SNoW User Guide

Andrew J. Carlson, Chad M. Cumby, Jeff L. Rosen, Dan Roth

Cognitive Computation Group
Computer Science Department
University of Illinois, Urbana/Champaign

Urbana, Illinois
August, 1999

Contents

Chapter 1

Introduction

SNoW is a learning architecture that is specifically tailored for learning in the presence of a very large number of features. and can be used as a general purpose multi-class classifier.

The current release of the SNoW architecture is the second generation of the original SNoW learning architecture developed by Dan Roth. SNoW stands for Sparse Network of Winnows. The learning architecture is a sparse network of sparse linear functions over a pre-defined or incrementally acquired feature space; several update rules may be used - sparse variations of the Winnow update rule, the Perceptron, or naive Bayes.

SNoW is a multi class learner, where each class is represented as a single *target* node, learned as a linear function over the feature space or as a combination of several of those, organized into clouds. Each of these representations is learned from labeled data in an incremental fashion. Both the representation architecture (i.e., which "features" are important) and the features' weight are determined by SNoW. Decision made by SNoW are either binary – indicating which of the labels is predicted for a given example, or continuous (in $[0, 1]$) – indicating a confidence in the prediction. Several other output modes are available.

SNoW has been used successfully in several applications in the natural language and visual processing domains; the release is meant to be used only for research purposes, with the hope that it can be a useful research tool for studying learning in these domains. Feedback of any sort is welcome.

Dan Roth
Urbana, IL. Aug. 1999.

The document is organized as follows. Chapter 2 contains the software license, under the University of Illinois terms. Users need to agree to it in order to used the software and register on-line. Chapter 3 describes how to install the SNoW system. Chapter 4 gives a brief overview of the learning architecture and the technical approach. A detailed description of how to use the system follows in Chapter 5, where all the command line arguments and modes of operation are described. Next, in Chapter 6, the formats of the various files used by the system is described. Finally, Chapter 7 is a tutorial showing how to use the system with various options.

A new user is encouraged to read all of this document, but the best starting place for learning to use the system is the tutorial. The tutorial gives a good sense of the required steps for using the system. Once a user is comfortable with the default method of using the system, the more detailed description of the command line options given in Chapter 5 may be more useful.

Chapter 2

License terms

UNIVERSITY OF ILLINOIS
"SNoW" SOFTWARE LICENSE AGREEMENT

(RTMO Technology #)
(Research and Academic Use License)

Cognitive Computation Group
University of Illinois at Urbana-Champaign

Downloading and using the SNoW software implies that you accept the following license:

Under this Agreement, The Board of Trustees of the University of Illinois ("University"), a body corporate and politic of the State of Illinois with its principal offices at 506 South Wright Street, Urbana, Illinois 61801, U.S.A., on behalf of its Department of Computer Science on the Urbana-Champaign Campus, provides the software ("Software") described in Appendix A, attached hereto and incorporated herein, to the Licensee identified below ("Licensee") subject to the following conditions:

- 1. Upon execution of this Agreement by Licensee, the University grants, and Licensee accepts, a royalty-free, non-exclusive license:
  - A. To use unlimited copies of the Software for its own academic and research purposes.
  - B. To make derivative works. However, if Licensee distributes any derivative work based on or derived from the Software (with such distribution limited to binary form only), then Licensee will (1) notify the University (c/o Assistant Professor Dan Roth, e-mail: danr@cs.uiuc.edu) regarding its distribution of the derivative work and provide a copy if requested, and (2) clearly notify users that such derivative work is a modified version and not the original Software distributed by the University.
  - C. To redistribute (sublicense) derivative works based on the Software in binary form only to third parties provided that (1) the copyright notice and any accompanying legends or proprietary notices are reproduced on all copies, (2) no royalty is charged for such copies, and (3) third parties are restricted to using the derivative work for academic and research purposes only, without further sublicensing rights.
- No license is granted herein that would permit Licensee to incorporate the Software into a commercial product, or to otherwise commercially exploit the Software. Should Licensee

Chapter 3

Installation

The SNoW system is available for download as gzipped tar archive from:

`http://L2R.cs.uiuc.edu/~cogcomp/`

Before downloading the archive, you must register as a user and accept the license agreement. You can then download the file `snow.tar.gz`, which contains complete source code (C++) for the SNoW program, the tutorial data files, and the documentation and license. The program can be compiled easily on most UNIX systems as well as on Windows machines as a console application. To install the system, unzip the downloaded file:

```
> gunzip snow.tar.gz
```

and unpack the tar archive:

```
> tar -xvf snow.tar
```

This will make a directory `Snow` under your current directory. Change directory to this:

```
> cd Snow
```

and compile the binary by typing `make`. If compilation is successful, you should now have an executable named `snow`.

Chapter 4

The SNoW Architecture

The SNoW learning architecture is a sparse network of linear functions over a pre-defined or incrementally learned feature space and is specifically tailored for learning in domains in which the potential number of features taking part in decisions is very large, but may be unknown a priori. Some of the characteristics of this learning architecture are its sparsely connected units, the allocation of features and links in a data driven way, its computational dependence on the number of active features rather than the total number of features and the utilization of a feature efficient update rule. SNoW has been used successfully on a variety of large scale learning tasks in the natural language domain [Roth, 1998, Golding and Roth, 1999, Roth and Zelenko, 1998, Munoz et al., 1999] and, more recently, in the visual processing domain.

In this release SNoW is to be thought of as a general purpose multi-class classifier. The class labels are called below the *target* nodes and they are learned as sparse linear functions over the input features. By *sparse* we mean here that each target may be learned as a function of a (small) subset of all the features known to the system, in a data driven way that is partially controlled by parameters set by the user.

The user defines the architecture of the learner. This means, as a minimum, defining the number of class representations to be learned, but may include also defining many more parameters of the architecture, including the update rules used, the number of learned units representing a target and more. Although any number of targets units can be defined, due to the decision mechanisms used, it is recommended that the number of targets used is the number of different classes (i.e., two in the case of a two-class problem).

When viewing SNoW simply as a classification system, the typical input would be a collection of labeled exampled, in a format specified in Chapter 6. The following section provides a slightly more abstract view that may be useful for people in the stage of modeling their problem as a learning problem.

4.1 The SNoW System

The SNoW (Sparse Network of Winnows[1])) learning architecture is a sparse network of linear units over a common pre-defined or incrementally learned feature space. Nodes in the input layer of the network represent simple relations over the input and are being used as the input features. Each linear unit is called a *target node* and represents relations which are of interest over the input

[1]To winnow: to separate chaff from grain.

examples, namely the class labels. Given a set of relations (i.e., *types* of features) that may be of interest in the input example, each input example is mapped into a set of features which are *active* (present) in it; this representation is presented to the input layer of SNoW and propagates to the target nodes. (Features may take either binary value, just indicating the fact that the feature is active (present) or real values, reflecting its strength. Target nodes are linked via weighted edges to (some of the) input features. Let $\mathcal{A}_t = \{i_1, \ldots, i_m\}$ be the set of features that are active in an example and are linked to the target node t. Then the linear unit is *active* if and only if $\sum_{i \in \mathcal{A}_t} w_i^t > \theta_t$, where w_i^t is the weight on the edge connecting the ith feature to the target node t, and θ_t is its threshold.

The learning policy is on-line and mistake-driven (except when naive Bayes is used); several update rules can be used within SNoW. The most successful update rule is a variant of Littlestone's Winnow update rule, a multiplicative update rule tailored to the situation in which the set of input features is not known a priori, as in the infinite attribute model [Blum, 1992]. A sparse variation of Perceptron is also available. This mechanism is implemented via the sparse architecture of SNoW. That is, (1) input features are allocated in a data driven way – an input node for the feature i is allocated only if the feature i is active in the input example and (2) a link (i.e., a non-zero weight) exists between a target node t and a feature i if and only if i has been active in an example labeled t. Thus, the architecture also supports augmenting the feature types from external sources in a flexible way.

The Winnow update rule has, in addition to the threshold θ_t at the target t, two update parameters: a *promotion* parameter $\alpha > 1$ and a *demotion* parameter $0 < \beta < 1$. These are being used to update the current representation of the target t (the set of weights w_i^t) only when a mistake in prediction is made. Let $\mathcal{A}_t = \{i_1, \ldots, i_m\}$ be the set of active features that are linked to the target node t. If the algorithm predicts 0 (that is, $\sum_{i \in \mathcal{A}_t} w_i^t \leq \theta_t$) and the received label is 1, the active weights in the current example are *promoted* in a multiplicative fashion: $\forall i \in \mathcal{A}_t, w_i^t \leftarrow \alpha \cdot w_i^t$. If the algorithm predicts 1 ($\sum_{i \in \mathcal{A}_t} w_i^t > \theta_t$) and the received label is 0, the active weights in the current example are *demoted*: $\forall i \in \mathcal{A}_t, w_i^t \leftarrow \beta \cdot w_i^t$. All other weights are unchanged. The key feature of the Winnow update rule is that the number of examples[2] it requires to learn a linear function grows linearly with the number of *relevant* features and only logarithmically with the total number of features. This property seems crucial in domains in which the number of potential features is vast, but a relatively small number of them is relevant (this does not mean that only a small number of them will be active, or have non-zero weights). Winnow is known to learn efficiently any linear threshold function and to be robust in the presence of various kinds of noise and in cases where no linear-threshold function can make perfect classification, and still maintain its abovementioned dependence on the number of total and relevant attributes [Littlestone, 1991, Kivinen and Warmuth, 1995].

The Perceptron update rule is used in a similar way within the sparse architecture. It takes only two parameters, a threshold and a learning rate. As in Winnow, whenever a mistake is made, the weight of an active feature is updated. In this case, it is updated by either adding the learning rate parameter or subtracting it, depending on whether the mistake is on a positive example or a negative one, respectively.

In case of naive Bayes, the weights are simply the logarithm of the fraction of the examples that are labeled according to the target (see, e.g., [Roth, 1999, Roth, 1998]. In addition, the relative weight of the target is used as "prior", and a fixed smoothing is used.

Notice that when using Perceptron and Winnow, examples that are positive for a target (i.e.,

[2]In the on-line setting [Littlestone, 1988] this is usually phrased in terms of a mistake-bound but is known to imply convergence in the PAC sense [Valiant, 1984, Helmbold and Warmuth, 1995].

which are labeled with the target) are considered negative for all other targets. This is not the case for naive Bayes. Each target takes into account only the examples labeled with it (i.e., the target representation is learned only from positive examples).

Once target subnetworks have been learned and the network is being evaluated, a decision support mechanism is employed, which selects the dominant active target node in the SNoW unit via a winner-take-all mechanism to produce a final prediction. It is possible for units' output to be cached and processed along with the output of other SNoW units to produce a more complicated decision support mechanism (as in, e.g., [Munoz et al., 1999]).

Chapter 5

Using SNoW

5.1 Execution Modes

SNoW may be run in three major modes. These are: Training, Testing and Evaluation. Training takes a set of labeled examples and generates a network which can then be used to make predictions on future examples. Thus, training must always be performed before testing or evaluations can be performed. Testing is performed on a set of (labeled or unlabeled) examples contained in a file. The results of the test are written to the terminal display or to a file. Evaluation is used to make a prediction based on a labeled or unlabeled example supplied on the command line.

Training consists of presenting labeled examples (see the section on example file format below) and learning a weight vector representation for each target concept. At the completion of training, the resultant network is written to the network file.

Testing consists of presenting examples to the system and predicting a label. This is done by using the representation learned for each of the targets to compute that target's activation level given the new example, and choosing the one with the highest activation. The results of testing can be output in a number of ways.

Several other modes of operations, including incremental learning, are supported.

5.2 Command line usage

5.2.1 Mode Selection

The basic usage is as follows:

```
snow -mode [options]
```

Where `-mode` must be one of the following:

`-train` : The system is run in training mode and the input file is considered to be a set of labeled training examples. Each example in the file is considered a positive example for all targets which are included (active) in the example, and negative examples for all other targets (absent from the example).

`-test` : The system is run in a batch test mode. The input file can be labeled or unlabeled testing examples. Each example in the file is presented to the system and classified. If the examples are labeled, the result of the classification can be compared with the label of the example and scored, with a final accuracy reported after all examples are presented. The result of each prediction can also be output to a file in a number of ways and scored externally.

`-evaluate` : The system is run in an interactive test mode. A single labeled or unlabeled example is supplied on the command line. The process terminates after making a prediction for this single example. The results are output and the target with the strongest activation is also returned to the operating system as the exit status of the process which made prediction. Note that running the system in evaluate mode loads the network once for each example, and thus is not the best way to process large sets of examples.

Some options are required, some are optional and some may be required, optional or superfluous depending on the *mode* in which the system is run. For a view of the required options, refer to Chapter 7, the Tutorial.

5.2.2 Architecture Definition

Defining the architecture is one of the required options. This can be done using an architecture file as described below, or directly in the command line. The main part of defining the architecture is setting up the target nodes. For each target node one has to specify which of the update rules is used to when learning it representation. If Winnow (-W) are Perceptron (-P) are used, it is possible to specify also their parameters; otherwise, default parameters are used.

Target definition is done by specifying for which target ID's a representation is learned. Either single ID's or ranges of targets can be given. For example, the simplest architecture file may be `-W 0-1`

It specifies two targets, for the target ID's 0 and 1, to be learned using the Winnow update rule with the default parameters. This architecture is suitable for a two-class situation.

For a more involved case consider an architecture file which reads:

```
-W 1.5,0.8,4.0,0.5:0-2,5,9
-P 0.1,4.0,0.2:1-3,4,8
```

Here, Winnow will be used to learn a representation for targets 0, 1, 2, 5, and 9, and Perceptron will be used for targets 1, 2, 3, 4, and 8. Note that when more than one algorithm is specified for a single target ID, the outputs of those algorithms will be combined to make a single prediction for that target.

Although the targets can be defined in the command line, we recommend using the architecture file option, especially when experiments are done. This would a allow a simple way of running with various parameters and architectures. The options for the architecture definition are specified below.

`-A <architecture file>` : Specifies the name of a file from which to read the desired architecture definition and parameters. The file may look, for example, like:

```
-W 1.5,0.8,4.0,0.5:0-1
-P 0.1,4.0,0.20:0-1
-e 1
-r 4
```

`-B :targets` : Specifies targets that will be trained using the naive Bayes algorithm.

`-P <learning_rate>, <threshold>, <default_weight>:targets` : Specifies targets that will be trained using the single layer perceptron algorithm, along with their parameters.

`-W <promotion>, <demotion>, <threshold>, <default_weight>:targets` : Specifies targets that will be trained using the winnow algorithm along with their parameters.

5.2.3 Algorithm Parameter Definition

The following parameters are all optional. As in all other cases, they can be defined as parts of the architecture file or in the command line.

`-b <k>` : Specifies the smoothing parameter to be used in Bayes learners (default 15).

`-d <none | abs:<k> | rel>` : Specifies the discarding method, if any. Absolute discarding discards components of the weight vector with a magnitude less than some threshold (specified as a real number k in the command line). Relative discarding compares the weights for a given feature within a network. The smallest of those weights is discarded and the rest are adjusted accordingly. Note that this method reduces the total weight (magnitude) of the weight vector. This method was developed specifically for winnow networks and is probably best used only with winnow networks. Discarding is done only when training, every 1000 examples. The default is `none`.

`-e <i>` : This option sets the eligibility threshold– a minimum number of times any given feature must be active before it can be added to a network. If unspecified, the default value is 2. For example, if `-e 3` is specified and feature 12836 appears twice in the training file, then this feature would not be included in the network. Something to consider when using this option is that the corresponding weight for the feature cannot be promoted or demoted until the feature is included in the network. This option doesn't apply to naive Bayes learners and is only applicable to training. Also, when running small experiments, it is sometimes helpful to set the eligibility threshold to 1.

`-i <+ | ->` : This option specifies whether to use incremental learning. That is, whether or not mistakes made during testing are used to update the network . If `-i +` is specified then mistakes made during testing are used to update the network and the network is written out after the test with "`.new`" appended to its original filename.

QUESTION HERE

It is possible to use this option in order to initially train with one algorithm, and then retrain the resulting network with a different algorithm. This option only affects perceptron and winnow networks. In order to use this option, test examples must be labeled.

`-l <+ | ->` : This option specifies whether test examples are labeled or not.

`-m <+ | ->` : This option specifies whether training examples should be treated as having multiple labels. If `-m +` is specified, the targets ID's that appear in a training example will not be treated as features for training, thus a target will not be learned as a function of other features. In unspecified, or if `-m -` is specified, targets will also be treated as features during training and can occur in the representation of other features. This option is of interest in text categorization applications when, typically, examples (documents) have multiple labels, but we do not learn one label in terms of the other.

`-p <k>` : This option specifies a prediction threshold which must be met in order for SNoW to make a prediction. If unspecified, it defaults to 0.0. This option can be used as a confidence filtering. Filter out cases in which SNoW is not confident enough in the prediction. In test mode, if the activations of the targets with the two highest activations differ by less than the prediction threshold, no prediction is made and a targetID of -1 is output. This only applies to the `accuracy` and `winners` output modes.

`-r <i>` : Specifies the number of cycles (passes) through the training data. If unspecified the default is 2. Multiple passes through the training data can sometimes improve the resulting network. In ce naive Bayes networks treat the training sample as a probabilistic estimate, multiple passes do not change the resultant network, i.e. only perceptron and winnow are affected by this option.

`-s <s | f>` : Specifies whether to use the sparse or full network option. This setting only affects perceptron and winnow networks. In a sparse (the default) network, features are only linked to targets (that is, are given a non zero weight) if both the feature and the target have appeared active together in an example. In contrast, a full network has each target linked to the same set of features, that is if a feature is linked to any target, then it is linked to all targets in that network.

`-w <k>` : Specifies a smoothing value for winnow and perceptron learners.

5.2.4 Input/Output Options

`-a <+ | ->` : Specifies whether to use highly accurate weights for features or to approximate them when writing out the network. If not specified, the weights are approximated. This has little effect on performance in most cases.

`-c <i>` : In training mode. The interval, in the number of examples presented, at which to output a snapshot of the network. So every *i* examples, a file named *networkfile.niiiii*, where networkfile is the filename specified with `-f` (see below), *n* is a literal 'n', and *iiiii* is a zero-padded value for the number of examples presented before the snapshot was created. This option is useful for producing a learning curve (accuracy vs. number of training examples). For example, if networkfile is 'myfile.net' and `-C 500` is specified, then the following files would be written after the first 1000 examples were presented:

```
myfile.net.n00500
myfile.net.n01000
```

`-E <errorfile>` : Specifies the name of a file in which to write information about mistakes during testing. If the file already exists it is overwritten. This option is only valid during testing.

`-F <networkfile>` : Specifies the name of a file in which the resulting networks are written to (after training) or read from (for testing).

`-I <inputfile>` : Specifies the input file from which examples are read. During training and testing the input file specifies the training examples and testing examples, respectively.

`-o <accuracy | winners | allpredictions | allactivations | allboth>` : Specifies which output mode to use when reporting results during test mode. The predictions in SNoW are done using a winner-take-all policy. That is, all the targets activation levels are compared and the predicted target ID is the one with the largest activation level. Several out modes are available, and are described below.

`accuracy` : This output mode requires labeled examples. SNoW compares each prediction to each example's label and keeps track of correct and incorrect predictions, outputting an accuracy report at the end of testing. This is the default output mode.

```
186 test examples presented
Overall Accuracy - 97.96%
```

`winners` : This mode outputs the targetID with the highest activation for each example in the test set. For example, if we had targets with ID's 0, 1, and 2, output might appear as:

```
1
2
0
2
1
```

`allpredict` : This mode outputs, for every example, a list of all targets and their predictions (1 or 0, indicating if it was the chosen target for this example or not). The target with the highest activation is predicted as true (1), and the rest are false (0). The output is sorted by ID.

```
Example 47
0:  1
1:  0
2:  0
Example 48
0:  0
1:  0
2:  1
```

`allactivations` : For each example, this mode outputs the activation of each target. The activation is a number between 0 and 1, with 1 representing a very strong positive prediction, and 0 representing a strong negative prediction. The output is sorted by activation.

```
Example 47
0:  0.95845
2:  0.60394
1:  0.44093
Example 48
2:  0.92312
0:  0.53439
1:  0.65443
```

`allboth` : This mode outputs, for every example, a list of all targets and both their activations and predictions as described in the above modes.

```
Example 47
0:  1 0.95845
2:  0 0.60394
```

```
1:  0 0.44093
Example 48
2:  1 0.92312
1:  0 0.65443
0:  0 0.53439
```

`-R <results_file>` : Specifies the name of a file in which the results of testing are output. If this parameter is unspecified, the output will be directed to the console.

`-T <testing_file>` : Using this option, a network can be trained and tested in the same invocation of snow. After training the network, test examples are read from the file specified using this flag, and output is given just as if snow was run in *test* mode. In this case, the network file is not saved.

Chapter 6

File Formats

Two types of files are used by SNoW. Examples files are required. They store the training and test examples and are only read by SNoW. Network files are written by SNoW during training and are read during testing. They store the representation for each of the target nodes, as well as some more information on the structure of the SNoW network. The use of Network files is not mandatory (if the -T option is being used) but is recommended for research purposes. In addition SNoW may use error files and files for reporting results.

6.1 Example Files

Example files (files consisting of examples) are ASCII text files. Each line of the file contains a single example and ends with a colon. SNoW represent examples as a list of indices of active features, and does not distinguish between features and class labels. All are considered features. Thus, each example consists of a list of non-negative numbers, indices of active features, some of which may be of special interest to the user that may like to distinguish them as targets. When SNoW receives an example, it searches it's list of target, to see if any of them is active in the example. If it is, this example is considered a positive example for this target. If it is not, it is considered a negative example for this target.

Thus, a training example can have many targets active in it (many labels) and their location within the list of active features does not matter.

In testing mode, SNoW does not need the labels, in principle. It reads the example evaluates all existing targets on it, and produces a prediction, via a winner-take-all policy. If the user wants SNoW to keep statistics of it's performance by itself (which is the default) then the true target (the label needs to be supplied along with the example. In this case, it is required that the label be the first (left most) feature in the example, so that SNoW can keep correct statistics, in case there are several indices for which targets exists active in the example. Examples are thus list of the form:

```
7,5,1,13:
0,3,1234,123456,12,987,234,556:
```

In addition, it is possible to supply SNoW with a strength of the feature. This number is supplied in parenthesis, behind the index. The default value (no parenthesis) is 1. An example may be:

```
7(1.5),5(3),10(0.6),13(10):
```

The affect of the strength is that when evaluating the example, the corresponding weight of the feature is multiplied by the strength.

label	x_1	x_2	x_3	x_4	x_5	x_6	SNoW Example
false	0	0	0	0	0	1	7:
false	0	0	0	0	1	0	6:
false	0	0	0	1	0	0	5:
true	0	0	1	0	1	1	1,4,6,7:
true	0	1	0	1	0	0	1,3,5:
false	1	0	0	1	0	1	2,5,7:
true	1	0	1	1	1	0	1,2,4,5,6:
true	0	1	0	1	1	1	1,3,5,6,7:

Table 6.1: Partial truth table and SNoW examples for x_2 v x_5

Example

Problem HERE: 1 - does not work; 2 - need to recommend two targets For example, consider a learning problem where a boolean concept over 6 boolean variables is to be learned. The value of 1 might be used to represent the label true. The values 2 through 7 could represent the variables x_1 through x_6 respectively. Each example is represented by a comma delimited list of labels and active features, and ends with a colon. As a concrete example, consider the concept x_2 v x_5. A partial truth table for this concept and the format of SNoW examples is given in table 1.

6.2 Error Files

Error files (generated with the `-E` command line option) can sometimes provide insight into why the network failed to learn the target concept exactly. Error files can only be created in test mode, using labeled test examples. Error files contain two sections. The first section records each learning algorithm which was used in the network. Following the algorithms is a list of examples which were labeled incorrectly. Each occurrence shows the example sequence number and each target's activation For example, in the following example there was a mistake on example 3 (the 3rd example in the input file), in which target 0 had an activation of 0.2022 and target 1 had an activation of 0.5445.

```
Algorithms:
1:  Winnow network:(0.2, 4, 1.25, 0.8)

Ex:  3 Prediction:  1 Label:  0
0:  0.2022*
1:  0.5445

Ex:  6 Prediction:  1 Label 0
0:  0.1488*
1:  0.8999
```

6.3 Network Files

Network files (specified by the `-F` command line flag) contain all of the information required for SNoW to recreate the structure generated during training. The first line of the network file specifies the type of network file— either `lean` or `accurate`. The `lean` type uses a lower degree of precision when writing out the weights of features to conserve space. However, this has been shown to have a negligible effect on performance in most cases.

The first line can thus be either:

```
networktype.lean
```

or:

```
networktype.accurate
```

Each target has a section of the file, containing information on the target itself, its algorithm and parameters, and information on its features. The header line for each target takes the following form:

`target` *`ID priorProbability confidence count algorithm algID parameters`*

For example:

```
target 2 1 0.473593433165 42 winnow 1 1.35 0.8 4 0.2
```

This specifies a target with a ID 2, prior probability 2, confidence 0.4736, and which appeared active in 42 examples. It uses a winnow algorithm with ID 1, alpha 1.35, beta 0.8, threshold 4, and default weight 0.2.

Following the target header, the target's features are enumerated. Each line corresponds to a single feature, in the format:

`ID : algID : featureID : count updates weight`

If the `lean` network type is used, lines will take the format:

`featureID : count updates weight`

For example:

```
1 :  2 :  34 :  13 6 0.36450000000000000462
```

This specifies a feature with corresponds to target concept 1 and algorithm 2, with featureID 34. It appeared in 13 positive training examples for target 1, had its weight updated 6 times and has a weight of about 0.3645.

Chapter 7

Tutorial

This tutorial is meant to demonstrate how to use many of the basic options of SNoW. We show how to start with training examples, train a classifier, and then test the classifier with some more examples. These are the basic steps to using SNoW. The example task given is context-sensitive text correction. The task is to train a classifier with many examples of the correct usage of the words "their" and "there" so that, given a test context, the classifier can decide which word best fits the context.

7.1 Training

We start with a file containing labeled examples. Our target concepts have ID's of `0` and `1`, and all other numbers appearing in the examples represent other features present (word colocations, parts of speech, etc.). The first example appearing in the training file is:

```
0,96,116,119,120,128,138,157,212,230,328,451,454,601,636,641,
646,773,774,815,872,897,937,1134,1160,1197,1231,1267,1461,1503,
1576,1640,1654,1838,1845,1878,1937,1941,1946,1953,1986,2012,
2387,2612,2958,3211,3221,3222,3233,3242,3308,3315,3318,3487,
3524,3526,3897,4037,4136,4404,6933,6991,7269,7298,7398,7488,
7539,7562,7755,7794,8032,8377,9336:
```

Here, the example has a label of `0`, meaning that it will be a positive example for target `0` and a negative example for all other targets (in our case, just target `1`. The label must always appear as the first feature, and in training, labels must always be present. All examples are terminated with a colon. The original sentence which the above example was generated from is:

```
In the interim between now and next year, we trust the House and Senate will put <<
their >> minds to studying Georgia's very real economic, fiscal and social problems
and come up with answers without all the political heroics.
```

The above example was generated from this sentence by using Feature Extractor, a program which generates features based on specified patterns— words near the target word, parts of speech near the target word, and numerous other options. In our set of data, the label `0` in examples represents the target concept "their" and the label `1` represents the target concept "there."

Given our training data (provided in the file traindata.feat), we can now train a classifier which will be able to classify new examples from outside the training set based on what the system learned about the features present in the training data. In order to train our network, we must invoke SNoW in training mode with our training examples as the input file. We do this as follows:

```
snow -train -I tutorial/traindata.feat -F tutorial/test.net -W :0-1
```

This gives the output:

```
SNoW - Sparse Network of Winnows Plus
Cognitive Computations Group - University of Illinois at Urbana/Champaign
Version 2.01.14
Network file:  'tutorial/test.net' Input file:  'tutorial/traindata.feat' Network Spec
-> w(:0-1),
```

The output from SNoW lets us know if there were any errors in the parameters we entered, and also gives information on the learning algorithm used. Here, we used a Winnow learning algorithm with default parameters by specifying the `-W :0-1` flag. This tells SNoW to use a default set of parameters (which work quite well for most experiments) and that our target concepts have ID's `0` and `1`. Different algorithms and parameters can be specified on the command line or in an Architecture file, as will be shown later in the tutorial.

The training made two cycles through our training data, which is the default. The number of cycles can be specified on the command line, and generally, the more cycles used, the closer the classifier comes to completely learning the training data.

7.2 Testing

We now have our network file, `test.net`, which contains the parameters of our Winnow algorithm as well as weights for all of the features which appeared in our training examples. Now that we've trained our network, we can proceed to testing it on some more examples.

```
snow -test -I tutorial/testdata.feat -F tutorial/test.net
```

Our test file contains labeled examples of exactly the same format as those used in testing, and we can just use the default output mode and let SNoW score our accuracy. In this mode, each example is given to the system and the resulting prediction output by the classifier is compared to the example's label. A mistake is scored if the two do not match. Here are our results:

```
SNoW - Sparse Network of Winnows Plus
Cognitive Computations Group - University of Illinois at Urbana/Champaign
Version 2.01.14
Network file:  'tutorial/test.net' Input file:  'tutorial/testdata.feat' Directing
output to console
850 test examples presented
Overall Accuracy - 97.18%
```

We can also verify how well the training data was learned by the classifier by testing with our training set as our test examples. If our accuracy is 100%, the classifier completely learned the training set.

To receive output on a more detailed level, we can specify different output modes on the command line. We can do this by using the `-o` ***outputmode*** flag. For example, executing SNoW as follows:

```
snow -test -I tutorial/testdata.feat -F tutorial/test.net -o allactivations
```

This gives the output:

```
Example 1
0:  0.351881
1:  0.934405
Example 2
0:  0.792367
1:  0.134328
Example 3
0:  0.110389
1:  0.780127
```

7.3 Other Options

If we want to use algorithms other than the default, we can specify them by using the `-W`, `-P`, and `-B` flags with their associated parameters. Each algorithm we define can be assigned to any number of targets. To run the tutorial experiment with algorithms we specify, we can run SNoW as follows:

```
snow -train -I tutorial/traindata.feat -F tutorial/test.net
-W 1.5,0.8,4.0,0.5:0 -P 0.1,4.0,0.20:1
```

This uses a Winnow algorithm for target 0 and a Perceptron for target 1. We could also execute SNoW like this:

```
snow -train -I tutorial/traindata.feat -F tutorial/test.net
-W 1.5,0.8,4.0,0.5:0-1 -P 0.1,4.0,0.20:0-1
```

This will use both a Winnow and Perceptron on each target, combining the results of the algorithms to calculate a single activation for each target.

Also, command line parameters can be specified in an *architecture file*. This file is specified with the `-A` flag, and contains parameters which don't need to be changed frequently. For example, when running an experiment over many datasets, the only parameters which change from dataset to dataset will usually be the `-I` inputfile and the `-F` network file flags. We could thus use an architecture file with most of our command line options. The file (named `archfile`) could read:

```
-W 1.5,0.8,4.0,0.5:0-1
-P 0.1,4.0,0.20:0-1
```

```
-e 1
-r 4
```

Using this architecture file, we have defined two algorithms, set the *eligibility threshold* to 1, and set SNoW to train on four cycles through the training data. Now we can use these parameters with any data files. We execute SNoW with the architecture file specified:

```
snow -train -I tutorial/traindata.feat -F tutorial/test.net -A archfile
```

Bibliography

[Blum, 1992] Blum, A. (1992). Learning boolean functions in an infinite attribute space. *Machine Learning*, 9(4):373–386.

[Golding and Roth, 1999] Golding, A. R. and Roth, D. (1999). A winnow based approach to context-sensitive spelling correction. *Machine Learning*, 34(1-3):107–130. Special Issue on Machine Learning and Natural Language.

[Helmbold and Warmuth, 1995] Helmbold, D. and Warmuth, M. K. (1995). On weak learning. *Journal of Computer and System Sciences*, 50(3):551–573.

[Kivinen and Warmuth, 1995] Kivinen, J. and Warmuth, M. K. (1995). Exponentiated gradient versus gradient descent for linear predictors. In *Proceedings of the Annual ACM Symp. on the Theory of Computing.*

[Littlestone, 1988] Littlestone, N. (1988). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318.

[Littlestone, 1991] Littlestone, N. (1991). Redundant noisy attributes, attribute errors, and linear threshold learning using Winnow. In *Proc. 4th Annu. Workshop on Comput. Learning Theory*, pages 147–156, San Mateo, CA. Morgan Kaufmann.

[Munoz et al., 1999] Munoz, M., Punyakanok, V., Roth, D., and Zimak, D. (1999). A learning approach to shallow parsing. In *EMNLP-VLC'99, the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora.*

[Roth, 1998] Roth, D. (1998). Learning to resolve natural language ambiguities: A unified approach. In *Proc. National Conference on Artificial Intelligence*, pages 806–813.

[Roth, 1999] Roth, D. (1999). Learning in natural language. In *Proc. Int'l Joint Conference on Artificial Intelligence*, pages 898–904.

[Roth and Zelenko, 1998] Roth, D. and Zelenko, D. (1998). Part of speech tagging using a network of linear separators. In *COLING-ACL 98, The 17th International Conference on Computational Linguistics*, pages 1136–1142.

[Valiant, 1984] Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.