27 AM com.gf.eacore.clientserver.DataServer fetchDataPackageFromFolderRandomly
INFO: Loaded data package webapps/dataPackages/syn_cardiotocography_17_10_549.gdp
Nov 17, 2013 10:16:27 AM com.gf.eacore.clientserver.DataServer fetchDataPackageFromFolderRandomly
INFO: Loaded data package webapps/dataPackages/syn_cardiotocography_17_101_246.gdp
Nov 17, 2013 10:16:27 AM com.gf.eacore.clientserver.DataServer fetchDataPackageFromFolderRandomly
INFO: Loaded data package webapps/dataPackages/syn_cardiotocography_17_10_650.gdp
Nov 17, 2013 12:16:28 PM com.gf.eacore.clientserver.DataServer fetchDataPackageFromFolderRandomly
INFO: Loaded data package webapps/dataPackages/syn_cardiotocography_17_101_753.gdp
```

files. Passing in a config file specifying the data package parameters, `dpaas` can be used to through a simple interface randomly create a limited amount of data packages from a much larger directory of raw data.

3.3.1 When to use DPaaS

In the case of creating data packages for EC-Star, one can examine the EC-Star logs to determine approximately the rate at which data packages are being requested to learn if it is necessary to preprocess all of the data or if using DPaaS may be more convenient and practical.

Examining the timestamps in the logs we can see the rate at which data packages should be supplied. Rather than have a single static directory from which data packages are randomly drawn we can use two directories. One directory to hold the data packages from which clients randomly draw from and another in which we can store data packages as they are continuously created. Data packages can be randomly switched into the pool of data packages the client is drawing from to keep up with the rate at which clients are requesting packages(Figure 3-10). If the requested data package configuration changes (e.g. change in size of data packages) the nodes creating the packages can be updated to begin producing the new configuration. Clients can begin to collect these new packages without waiting for the entirety of the original dataset to be reprocessed, something that could take many hours depending on the size of the dataset.

3.3.2 DPaaS test example

To test DPaaS on a local system, we can simulate writing and reading to and from a test log, calling on DPaaS to create more data packages when some threshold number of packages have been created. To begin the data package creation demo simply `cd` in to the `demo` directory and run demo.sh

```
$ bash demo.sh
```

This short script first calls `writelog.py`, which begins to continuously write to the

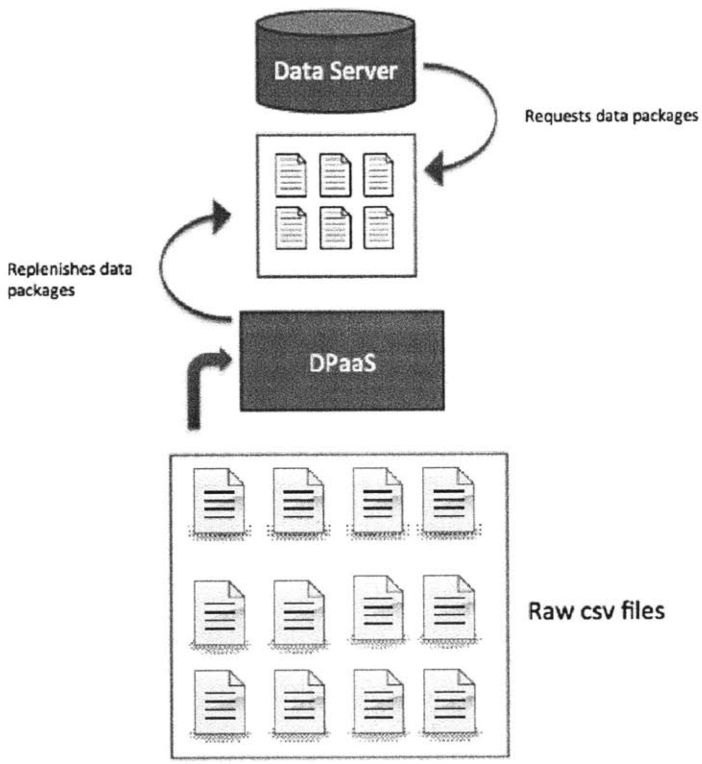


Figure 3-10: When using data packages as a service, the data host draws from a pre-made directory of data packages. As the pool of data packages runs out, DPaaS creates more from the pool of raw data files.

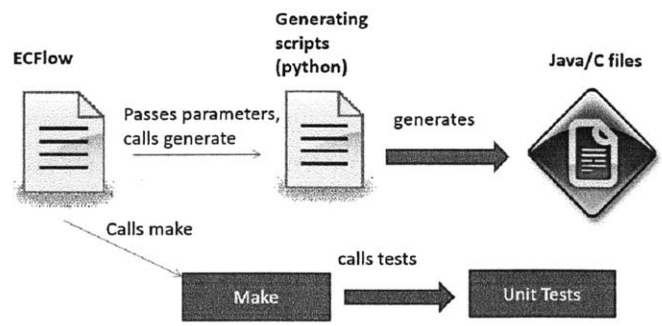


Figure 3-11: Code preparation overview.

file `testlog.log` in the background. It then calls `readlog.sh` which will continuously check this log, creating more data packages (according to the configuration file in `src/dpconfig.cfg`) after some threshold number of files have been created. `readlog.sh` takes three arguments. The first is the location of the log to be read. The second is the number of new files requested in the log before new packages will be created.The final argument is the number of seconds to sleep in between checking the log file (the first argument) for updates. To adjust the parameters of the created data packages, make changes to `dpconfig.cfg`.

num_files Increasing the parameter `num_files` will increase the number of source csv files that will be turned into data packages.

num_rows This parameter determines how many rows will be in each data package. The lower this number, the more data packages will be created from the same number of source files. Note that the length of time it takes to create the packages depends mostly on the number of packages created, not the combined storage size of the packages.

random_seed The `random_seed` parameter is not used for demo purposes and changing it will have no effect.

3.4 Code Preparation

Some aspects of the source code for EC-Star must be generated and compiled in advance because values can depend on the specific dataset you are working with. Before the code can be generated you must first create the file `conditions.txt`. Since `conditions.txt` will depend on the data packages you must also let EC-Flow know which directory contains the data packages either through the command line or a config file.

```
$ python ecflow.py --conditions -d ProcessedData
```

Once conditions.txt has been creating code can be generated with the -g flag and made with --make(Figure 3-11).

```
$ python ecflow.py -g
```

During code generation a series of python scripts are called which will generate the C files and Java files that vary depending on the input data packages being used for the run, as well as other run specific values such as lag time.

Fitness The fitness function to be used is defined in this step and is placed in the generated C and Java code. This fitness function can be passed into EC-Flow (selected from a pre defined list) which is then written into the generated client and server code.

Feature Discretisation By default, EC-Flow will automatically generate feature discretisation values by looking at conditions.txt and selecting a uniform number of values between the min and max values of each feature. It will placed in a root directory file called buckets.txt which will later be read by the generating scripts. If you wish to supply your own buckets.txt pass in the flag --readbuckets to EC-Flow. This will tell EC-Flow not to generate its own buckets.txt allowing the user to pass one in without it being overwritten.

```
$ python ecflow.py --make
```

Running make uses the make file in the directory to compile the C code and run unit tests. The C code will then be copied into the client directory. Note that the generation and making of the code assumes the directory structure for the client and server code is already present for the generated files to be placed into. Making the server files results in the following jar files being placed in the target/ directory

```
class-server-jar-with-dependencies.jar
pool-server-jar-with-dependencies.jar
```

Making the C code will result in the executable file Bp_client_exec as well as the configuration file client.cfg being placed in client/src/resources.

3.5 Setup EC-Star run

Now that the data packages have been created and the EC-Star code has been generated and configured an actual EC-Star run can be started. The work to be done in this step will vary depending on the hardware setup you are attempting to proceed with. For a small scale local run little more configuration is needed and all that is needed is to start the clients and servers. For a distributed setup further configuration will be necessary. The next chapter walks through how to configure various types of EC-Star setups.

3.6 Analysing Results

Once solutions have been found they can be downloaded from the pool server and and tested against the test partition of the dataset. This can be done either on the local computer running EC-Star or a remote computer depending on the setup available. The next chapter walks through sample scripts to test the results.

Chapter 4

Example EC-Star run, ABP Data

This section presents a sample run of EC-Star going through the commands necessary to move from raw csv files, up through launching a run of EC-Star. In the following example we use the problem of predicting arterial blood pressure as our dataset. The data describes through time the arterial blood pressure (ABP also referred to as BP) signal of a patient, which is a periodic signal that correlates with the frequency of a heartbeat[3]. The data for each includes features derived from the mean pressure values measured in mmHg for a given beat. By analysing the data the goal is to be able to predict short term future values of the ABP which can aid in the treatment of patients.

4.1 Creating the Data Packages

Starting with a directory of BP csv files we wish to transform this directory of raw csv files into a directory of EC-Star data packages of a specific size. Each line in each csv file consists of 9 features as well as a label. For our data packages, we will take all features and limit each data package size to 300 lines a piece (discarding a few left over lines at the end). In addition a header must be placed on each data file. Passing in a configuration file(Figure 4-1), these steps can all be performed with the command.

```
$ python ecflow.py -c config.cfg --all
```

Note that in addition to the header, the ending data package(Figure 4-3) lacks scientific notation, such values have been converted into decimal so that they can be read by EC-Star. In addition 1 has been subtracted off of the label(the final line) in each row to make them 0 indexed.

Figure 4-1: Config file (config.cfg)

```
[configs]
 data_dir = /home/evo-gf/ECSTAR/BPData/
 data_dest = /home/evo-gf/ECSTAR/ProcessedData/
 #TAKE ALL FEATURES
 FEATURES = []
 ROWS = 300
 exact = true
```

Figure 4-2: Raw csv file

```
73.704,6.3187,-0.00094293,4.3605,0.69608,-0.0017475,8.4688e-06,100,55,2
70.661,3.1971,-0.00067276,13.415,-1.8354,-0.011407,-2.0078e-05,80,67,2
71.446,4.1731,-0.00013142,5.584,1.3112,-0.0010823,-3.1455e-05,83,62,2
72.448,2.47,0.00046213,2.1085,0.20528,0.00057651,-2.3426e-05,73,73,2
70.885,3.1221,-0.00045619,2.5173,0.083632,-0.0011856,-0.00012744,72,72,2
70.409,2.499,0.00028749,2.3742,-0.34978,-0.016183,-0.00015269,72,72,2
68.689,2.6465,0.00022919,2.224,-0.17673,-0.0029468,-0.00015216,72,72,2
69.12,2.8538,1.8462e-05,2.3003,0.045016,-0.014447,-0.00014337,72,72,2
73.184,2.7885,0.00089788,2.2236,-0.14085,0.0022069,-0.00025025,73,73,2
74.002,2.7135,-1.1935e-05,2.0483,0.0057571,-0.0062129,-0.00019792,72,72,2
74.756,2.6364,-0.00065962,1.9315,-0.13984,-0.00059797,-0.00035315,71,71,2
72.209,2.8126,0.00015768,2.7075,-0.065053,-0.0087458,-0.0002053,71,71,2
```

Figure 4-3: Resulting data package(.gdp file, truncated)

```
example_file.gdp
samples 1
sampleIds 0
events 184
fields 12
sampleId, eventId,v1,v2,v3,v4,v5,v6,v7,v8,v9,label
0,0,73.704,6.3187,-0.00094,4.3605,0.69608,-0.00174,0.000000,100.0,55.0,1
0,1,70.661,3.1971,-0.00067,13.415,-1.8354,-0.01140,-0.000020,80.0,67.0,1
0,2,71.446,4.1731,-0.00013,5.584,1.3112,-0.00108,-0.000030,83.0,62.0,1
0,3,72.448,2.47,0.00046,2.1085,0.20528,0.00057,-0.000020,73.0,73.0,1
0,4,70.885,3.1221,-0.00045,2.5173,0.08363,-0.00118,-0.00012,72.0,72.0,1
0,5,70.409,2.499,0.00028,2.3742,-0.34978,-0.01618,-0.00015,72.0,72.0,1
0,6,68.689,2.6465,0.00022,2.224,-0.17673,-0.00294,-0.00015,72.0,72.0,1
```

4.2 Generating the Code

Now that the data packages have been created we must generate code for EC-Star specific to this data. So that EC-Star knows the layout of the data we must first generate `conditions.txt`. Finally the code must be made. Since according to our last configuration file we placed our generated data packages in the directory `ProcessedData/`, we must now update EC-Flow to point to the new directory so that it knows what files to look over when creating `conditions.txt`. This can be done by updating `config.cfg`, or more simply by passing in `ProcessedData/` into the command line call(command line parameters override config file parameters). For the purpose of generating code, we can also update our configuration file to include the values for the lag and lead settings (see appendix A Table A.2 for more config file details)

```
tick = TRUE
 lag = 0
lead = 0
```

We can perform all of the tasks to generate and make the code at once with the command.

```
$ python ecflow.py -c config.cfg -d ProcessedData -C -g --make
```

`-C`, `-g`, and `--make` are the commands to creating `conditions.txt`, generate new code, and make the code respectively. It is important to note that `conditions.txt` should be recreated each time you are working with a new set of data packages.

4.3 Running EC-Star

With the data packages created and the code generated and compiled we must now start the client and servers for the actual run. This step can vary in the amount of preparation needed depending on if the clients and servers are all on a local machine or spread among various computers. In this section we walk through 3 types of setups, running everything locally, distributing the EC-Star runs among multiple virtual machines, and setting up a large scale Grid Machine run.

4.3.1 Local Run

The simplest case of an EC-Star is to run everything locally. Running locally allows us to do experiments and ensure that the system works before scaling it up to larger datasets and more computing resources In this case the clients and servers are launched on the same machine and the data packages are also hosted locally.

Software Installation Before beginning, the following example assumes that the software necessary to run EC-Star locally has already been installed. Before attempting to start EC-Star, ensure that you have the following software installed in addition to the EC-Star code itself (version numbers shown were those used in this example).

Java (1.6.0_24)
GCC (4.6.3)
Python (2.7.3)
MySql (14.14)
Maven (2.2.1)
Make (3.81)
tomcat (7.0.26)
EC-Star GFDataServer

For the MySql database, if not already created you must create and configure a database for use by EC-Star(See Appendix C for example). The EC-Flow code itself consists of `ecflow.py` and the folder `ecflow_scripts` which contains several python and bash script use by `ecflow.py`. After checking out the code for EC-Star, the, `ecflow.py` and `ecflow_scripts` should be placed in the top level directory of the EC-Star code. The code for EC-Flow can be found in the git repo at `https://webdav.csail.mit.edu/groups/EVO-DesignOpt/ECFlow.git/` while the code for EC-Star can be found at `https://webdav.csail.mit.edu/groups/EVO-Des`
.

Placing Data Packages You may have partitioned your data into test and training sets manually or you may have used EC-Flow to create such a partition for you resulting in a `splits.csv` file. In this case we can move the data packages to the appropriate locations using the command

```
$ python ecflow.py -c config.cfg --prep
```

which will move data packages to the test, train and tomcat directories. By default this will look for the file `splits.csv`, and take the first fold, moving the training data to `data/train` the testing data to `data/test` and also moving the training date to `/var/lib/tomcat_dir/webapps/dataPackages` where EC-Star reads data packages from. EC-Flow will automatically clear files out from these old directories, but the directories themselves should already exist.

Configuration Files Next we must point EC-Star at the correct location to find the data packages. This IP address can be changed in the file `client.cfg` located under src/main/resources. In `client.cfg` shown in figure 4-4, the default IP address for the `DataHost` is by default set to the correct local IP address.

Figure 4-4: Sample `client.cfg`.

```
# Programmatically generated file
# File: client.cfg

RPCHost = 127.0.0.1
RPCPort = 8181
DataHost = 127.0.0.1
DataPort = 8080
MaxNumberOfRules = 16
DefaultPoolSize = 500
ElitistPercentage = 20
Debug = true
fitnessHistory = false
maturityAge = 10
```

Starting EC-Star Finally, we start the class server,pool server and client executable as seen in the below script.

```
#! /bin/bash
./run_class-server.sh &> class.log & #start class server
./run_pool-server.sh &> pool.log & #start pool server

cd client/src/resources
if [ -e clientState.esb ] #remove client state log
then
    rm clientState.esb
fi
#Wait for servers to start
sleep 5

#start client
./Bp_client_exec client.cfg &> ../../../client.log &
```

This will start EC-Star locally, the progress of which can be monitored in `class.log`, `pool.log` and `client.log`.

4.3.2 Distributed Data Host, Clients, and Servers

To run EC-Star efficiently on more data then can be handled locally the data host, client and servers can be configured to run on different computers. This largely involves adjusting configuration files to point EC-Star at the correct IP addresses, and making sure data is on the correct server. In our example we use the OpenStack platform to create 3 virtual machines, one for each role data host, client and server. Once each VM has been instantiated we can configure them for each role as follows.

Data Host

Setting up the data host on a server running tomcat is simply a matter of taking all the data packages you expect your client to have access to, and placing them in the directory `/var/lib/tomcat_dir/webapps/dataPackages`. In addition, you should ensure that port 8080 is open so the clients can access the files.GFDataServer must also be installed on the machine to serve as the data host. Once tomcat has been started, you can test that the files are accessible by manually seeing if you can download them from

`http://127.0.0.1:8080/GFDataServer/data/`

replacing the local IP address with the IP address of the data host if attempting to download from another computer.

Clients

To setup a client, we repeat the same process of generating and making the code that we did when running locally. When running just a client on a machine, we only need `Bp_client_exec` and `client.cfg` found in the `client/src/resources` directory of EC-Star. These files, once created, can be zipped and transferred to any machines serving as clients. We also need not store the data packages in the tomcat directory because they will be requested from the data host set up in the previous step. Since we are using a remote date host, we must update the file `client.cfg` changing the field `DataHost` from the default IP address to the address of the data host created in the previous step. We must also update the field `RPCHost` to point to the IP address of the pool servers(created in the next step). Once the configuration files have been updated we can start the clients.

```
#! /bin/bash
cd client/src/resources
if [ -e clientState.esb ] #Remove client state log
then
    rm clientState.esb
fi
#start client
./Bp_client_exec client.cfg &> ../../../client.log &
```

Multiple clients can be run on one server because they are designed to run in a volunteer manner and therefore rarely use all available resources when run on dedicated hardware. However, you must launch the clients in separate directories to avoid overwriting to and reading from the same state files.

Servers (Pool Server and Class Server)

To set up the servers. we must place the generated EC-Star server code (ie the jar files) on the machine . Next in the file `class-server.properties` and the file `pool-server.properties` we update the field `gf.serverclasses.dbhost` to point to the class server(same machine as the pool server in this setup).

`gf.serverclasses.dbhost` = X.X.X.X (ip of pool/class server)

Finally, start the class and pool servers on the machine.

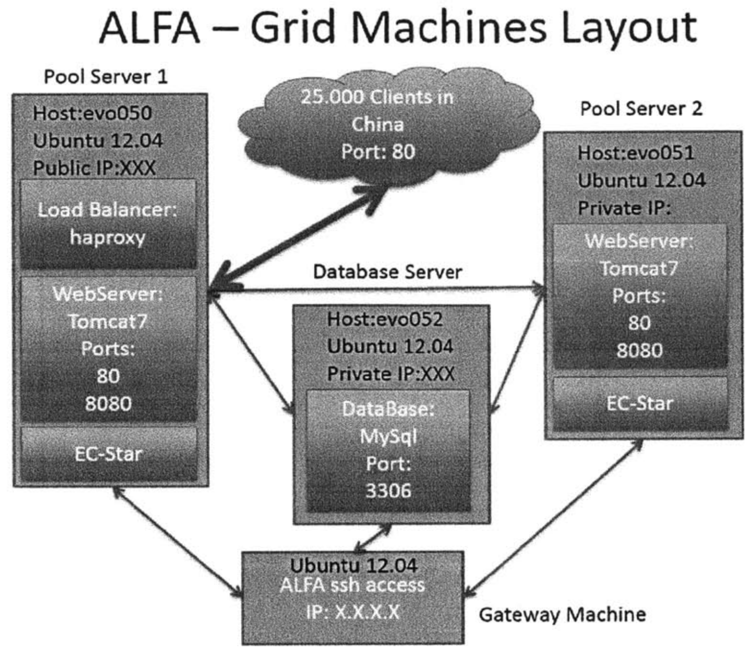


Figure 4-5: This layout consists of 2 pool servers, a database server, the clients, and a gateway machine to configure the servers from. The pool servers are setup as virtual machines running Ubuntu loaded with the EC-Star software. They have 4 core cpus and 8GB of memory each. The database server also has a 4 core cpu with 22GB of memory. The clients are run on the grid machines. The grid machines consist of a volunteer compute network of computers in china. This setup of EC-Star can handle 3,500 clients without a problem and should be able to scale up to 25,000.

```
#! /bin/bash
./run_class-server.sh &> class.log &
./run_pool-server.sh &> pool.log &
```

4.3.3 Grid Machine Setup

In the Grid Machine setup we prepare to scale EC-Star to a large scale installation capable of running thousands of clients (Figure 4-5).

Clone Virtual Machines

Clone 3 instances of an Ubuntu virtual machine with tomcat installed. This should include a computer to serve as the database server, as well as computers to serve as the pool servers. In the above shown example we have 2 computers, evo050 and evo051 to act as our two pool servers. The class servers are also hosted on these two machines. Computer evo052 serves as the database server. Finally one additional machine is provisioned as a gateway machine to the others. Through this machine we can ssh into the others to configure them for EC-Star.

Configure firewall

The firewalls of the pool servers and data server must be configured to secure the server to allow access to designated ports for the pool servers(ports 80 and 8080), database (port 3306) server and `ssh` access (port 22)to other machines. `ufw` - Uncomplicated Firewall, is a convenient tool for this step to open up the necessary ports.

Update Configuration files

So that the machines know which machines that they are communicating with, the files `pool-server.properties` and `class-server.properties` must be updated with the appropriate IP addresses for the servers created in the previous steps. This can be performed in the same manner as shown in the previous section.

Install Load Balancer

Install and configure `HAproxy` load balance server on one of the pool servers, this will balance the load of requests from the clients to the servers.

Start pool sever

To run a Pool server via `Tomcat`, you need to build the .jar file by running `Ant`, stop `Tomcat`, copy the new jar file to the `Tomcat` folder, and then start `Tomcat`.

Setup data server

As in the previous setup(section 4.3.1), the data server should have the data packages placed in the directory `/var/lib/tomcat_dir/webapps/dataPackages`.

Code QA

One the code has been completed and compiled it must be sent to Grid Machines for QA. The code must be checked that it can run within the footprint limitations of the volunteer nodes and not disrupt it.

Launch clients

On an individual basis, clients can be launched following the same procedure as shown in section 4.3.2. Simply copy the client code to the computer and adjust the configuration files to point the IP addresses at the correct machines. This provides an effective way to locally see how a large scale run is behaving by creating your own client.

4.4 Analysing Results

Once EC-Star has had time to run and produce solutions, the solutions can be evaluated by checking them against the partition of data set aside for training. This can be performed either locally or on a remote computer.

First we must pull solutions from the pool server. We can use the following bash script to do this. The script will pull solutions from the pool server and place them in `results/saved_genes/solutions_dir`.

```
#!/bin/bash
NR_GENES=1
POOL_SERVER=127.0.0.1 #Running locally
PORT=8181

#Directory genes will be placed in
SAVE_DIR=results/saved_genes/solutions_dir/

java -cp target/pool-server-jar-with-dependencies.jar \
edu.mit.evodesign.bp.test.Classifier save ${POOL_SERVER}\
${PORT} ${SAVE_DIR}/ ${NR_GENES} > save_genes.log
```

Now that there is an example solution in `results/saved_genes/solutions_dir` the solution can be tested by running the java classifier from the command line.

```
$ java -cp target/pool-server-jar-with-dependencies.jar
edu.mit.evodesign.bp.test.Classifier testAll results/saved_genes/solution_dir

data/test/ > accuracy.log
```

Here the solution in `results/saved_genes/solution_dir` will be tested against the files in `data/test/`. The results will be output to the file `accuracy.log`. If EC-Star is running on a remote computer, the testing framework can be configured to take in an IP address of the machine containing the solutions to evaluate. See Appendix C, Sample Code 2 for a more in depth script example. The the file `accuracy.log` will contain statistics you can use to analyze the how effective the solutions are including confusion matrices and overall accuracy percentages.

Chapter 5

Conclusion and Future Work

Conclusion EC-Star allows the running of evolutionary algorithms in a highly distributed manner, that is also cost efficient because of the use of a volunteer compute network. Overall EC-Flow provides a framework for easily using EC-Star by providing automation to the greatest extent possible at each step in the process. This automation make the use of EC-Star more standardized, user friendly, and less error prone because not as much human input is required. Finally by taking the complexities of manipulating large amounts of data away from the user, it makes the entire process much more efficient and leaves the user to focus on the actual machine learning task at hand.

Future Work Through the use of EC-Flow, every step in the pipeline of using EC-Star is made easier due to the automation EC-Flow provides. There are however ways in which EC-Flow could be further extended to be both more efficient, as well useful in more situations. Firstly, several steps of EC-Flow could benefit from parallelization which could reduce time necessary to do steps such as creating data packages, and testing solutions. `Sleipnir`(Section 3.1.2) already begins to touch on this possibility allowing individual scripts to be executed in parallel virtual machines. However this functionality could further be extended to the entirety of the EC-Flow pipeline in steps where work is able to be split up. In addition, the paradigms introduced in EC-Flow might also be extendable to other machine learning frameworks besides EC-Star. While some aspects of EC-Flow are clearly EC-Star specific many aspects of EC-Flow are likely generalisable to other frameworks, and EC-Flow could in the future be extended to accommodate them.

Appendix A

Tables

Table A.1: EC-Flow Action Options

Command Line Flag	Description
-h,–help	Shows the help message describing all ecflow parameters.
–chunk	Chunk the csv files into files of the specified size.
-s,–select	Select features from the data. Also converts scientific notation to decimal in any selected features.
–adjust,	Goes through the data packages adjusting the lead time in each. (specify lead time through config file or –lead)
-X,–header	Add header to the files. Must pass in he list of feature names for the header using -f or the config file. (Should be the last step called in data package creation.
-C,–conditions	Creates conditions.txt,places it in src/main/resources
–split	Creates a splits.csv file
–xfold	Produces two splits files. Split the files into 10% test and 90% train(90-10_splits.csv). Then splits the the 90% partition into 10 folds(splits.csv).
–clean,	Removes all data packages that have NaN values
-g,–gen	Runs scripts which generate the client and server code
–make	Uses the make file in the top level directory to make the code
–zip	Calls gfcompress on all files in target directory
–unzip	Calls gfuncompress on all files in target directory
–move	Moves a split (given by config file or –move_split) using splits.csv to train_dest and test_dest
–prep	Moves a split (given by config file or –move_split) using splits.csv to train_dest and test_dest and /var/lib/tomcat6/webapps/dataPackages
–readbucket	Tells ECFlow not to overwrite `buckets.txt` so that user defined buckets can be passed in

Table A.2: EC-Flow Configuration Options

Config File Parameter	Default Value	Cmd line flag	Description
data_dir	src_data/	-d, –data	Source directory of csv files that you wish to process. The path given must be absolute.
data_dest	ProcessedData/	-o,–data_dest	Directory processed data packages will be placed in if the data processing procedures creates new files(e.g. chunking).
FEATURES	[]	-f, –feats	list of features to be selected from the data. Passing in [] (the empty list) takes all features.
TRAIN_PERCENT	90	–train_percent	Percentage of files to be used in training when making a partition of the data into testing and training files.
test_dest	data/test	–test_dest	Directory testing files will be moved to if –move is called.
train_dest	data/test	–train_dest	Directory training files will be moved to if –move is called.
ROWS	400	-r, –rows	Number of lines per data package to chunk files into during data package creation.
exact	true	NA	If set to true all data packages that do not have the exact number of specified rows will be deleted. Otherwise they will remain.
move_split	0	–move_split	If –move is used, `move_split` specifies which fold to read from the `splits.csv` file.
NA	splits.csv	split_file	Tells ecflow to use an alternative splits file.
lead	0	`--lead`	If a lead time is desired in the data packages, this value adjusts how far the values are shifted.
lag	0	`--lag`	This value sets the max_tick index in the c and java code.
tick	TRUE	`--tick`	If set to TRUE it indicates to ECStar that there will be a not time lag. If FALSE, there will be a time lag.

Table A.3: Data Package Creation Timings

	1 GB	4 GB	12 GB
3 nodes	398 secs	1672 secs	n/a
6 nodes	232 secs	1200 secs	4530 secs
speedup	1.71x	1.4x	n/a

Timings measured in seconds taken to process given amount of csv files into ECStar data packages and are meant to indicate the relative speeds to accomplish the task for various levels of parallelism.

Appendix B

Figures

Appendix C

Source Code

Sample SQL Code 1. script to set up a database to host ECStar solutions on the pool server

```
/*
 * -- Programmatically generated file --
 * File: serverTrainingDb.sql
 */

CREATE DATABASE `mitServer`;

USE `mitServer`;

DROP TABLE IF EXISTS `classes`;

CREATE TABLE `classes` (
        `classid` char(10) DEFAULT NULL,
        `timestamp` datetime DEFAULT NULL,
        `totalHandshakes` bigint(20) DEFAULT NULL,
        `handshakes` bigint(20) DEFAULT NULL,
        `realhandshakes` bigint(20) DEFAULT NULL,
        `convergenceFactor` float DEFAULT NULL,
        `indicatorSet` blob,
        `snapshotHandshakes` bigint(20) DEFAULT NULL,
        `snapshotConvergenceFactor` float DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS `pool`;

CREATE TABLE `pool` (
        `geneid` char(50) DEFAULT NULL,
        `masterfitness0` double DEFAULT NULL,
        `age` bigint(20) DEFAULT NULL,
        `classid` char(10) DEFAULT NULL,
        `xml` blob,
        `avgFitness` bigint(20) DEFAULT NULL,
        `autoinc` bigint(20) NOT NULL AUTO_INCREMENT,
        `xml_vb` varbinary(65000) DEFAULT NULL,
        `dead` bigint(20) DEFAULT 0,
        UNIQUE KEY `autoinc` (`autoinc`)
```

```
) ENGINE=MyISAM AUTO_INCREMENT=72205 DEFAULT CHARSET=latin1;

DROP TABLE IF EXISTS `redirections`;

CREATE TABLE `redirections` (
        `classFrom` char(10) DEFAULT NULL,
        `classTo` char(10) DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;


/*
 *The below code creates users gfReader, and
 *gfWriter to read and write from the
 *database with password 'password'.
 */
DELETE from mysql.user where user = 'gfWriter';
DELETE from mysql.db where user = 'gfWriter';
DELETE from mysql.user where user = 'gfReader';
DELETE from mysql.db where user = 'gfReader';
FLUSH privileges;

CREATE USER 'gfReader'@'localhost'
IDENTIFIED BY 'password';
GRANT SELECT ON mitServer.* TO 'gfReader'@'%'
IDENTIFIED BY 'password';
GRANT SELECT ON mitServer.* TO 'gfReader'@'localhost'
IDENTIFIED BY 'password';
CREATE USER 'gfWriter'@'localhost' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON mitServer.* TO 'gfWriter'@'%'
IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON mitServer.* TO 'gfWriter'@'localhost'
IDENTIFIED BY 'password';
}
```

Sample Code 2. Script to check accuracy of a solution against test data packages. POOL_SERVER should refer to the IP address where the pool server is running (be it local as in the below example or a remote machine).

```
#! /bin/bash

NR_GENES=1
POOL_SERVER=127.0.0.1
PORT=8181
PREFIX=$(date --utc +'%s') #Timestamp for unique solution

POOL_SERVER_PROCESS=$(ps aux | grep pool-server | grep -v grep )
if [ -z "${POOL_SERVER_PROCESS}" ];
then
    echo "No local pool server"
    exit
fi

SAVED_GENES_DIR=saved_genes/${PREFIX}

if [ ! -e ${SAVED_GENES_DIR} ]
then
    mkdir ${SAVED_GENES_DIR}
fi

# Save genes
cd ..
SAVE_DIR=results/${SAVED_GENES_DIR}

java -cp target/pool-server-jar-with-dependencies.jar \
edu.mit.evodesign.bp.test.Classifier save ${POOL_SERVER}\
${PORT} ${SAVE_DIR}/${PREFIX} ${NR_GENES} > save_genes.log

if [[ "$( ls -1 ${SAVE_DIR}/${PREFIX}*xml | wc -l )" -eq "0" ]]; then
    echo "No files in ${SAVE_DIR}/${PREFIX}"
    exit 0
fi
for packages in "data/train" "data/test"
do
    name=$( basename ${packages} )
    log_file=${SAVE_DIR}/${PREFIX}_${name}.log

    # TestAll genes
    java -cp target/pool-server-jar-with-dependencies.jar \
    edu.mit.evodesign.bp.test.Classifier testAll ${SAVE_DIR}\
    ${packages} > ${log_file}

    cat globalConf.log >> totGlobalConf.log

    conf_file=${SAVE_DIR}/${PREFIX}_${name}_confusion.dat

    # Local confusion matrix
    grep -A 16 -e "Local Confusion Matrix"\
    ${log_file} | grep -e "^[012]" > ${conf_file}
```

```
    ind_acc_file=${SAVE_DIR}/${PREFIX}_${name}_ind_acc.dat

    # Individual accuracy
    grep -H -e "Individual accuracy" ${log_file} > ${ind_acc_file}
    rm ${log_file}
done
```

Sample Code 3. Sleipnir wrapper script.
This python script creates an instance of sleipnir, and uses the script `chunk.sh` to do file chunking. In the below script (and with sleipnir in general) it is assumed that your OpenStack account is already set up and your environment is set up to use OpenStack's `nova` command line tool. The only step once you have created an instance of sleipnir, is to call the `run_job`.The first two arguments are the name of the script and the arguments to the script in the form of an array. You must then pass in the local source directory.This is the path to the files on the local computer used by sleipnir to create a partitioning of the data. Finally pass in the source and destination directories (should be relavitve to the nodes sleipnir is running) and the number of workers (equal to the number of workers listed when we created the instance of sleipnir).

```
from sleipnir import Sleipnir
#Create Sleipnir Instance and pass in OpenStack node names
#(IP address will be automatically looked up)
cloud = Sleipnir(['Sleipnir_0','Sleipnir_1','Sleipnir_2'])

#src directory (data is stored on an nfs)
src = '/data/alfa/ecstar-abp/abpData/'

#directory for output files
dest='/data/alfa/ecstar-abp/abpChunked/'

#src directory (on local computer)
local_src = '/data/alfa/ecstar-abp/abpData/'

#Call run_job method of Sleipnir
rows = "1000"

#args:
#script,[script args], src directory, destination directory, number of workers
cloud.run_job("chunk.sh",[rows],local_src,src,dest,3)
```

Sample Code 4. Chunking Sleipnir Script

The below script for chunking using sleipnir is nearly identical to that used by ECFlow. Note that the only requirements imposed by sleipnir is that the first input be the `src` directory and the final input be the output directory. Any other inputs are optional. In this case there is one additional input, the number of rows.

```
#! /bin/bash
#Puts chunked files in original directory and removes original files
DIR=$1 #src directory
ROWS=$2 #if 0, do not split files
OUTPUT_DIR=$3 #output directory
SUFFIX='.csv'
CORES=4

T="$(date +%s)"
echo "Splitting"
find ${DIR} -name "${PREFIX}*${SUFFIX}" | \
xargs -P ${CORES} -I {} split -a 3 -d -l ${ROWS} {} $( basename {} )_


echo "Renaming file"

find ${DIR} -name "${PREFIX}*" -type f | \
xargs -P ${CORES} -I '{}' rename 's/.csv_/_/g' '{}'

echo "Adding ${SUFFIX}"
find ${DIR} -name "${PREFIX}*" -type f | \
xargs -P ${CORES} -I {} mv {} {}${SUFFIX}


T="$(($(date +%s)-T))"
echo "Time in seconds: ${T}" > time.txt
#move to output directory on afs
T="$(date +%s)"
find ${DIR} -type f | xargs -P ${CORES} -I {} mv {} $OUTPUT_DIR
T="$(($(date +%s)-T))"
echo "Time to move in seconds: ${T}" >> time.txt
```

Bibliography

1. O'Reilly UM, Wagy M, Hodjat B (2012) Ec-star: A massive-scale, hub and spoke,distributed genetic programming system. In: Genetic Programming Theory and Practice X, Springer

2. Hodjat B,Hemberg E, Shahrzad H and O'Reilly UM (2013): Maintenance of a Long Running Distributed Genetic Programming System For Solving Problems Requiring Big Data

3. Hemberg E, Veeramachaneni K, Dernoncourt F, Wagy N, O'Reilly UM (2013): Efficient Training Set Use For Blood Pressure Prediction in a Large Scale Learning Classifier System