

**NIST Form-Based Handprint Recognition System
(Release 2.0)**

NISTIR 5959

**Michael D. Garris, James L. Blue, Gerald T. Candela,
Patrick J. Grother, Stanley A. Janet, and Charles L. Wilson**

National Institute of Standards and Technology,
Building 225, Room A216
Gaithersburg, Maryland 20899

ACKNOWLEDGEMENTS

We would like to acknowledge the Internal Revenue Service and the Bureau of the Census who provided funding and resources in conjunction with NIST to support the development of this standard reference optical character recognition system.

TABLE OF CONTENTS

1. INTRODUCTION	1
1.1 First System Release	2
1.2 Second System Release	3
1.3 Document Organization	3
2. INSTALLATION INSTRUCTIONS	5
2.1 Installing from CD-ROM	5
2.2 Organization of Software Distribution	7
2.3 Source Code Subdirectory	9
2.4 Automated Compilation Utility	10
3. INVOKING TEST-BED PROGRAMS	13
3.1 mis2evt - computing eigenvector basis functions	13
3.2 mis2pat1 - generating patterns for the PNN classifier	15
3.3 hsfys1 - running the updated version of the original NIST system	17
3.4 mis2pat2 - generating patterns for training the MLP classifier	19
3.5 trainreg - training to register a new form	21
3.6 hsfys2 - running the new NIST recognition system	22
4. ALGORITHMIC OVERVIEW OF NEW SYSTEM HSFSYS2	24
4.1 The Application	24
4.2 System Components	24
4.2.1 Batch Initialization; src/lib/hsf/run.c; init_run()	25
4.2.2 Load Form Image; src/lib/image/readrast.c; ReadBinaryRaster()	26
4.2.3 Register Form Image; src/lib/hsf/regform.c; genregform8()	26
4.2.4 Remove Form Box; src/lib/rmline/remove.c; rm_long_hori_line()	27
4.2.5 Isolate Line(s) of Handprint; src/lib/phrase/phrasmap.c; phrases_from_map()	28
4.2.6 Segment Text Line(s); src/lib/adseg/segchars.c; blobs2chars8()	29
4.2.7 Normalize Characters; src/lib/hsf/norm8.c; norm_2nd_gen_blobs8()	31
4.2.8 Extract Feature Vectors; src/lib/nn/kl.c; kl_transform()	32
4.2.9 Classify Characters; src/lib/mlp/runmlp.c; mlphypscons()	32
4.2.10 Spell-Correct Text Line(s); src/lib/phrase/spellphr.c; spell_phrases_Rel2()	32
4.2.11 Store Results; src/lib/fet/writfet.c; writfetfile()	34
5. PERFORMANCE EVALUATION AND COMPARISONS	35
5.1 Accuracies and Error Rates	35
5.2 Error versus Rejection Rate	40
5.3 Timing and Memory Statistics	41
6. IMPROVEMENTS TO THE TEST-BED	44
6.1 Processing New Forms with the HSFSYS2	44
7. FINAL COMMENTS	45
8. REFERENCES	46

A. TRAINING THE MULTI-LAYER PERCEPTRON (MLP) CLASSIFIER OFF-LINE	48
A.1 Training and Testing Runs	48
A.2 Specification (Spec) File	49
A.2.1 String (Filename) Params	49
A.2.2 Integer Params	50
A.2.3 Floating-Point Params	51
A.2.4 Switch Params	52
A.3 Training the MLP in <i>hsfsys2</i>	55
A.4 Explanation of the output produced during MLP training	56
A.4.1 Pattern-Weights	56
A.4.2 Explanation of Output	56
A.4.2.1 Header	56
A.4.2.2 Training Progress	57
A.4.2.2.1 Second progress lines	58
A.4.2.2.2 First progress lines	59
A.4.2.2.3 Pruning lines (optional)	60
A.4.2.3 Confusion Matrices and Miscellaneous Information (Optional)	61
A.4.2.4 Final Progress Lines	63
A.4.2.5 Correct-vs.-Rejected Table (Optional)	64
A.4.2.6 Final Information	65

NIST Form-Based Handprint Recognition System

(Release 2.0)

Michael D. Garris (mgarris@nist.gov)

James L. Blue, Gerald T. Candela, Patrick J. Grother,
Stanley A. Janet, and Charles L. Wilson

National Institute of Standards and Technology,
Building 225, Room A216
Gaithersburg, Maryland 20899
FAX: (301)840-1357

ABSTRACT

The National Institute of Standards and Technology (NIST) has developed a new release of a standard reference form-based handprint recognition system for evaluating optical character recognition. As with the first release, NIST is making the new recognition system freely available to the general public on CD-ROM. This source code test-bed, written entirely in C, contains both the original and the new recognition systems. New utilities are provided for conducting generalized form registration, intelligent form removal with character stroke preservation, robust text-line isolation in handprinted paragraphs, adaptive character segmentation based on writing style, and sophisticated Multi-Layer Perceptron (MLP) neural network classification. A software implementation of the machine learning algorithm used to train the new MLP is included in the test-bed, enabling recipients to train the neural network for pattern recognition applications other than character classification. A host of data structures and low-level utilities are also provided. These include the application of spatial histograms, affine image transformations, simple image morphology, skew correction, connected components, Karhunen Loève feature extraction, dictionary matching, and many more. The software test-bed has been successfully compiled and tested on a host of UNIX workstations including computers manufactured by Digital Equipment Corporation, Hewlett Packard, IBM, Silicon Graphics Incorporated, and Sun Microsystems.¹ Approximately 25 person-years have been invested in this software test-bed, and it can be obtained free of charge on CD-ROM by sending a letter of request via postal mail or FAX to NIST. This report documents the new recognition software test-bed in terms of its installation, organization, and functionality.

1. INTRODUCTION

In August of 1994, the National Institute of Standards and Technology (NIST) released to the public a standard reference form-based handprint recognition system for evaluating optical character recognition (OCR) [1]. The system served as a vehicle for transferring recognition and performance assessment technology from our government laboratory to system developers and researchers in the private sector. As of August 1996, over 700 copies of the technology had been distributed to more than 40 countries around the world. This was NIST's first large-scale public domain OCR technology transfer, and by all accounts it has been a tremendous success.

Since 1994, NIST has continued to conduct research in form-based handprint recognition. This research is critical to the continued advancement of the technology. This is especially true with regards to system integration. Form-based OCR has the potential of solving many economically important problems using state-of-the-art technology, but currently there is no universal off-the-shelf solution available for large-scale, centralized forms processing applications. These applications are comprised of many tasks or functional components, and the literature contains a plethora of algorithms and techniques for accomplishing these various tasks [2]. Even so, one cannot expect to be able to arbitrarily pick and choose techniques available as off-the-shelf products, organize them into a standard work flow, and proceed to universally solve applications. The fact is, interactions between components are often nonlinear and

1. Specific hardware and software products identified in this paper were used in order to adequately support the development of the technology described in this document. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the equipment identified is necessarily the best available for the purpose.

non-additive [2]. The economically useful systems being deployed today are successful because they are constructed from components that have been customized to capitalize on all the constraints afforded by a particular application. The more constraints available and incorporated, the higher the probability of success. Therefore, these systems are defined more by their intended application than by available general purpose technology.

The interactions between recognition system components are complex and difficult to model, therefore it is not possible with conventional knowledge to measure the performance of a component in isolation and to predict a component's impact on overall system performance. The only meaningful way to compare alternative components for use in an application is by integrating each alternative into an end-to-end system and comparing their impact on overall system performance. This has been the focus of much of our research, as effective performance assessment facilitates the comparison of technical alternatives and more importantly helps insure successful deployment of technology to specific applications. To support this research, NIST has developed numerous algorithms and techniques, and to study their impact on recognition performance, these components have been integrated into a prototype (or test-bed) system. This software test-bed is what comprises this new release of the NIST form-based handprint recognition system.

Using the software test-bed, a component of the system may be easily replaced by an alternative algorithm. The same set of input data can be run through the augmented system, and performances between the original and augmented system can be compared. Also, by retraining and testing the recognition system in a controlled fashion, training sets can be collected and evaluated that improve system robustness. Developers may find that the techniques provided in the standard reference test-bed provide complementary results to their own systems. If this is the case, then combining their recognition results with those from NIST may improve overall recognition performance.

A CD-ROM distribution of this software can be obtained free of charge by sending a letter of request via postal mail or FAX to Michael D. Garris at the address above. Requests made by electronic mail will not be accepted, however electronic mail is encouraged for handling technical questions. The letter, preferably on company letterhead, should identify the requesting organization or individuals. Any portion of this software test-bed may be used without restrictions because it was created with U.S. government funding. This software test-bed was produced by NIST, an agency of the U.S. government, and by statute is not subject to copyright in the United States. Redistribution of this standard reference software is strongly discouraged as any subsequent corrections or updates will be sent to registered recipients only. Recipients of this software assume all responsibilities associated with its operation, modification, and maintenance.

1.1 First System Release

This new software release contains the latest technology from our laboratory. Due to the factors described above, there is no *best* algorithm for a specific system component, and there is no *best* suite of components to comprise a universal system. The question should not be which component algorithm is best, but rather which combination of algorithms performs best for a particular application. What works best for one application may not work as well for another. Therefore, this new technology does not necessarily replace or make the technology distributed in the first release obsolete. As a result, the new software distribution contains both the new and the original recognition systems. The new system is an embellishment to the old one.

The software provided with the first release remains mostly intact. We are happy to say that, among the more than 700 recipients over that last two years, there were only a handful of bugs reported from the 19,000 lines of code distributed. These included a couple of syntax errors and a few memory inefficiencies and leaks. None of these problems were reported to cause fatal errors at run-time. By correcting one memory inefficiency, the time required by the dictionary matching process was cut by more than 25%. Other inefficiencies removed include changing system calls from *calloc()* to *malloc()* wherever possible, thus avoiding the overhead of unnecessarily zeroing out memory. By making implementation changes to the existing algorithms, the first system's execution time was reduced by more than 40%, and memory allocation requirements were reduced by 35%. The file *doc/changes.txt* lists the changes made to the source code between its first and second release.

1.2 Second System Release

As already mentioned, the new release contains the latest improved technology from our laboratory. Alternatives to system components are provided that are more general, more robust, and statistically more adaptive. With the new recognition system, the application remains the same. Both the new and the old systems are designed to read handwritten responses on Handwriting Sample Forms (HSF) like those distributed in *NIST Special Database 19* (SD19) [3]. An example of one of these completed forms is shown in Figure 1. The new system incorporates new methods for form registration [4], form removal [5], text line isolation in handprinted paragraphs [6], character segmentation [7], and new pattern classification [8]. The only component remaining virtually the same from the original system is the dictionary-based spelling correction [9].

1.3 Document Organization

This document provides installation instructions, describes the organization of the software test-bed including its compilation and invocation, and presents a high-level description of each of the major algorithms utilized in the new recognition system. Section 2 contains instructions for installing the test-bed from CD-ROM. This includes a description of the test-bed's organization, the size of various parts of the distribution, and instructions on compiling the provided software. Section 3 documents how each of the provided programs (excluding classifier training) were used to generate the supporting files provided in the distribution and how these programs can be invoked on new sets of data. Section 4 describes the major algorithms designed and integrated into the new NIST recognition system. Section 5 contains comprehensive performance evaluation results. This section compares three recognition systems: the original system as it was distributed in the first release, an updated version of the original system as it is distributed in this release, and the new NIST recognition system containing the latest technology developed in our laboratory. Results are reported from running these systems across all of SD19. Statistics and comparisons are reported on character, word, and field-level accuracies, error versus reject performances, system timings, and memory usages. To conduct this evaluation, a total of 3669 writers, 109,200 words, and 667,758 characters were used in the tests. Improvements to the software test-bed are discussed in Section 6 along with a short description of how the new recognition system can be set up to process new and different types of forms. A few final comments and concluding remarks are provided in Section 7, and references are listed in Section 8. Note that all NIST publications referenced in this document are provided in PostScript format on the CD-ROM.

The NIST recognition software test-bed not only contains pre-trained classifiers, but it provides extensive training data along with the machine learning algorithms implemented in software for retraining the classifiers. In fact it is possible for recipients of this test-bed to train the provided classifiers on other pattern recognition applications, in addition to character classification. The new NIST recognition system utilizes a sophisticated Multi-Layer Perception (MLP) neural network-based classifier, the training program for which is documented in Appendix A.

HANDWRITING SAMPLE FORM

NAME	DATE	CITY	STATE	ZIP
[REDACTED]	08-03-89	Holland	Mi.	49424

This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below.

0 1 2 3 4 5 6 7 8 9		0 1 2 3 4 5 6 7 8 9		0 1 2 3 4 5 6 7 8 9	
0123456789		0123456789		0123456789	
97	420	5290	15880	932784	
97	420	5290	15880	932784	
459	6104	53943	420501	69	
459	6104	53943	420501	69	
3291	60118	047763	56	607	
3291	60118	047763	56	607	
35424	183567	52	067	1258	
35424	183567	52	067	1258	
193828	83	768	7146	79293	
193828	83	768	7146	79293	

ixnvlksjbuhtpwoyqgefmdrcz

IXNVLKSJBUHTPWOYQGEFMDRCZ

EDOSMZLTUHGXRWKA FNVJYQIPCB

EDOSMZLTUHGXRWKA FNVJYQIPCB

Please print the following text in the box below:

We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.

<p>We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the Common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.</p>

Figure 1. An example of a completed HSF form from SD19.

2. INSTALLATION INSTRUCTIONS

The public domain recognition system software is designed to run on UNIX workstations and has been successfully compiled and tested on a Digital Equipment Corporation (DEC) Alpha, Hewlett Packard (HP) 9000, IBM RS6000, Silicon Graphics Incorporated (SGI) Indy and Challenge, and Sun Microsystems (Sun) SPARCstation 10 and SPARCstation 2. Porting the software to smaller Personal Computer (PC) platforms is left entirely to the recipient as NIST does not have resources allocated to support such efforts at this time.

As mentioned in the introduction, this distribution contains two different recognition systems. An updated version of the original system, *hsfsys1*, is provided along with a new and improved system, *hsfsys2*. Unlike the first release which contained some isolated FORTRAN, the new software release is written completely in C (traditional Kernighan & Ritchie, not ANSI) and is organized into 15 libraries. In all, there are approximately 39,000 lines of code supporting more than 725 subroutines. Source code is provided for tasks such as form registration, form removal, field isolation, field segmentation, character normalization, feature extraction, character classification, and dictionary-based postprocessing. A host of data structures and low-level utilities are also provided. These utilities include the application of CCITT Group 4 decompression [10][11], IHead file manipulation [1], spatial histograms, Least-Squares fitting [12], affine image transformations, skew correction, simple morphology [13], connected components, Karhunen Loève (KL) feature extraction [14], Probabilistic Neural Network (PNN) classification [15], Multi-Layer Perceptron (MLP) classification [16], and Levenstein distance dynamic string alignment [17].

Several other programs are provided that generate data files used by the two recognition systems. The first program, *mis2evt*, computes a covariance matrix and generates eigenvectors from a sample of segmented character images. The next program, *mis2pat1*, produces prototype feature vectors for use with the PNN classifier in *hsfsys1*, while *mis2pat2* produces prototype feature vectors for use with the new MLP classifier in *hsfsys2*. The program *mlp* trains an MLP neural network on the feature vectors produced by *mis2pat2* and stores the resulting weight matrices to file for later use in classification. These feature vectors are computed using segmented character images and the eigenvectors produced by *mis2evt*. To support these programs, a training set of 1499 writers contributing 252,124 segmented and labeled character images is provided in the distribution. These writers correspond to partitions *hsf_4*, *hsf_6*, & *hsf_7* in SD19.

2.1 Installing from CD-ROM

The NIST recognition software is distributed on CD-ROM in the ISO-9660 data format [18]. This format is widely supported on UNIX workstations, DOS or Windows-based personal computers, and VMS computers. Therefore, the distribution can be read and downloaded onto these various platforms. Keep in mind that the source code has been developed to compile and run on UNIX workstations. If necessary, it is the responsibility of the recipient to modify the distribution source code so that it will execute on their particular computer architectures and operating systems.

Upon receiving the CD-ROM, load it onto your computer using a CD-ROM drive equipped with a device driver that supports the ISO-9660 data format. You may need to be assisted by your system administrator as mounting a file system usually requires root permission. Then, recursively copy its contents into a read-writable file system. Table 1 lists the size (in kilobytes) of the directories on the CD-ROM before and after compilation. The entire distribution requires approximately 360 Megabytes (Mb) to copy. The top-level distribution directory *doc* contains just over 105Mb of PostScript reference documents, and the directory *train* about 27.5Mb of training data. These files are not necessary to compile and run the recognition systems, so they do not have to be copied from the CD-ROM if disk space is limited on your computer. However, the segmented characters within *train* are required if you wish to retrain any of the neural network classifiers. The entire distribution requires approximately 365Mb upon compilation.

Directory	Pre-Comp	Post-Comp
./bin	1	1146
./data	637	637
./dict	1	1
./doc	105276	105276
./include	99	99
./lib	0	796
./man	54	54
./src	2232	5310
./tmpl	97	97
./train	27493	27493
./weights	224110	224110
./weights/pnn	55145	55145
./weights/mlp	168964	168964
Total	360000	365019

Table 1. Sizes (in 1024 byte blocks) of distribution directories before and after compilation.

The CD-ROM can be mounted and the entire distribution copied with the following UNIX commands on a Sun SPARCstation:

```
# mount -v -t hfs -o ro /dev/sr0 /cdrom
# mkdir /usr/local/hfsys2
# cp -r /cdrom /usr/local/hfsys2
# umount -v /cdrom
```

where */dev/sr0* is the device file associated with the CD-ROM drive, */cdrom* represents the directory to which the CD-ROM is mounted, and */usr/local/hfsys2* is the directory into which the distribution is copied. If the distribution is installed by the root user, it may be desirable to change ownership of the installation directory using the *chown* command. CD-ROM is a read-only medium, so copied directories and files are likely to retain read-only permissions. The file permissions should be changed using the *chmod* command so that directories and scripts within the copied distribution are readable, writable, and executable. All catalog files should be changed to be read-writable. In general, source code files can remain read-only. Section 2.2 identifies the location of these various file types within the distribution. Specifically, the file *bin/catalog.csh* must be assigned executable permission, and files with the name *catalog.txt* under the top-level *src* directory must be assigned read-writable permission.

By default, the distribution assumes the installation directory to be */usr/local/hfsys2*. If this directory is used, the software can be compiled directly without any path name modifications. To minimize installation complexity, the directory */usr/local/hfsys2* should be used if at all possible. If insufficient space exists in your */usr/local* file system, the installation can be copied elsewhere and referenced through a symbolic link.

If you decide to install this distribution in some other directory, then editing a number of source code files will be necessary prior to compiling the programs. Edit the line "PROJDIR = /usr/local/hfsys2" in the file *makefile.mak* in the top-level installation directory, replacing */usr/local/hfsys2* with the full path name of the installation directory you have chosen. Likewise replace all references to */usr/local/hfsys2* in the files *hfsys.h* and *hfsys2.h* found in the top-level directory *include*. Remember, to make these file modifications, the permission of these files will have to be changed first. Once these edits are made, follow the instructions in Section 2.4 for compilation.

2.2 Organization of Software Distribution

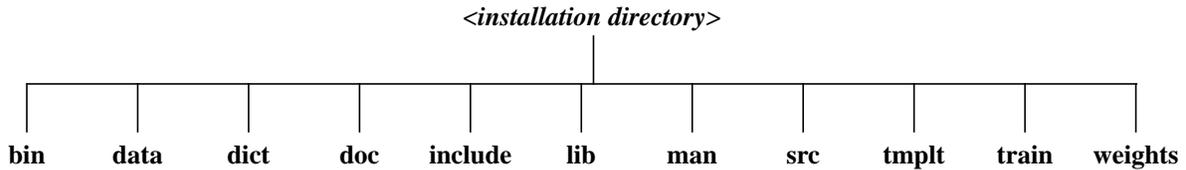


Figure 2. The top-level directory structure in the software distribution.

The top-level directories in this distribution are shown in Figure 2. The first directory, *bin*, holds all distributed shell scripts and locally compiled programs that support the recognition system. The full path name to this directory should be added to your environment's search path prior to compilation. Upon successful compilation, the programs *mis2evt*, *mis2pat1*, *hsfsys1*, *mis2pat2*, *trainreg*, *mlp*, and *hsfsys2* are installed in the top-level *bin* directory. Instructions on running these programs are provided in Section 3 with the exception of *mlp* which is discussed in Appendix A. The directory *bin* also contains the file *catalog.csh* that must be assigned executable permission. This file is a C-shell script that is used to automatically catalog programs and library routines.

The directory *data* contains 10 subdirectories, *f0000_14* through *f0009_06*, containing completed HSF forms from SD19. Each subdirectory holds a form image in an IHead format file with extension *pct*, a reference file with extension *ref* (listing the values the writer was instructed to enter in each field), three system output files generated by *hsfsys1*, and two system output files generated by *hsfsys2*. The hypothesis file with extension *hy1* lists all the field values recognized on the form by *hsfsys1*, while the file with extension *hy2* lists all those recognized by *hsfsys2*. The confidence file with extension *co1* lists the corresponding confidence values for each character classification reported by *hsfsys1*, while the file with extension *co2* lists confidence values for *hsfsys2*. A timing file with extension *ti1*, generated by *hsfsys1*, is also provided in each form directory. A single timing report, *hsfsys2.ti2*, generated by *hsfsys2* running across all 10 forms, is stored in the top-level directory *data*. All of these system output files were generated at NIST on a Sun SPARCstation 2 with a Weitek CPU upgrade.

The directory *dict* contains the dictionary file *const.mfs* listing in alphabetical order all the words present in the Preamble to the U.S. Constitution. The directory *include* holds all the header files that contain constants and data structure definitions required by the recognition system source code. The directory *lib* holds all locally compiled object code libraries used in compiling the distribution programs. The directory *src* contains all the source code files (excluding header files in the top-level directory *include*) provided with the recognition system distribution. The organization of *src* subdirectories is discussed in Section 2.3.

Documentation on this software test-bed is provided in the top-level directory *doc*. The file *changes.txt* lists all the source code modifications made to the software between the first and second releases. A significant number of PostScript reference documents are also contained in this directory. The PostScript file for this specific document is *hsfsys2.ps*. The remaining files in this directory form a bibliography of papers and reports published by NIST that are relevant to this software release. For example, the user's guide for the first release of the software, NISTIR 5469 [1], is contained in the file *bib_15.ps*. NISTIR 5469 should be referenced for its algorithmic description of the original recognition system which in the new release is renamed *hsfsys1* and contains only minor modifications. The installation and organizational notes included in *bib_15.ps* are made obsolete by the notes provided in this (the new release) user's guide. The bibliography files are assigned file names according to the order of their publication date. The text file *bib_lis.txt* cross-references all the bibliography file names to their associated publication titles and full references. All but three of the bibliography files are PostScript documents ending with the extension *ps*. The files *bib_05.tar* and *bib_13.tar* were created with the UNIX *tar* command, and they contain multiple PostScript files. For example, the PostScript files contained in the file *bib_05.tar* can be extracted into the current working directory using the following command:

```
% tar xvf bib_05.tar
```

The files *bib_14.ps* and *bib_14.z* contain the Second Census Optical Character Recognition Systems Conference report [19]. The first part is a PostScript file, whereas the second part is a UNIX compressed tar file. To extract the PostScript files archived in *bib_14.z*, use the following command. Warning, extracting these files requires a large amount of disk space.

```
% zcat < bib_14.z | tar xvf -
```

On-line documentation is also provided in the top-level directory *man* in the form of UNIX-style manual pages. These manual entries give instructions on running each of the programs provided in the test-bed. For example, assuming the installation directory is */usr/local/hsfsys2*, one can bring up a manual page on the screen for the program *mis2evt* by typing the following command on a Sun workstation. Command options may vary on your particular system.

```
% man -M /usr/local/hsfsys2/man mis2evt
```

The directory *tmplt* contains files pertaining to the processing of HSF forms. A blank HSF form is provided in both Latex and PostScript formats. The Latex file *hsf_0.tex* or the PostScript file *hsf_0.ps* can be printed, filled in, scanned at 12 pixels per millimeter (300 dpi), and then recognized by both recognition systems. The points used to register an HSF form in *hsfsys1* are stored in the file *hsfreg.pts*. The coordinates used to register forms in *hsfsys2* are stored in the file *hsfgreg.pts*. *Hsfsys2* uses a generalized method of form registration that is automatically trained with the program *trainreg*. Points defining the location of each HSF entry field are stored in the file *hsftmplt.pts*. A registered blank HSF form image is stored in the file *hsftmplt.pct*, and a dilated version of this form used for form removal in *hsfsys1* is stored in the file *hsftmplt.d4*.

A large sample of training data is provided in the top-level directory *train*. As mentioned earlier, there are 252,124 segmented and labeled handprint characters contained in this directory. In all there are 179,829 images of handprint digits, 35,783 lowercase letters, and 36,512 uppercase letters. The handprint from 1499 different writers are represented in this set of character images, which is divided among three subdirectories *hsf_4*, *hsf_6*, and *hsf_7*. These subdirectories correspond directly to those distributed in SD19. The images of segmented characters are stored in the Multiple Image Set (MIS) file format [1]. Each MIS file ends with the extension *mis*. Those files beginning with *d* contain data related to handprint digits, files beginning with *l* correspond to lowercase letters, and files beginning with *u* correspond to uppercase letters. The four digit number embedded in each file name is an index identifying the writer. For each MIS file in the training set, there is an associated classification file containing the identity of each character contained in the MIS file. These classification files end with the extension *cls*. The first line in a classification file contains the number of character images contained in the corresponding MIS file. All subsequent lines store the identity (in hexadecimal ASCII representation) of each successive character image. MIS files containing images of lowercase letters have a second classification file associated with them that ends with the extension *cus*. These files store the identity of each lowercase letter as their corresponding uppercase equivalent. For example, an image of the lowercase character 'k' is stored in a *cls* file as 6b, whereas it is stored in a *cus* file as 4b (the hexadecimal ASCII representation for the uppercase character K). The labelling of lowercase letters as uppercase is used when classifying characters in the Preamble box.

The last top-level directory *weights* holds the files associated with feature extraction and character classification. This directory is divided into two subdirectories. Subdirectory *pnn* contains files that support the PNN classifier used in *hsfsys1*, whereas the subdirectory *mlp* contains files that support the MLP classifier used in *hsfsys2*. The files under each of these two subdirectories are organized according to the types of fields found on an HSF form. *Digit* contains files for numeric recognition, *lower* for lowercase recognition, *upper* for uppercase recognition, and *const* for Preamble recognition. Within the *weights/pnn* subdirectories, files with the extension *evt* were generated by the program *mis2evt* and contain eigenvector basis functions used to compute Karhunen Loève (KL) coefficients. The pattern (or prototype) files with the extension *pat* contain training sets of KL prototype vectors and a search tree [20] used by the PNN classifier. Files with extension *med* in this subdirectory contain class-based median vectors computed from the prototypes stored in the corresponding *pat* file. Pattern and median vector files stored under *pnn* were computed by the program *mis2pat1*.

The files in *weights/pnn/digit: h6_d.evt, h6_d.pat, and h6_d.med* were computed from 61,094 images of digits in *train/hsf_6* and are used by *hsfsys1* to compute features and classify segmented images of digits. The files in *weights/pnn/lower: h46_l.evt, h46_l.pat, and h46_l.med* were computed from 24,205 lowercase images in both *train/hsf_4* and *train/hsf_6* and are used by *hsfsys1* to compute features and classify lowercase characters. The files in *weights/pnn/upper: h46_u.evt, h46_u.pat, and h46_u.med* were computed from 24,420 uppercase images in both *train/hsf_4* and *train/hsf_6* and are used by *hsfsys1* to compute features and classify uppercase characters. The files in *weights/pnn/const: h46_ul.evt, h46_ul.pat, and h46_ul.med* were computed from 48,625 images of both lower and uppercase in *train/hsf_4* and *train/hsf_6* and are used by *hsfsys1* to compute features and classify characters for lower and uppercase combined. Two additional pairs of *evt, pat, and med* files are provided so that computers with limited memory of at least 8 Megabytes are able to execute all options of *hsfsys1*. The files in *weights/pnn/const: h6_ul_s.evt, h6_ul_s.pat, and h6_ul_s.med* were computed from 24,684 images of both lower and uppercase only in *train/hsf_6*, whereas in *weights/pnn/digit: h6_d_s.evt, h6_d_s.pat, and h6_d_s.med* were computed from 21,293 images of digits in *train/hsf_6*. In general, the recognition accuracy of the PNN classifier decreases as the number of prototypes is decreased. Therefore, the larger pattern files should be used when possible.

Unlike the PNN classifier, the MLP classifier requires extensive off-line training, and this is performed by the program *mlp*. The MLP classifier also uses KL feature vectors, but in a different file format than is used by PNN. The program *mis2evt* is used to compute eigenvector basis functions, and *mis2pat2* is used to generate pattern files for use with the program *mlp*. Within the *weights/mlp* subdirectories, eigenvectors are stored in files with extension *evt*, pattern files with extension *pat*, and output weights files from the program *mlp* are stored with extension *wts*. The same set of writers and characters was used to train the MLP classifier (on digits, lowercase, uppercase, and mixed case for the Preamble box) that were used to generate the patterns files for the PNN. An additional set of 500 writers contained in *train/hsf_7* was used as an evaluation set during the off-line training of the MLP. Appendix A describes how the program *mlp* was used to generate the provided weights files.

2.3 Source Code Subdirectory

The organization of subdirectories under the top-level directory *src* is illustrated in Figure 3. The subdirectory *src/bin* contains all program main routines. Included in this directory is a *catalog.txt* file providing a short description of each program provided in the test-bed. In this distribution there are seven programs and therefore seven subdirectories in *src/bin: mis2evt, mis2pat1, hsfsys1, mis2pat2, mlp, trainreg, and hsfsys2*. The program *mis2evt* takes MIS files of segmented character images and computes eigenvectors from the collection; *mis2pat1* generates a patterns file and median vector file for use with the PNN classifier; *hsfsys1* is an updated version of the recognition system distributed in the first software release; *mis2pat2* generates patterns files to be used in training the MLP classifier; *mlp* is the off-line training program that computes weights for the new MLP classifier; *trainreg* trains the generalized form registration module (used in the new recognition system) on new types of forms; and the last program, *hsfsys2*, is the new recognition system that contains the latest technology from our laboratory and performs significantly better than its older counterpart. Each of these program directories contains a C source code file containing the program's main routine (designated with the extension *c*) and a number of different architecture-dependent compilation scripts used by the UNIX *make* utility (designated with the root file name *makefile*). The use of the *make* utility is discussed in Section 2.4. Upon successful compilation, the directories under *src/bin* will contain compiled object files and a development copy of each program's executable file. Production copies of these programs are automatically installed in the top-level directory *bin*.

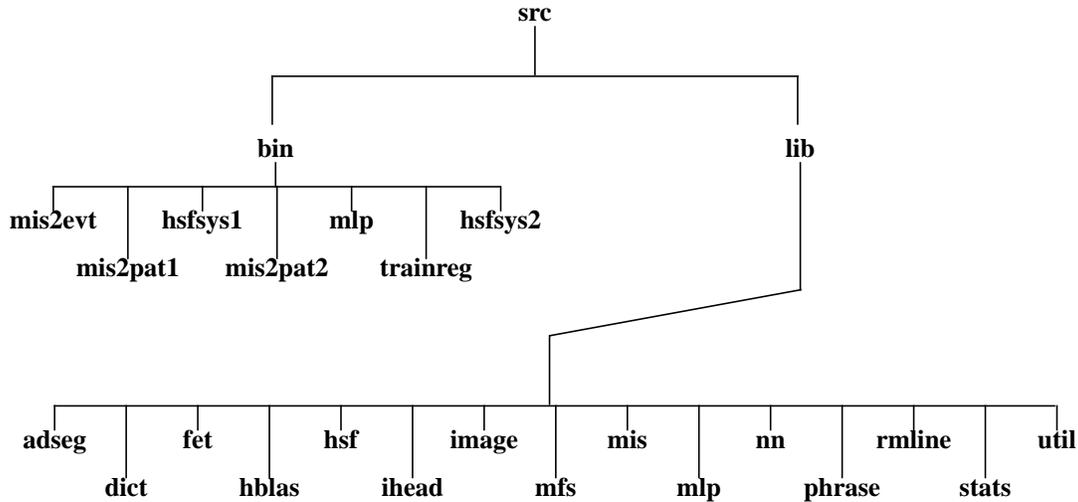


Figure 3. Directory hierarchy under the top-level directory *src*.

The subdirectory *src/lib* contains the source code for all the recognition system's supporting libraries. This distribution has 15 libraries each represented as a subdirectory under *src/lib*. Each library contains a suite of C source code files designated with the extension *c* and a set of different architecture-dependent compilation scripts designated with the root file name *makefile*. Also included in each library subdirectory is a *catalog.txt* file providing a short description of each routine contained in that specific library. Upon successful compilation, each library subdirectory under *src/lib* will contain compiled object files (with file extension *o*) and a development copy of each library's archive file (with file extension *a*). Production copies of the library archive files are automatically installed in the top-level directory *lib*.

The *adseg* subdirectory contains routines that support the adaptive segmentation method [7] used by *hsfsys2*; *dict* contains routines responsible for dictionary manipulation and matching [17], *fet* is responsible for manipulating Feature (FET) structures and files; and *hblas* contains several basic linear algebra subroutines (blas). If the user's computer system already has a "real" blas library installed, it may be more efficient to compile the test-bed programs by linking the system's library in place of the one provided. The *hsf* library is responsible for form processing with respect to HSF forms, *ihead* contains routines for manipulating IHead structures and files, *image* contains general image manipulation and processing routines; the *mfs* library is responsible for manipulating Multiple Feature Set (MFS) structures and files; *mis* library is responsible for manipulating Multiple Image Set (MIS) structures and files; the *mlp* library holds all the supporting routines for the new MLP classifier [8]; *nn* contains general feature extraction [14] and neural network routines including the PNN classifier [1]; *phrase* holds routines responsible for processing the segmented text from paragraph fields like the Preamble box on HSF forms [6]; *rmline* holds routines that conduct intelligent line removal from forms while preserving character stroke information [5]; the *stats* subdirectory contains general statistics routines; and lastly, *util* contains a collection of miscellaneous routines.

2.4 Automated Compilation Utility

Before compiling the standard reference software test-bed, the full path name to the top-level directory *bin* in the installation directory must be added to your shell's executable search path. For example, if the distribution is installed in */usr/local/hsfsys2*, your search path should be augmented to include */usr/local/hsfsys2/bin*. It may also be necessary to edit the path names contained in a number of files as discussed in Section 2.1.

Compilation of the software in the test-bed is controlled through a system of hierarchical compilation scripts used by the UNIX *make* utility. Each one of these scripts is contained in a file with the root name *makefile*. This auto-

mated compilation system is responsible for installing all architecture-dependent source code files and compilation scripts, removing all compiled object files and development copies of libraries and programs, automatically generating source code dependency lists, and installing production versions of libraries and programs. One *makefile.mak* file exists in the top-level installation directory, and one *makefile.mak* file exists in each of the *src*, *src/bin*, and *src/lib* sub-directories. These compilation scripts are architecture independent and contain Bourne shell commands.

This standard reference software test-bed has been successfully ported and tested on the various UNIX computers listed in Table 2. There are numerous differences between these different computers and their operating systems. Common discrepancies include differences in the syntax of compilation scripts and their built-in macro definitions; some operating systems require manually building the symbol table in archived library files, while other systems update these symbol tables automatically; every one of these operating systems has an *install* command, but each requires its own special set of arguments; each manufacturer's compilers has different options and switches for controlling language syntax and optimization; and so on. To account for these variations, there are architecture-dependent compilation scripts provided for each program and library in the distribution. These compilation scripts have the root file name *makefile* and end with an extension identifying their corresponding architecture. The right column in Table 2 lists the set of extensions used to identify architecture groups for the computers and operating systems tested.

Man.	Model	O.S.	Ext.
DEC	Alpha 3000/400	OSF/1 V1.3	osf
SGI	Indy & Challenge	IRIX 5.3	sgi
IBM	RS6000 Model 370	AIX 4.1	aix
HP	9000/735	HP-UX A.09.05	hp
Sun	SPARCstation 10	SunOS 5.4 (Solaris)	sol
Sun	SPARCstation 2	SunOS 4.1.3	sun

Table 2. Machines tested and their identifying file extensions.

There are also a number of architecture-dependent source code files provided in the distribution. These files share the same root file name and end with an architecture-identifying extension consistent with those used for compilation scripts. There are architecture-dependent source code files provided to support DEC-like machines that use an Intel-based byte order to represent unformatted binary data. All unformatted binary data files provided in this distribution were created on machines using the Motorola-based byte order. When these files are read by a machine using a different byte order, the bytes must be swapped before the data can be used. The overhead of swapping the bytes in these data files can be avoided by regenerating them with locally compiled versions of *mis2evt*, *mis2pat1*, and *mis2pat2* on your computer. The libraries in *src/lib* contain the following architecture-dependent source code files: *image/byte2-bit.{osf,sun}*, *nn/basis_io.{osf,sun}*, *nn/pat_io.{osf,sun}*, *nn/kd_io.{osf,sun}*, *util/ticks.{osf,sun}*, *mlp/getpat.{osf,sun}*, and *mlp/rd_words.{osf,sun}*.

It was stated earlier that the automated compilation system is responsible for installing all architecture-dependent source code files and compilation scripts, removing all compiled object files and development copies of libraries and programs, automatically generating source code dependency lists, and installing production versions of libraries and programs. These tasks are initiated by invoking the *make* command at the top-level installation directory. All subsequent lower-level *makefile.mak* scripts are invoked automatically in a prescribed order, and the 39,000 lines of source code are automatically maintained and object files and executables are kept up to date. The *make* command can be invoked from the location of any lower-level *makefile.mak* file to isolate specific portions of the source code for recompilation. However, the details of doing this are slightly involved and left to the installer to pursue on his own.

The NIST recognition software test-bed is entirely coded in C. Assuming the installation directory is */usr/local/hfsys2*, the following steps are required to compile the distribution for the first time on your UNIX computer:

```
% cd /usr/local/hsfsys2
% make -f makefile.mak instarch INSTARCH=<arch>
% make -f makefile.mak bare
% make -f makefile.mak depend
% make -f makefile.mak install
```

The first *make* invocation uses the *instarch* option to install architecture-dependent files required to support the compilation and execution of the distribution's programs and libraries. The actual architecture is defined by replacing the argument *<arch>* with one of the extensions listed in Table 2. For example, "INSTARCH=sun" must be used to compile the distribution on computers running SunOS 4.1.X. If you are installing this software on a machine not listed in Table 2, you first need to determine which set of architecture-dependent files is most similar to those required by your particular computer. Invoke *make* using the *instarch* option with INSTARCH set to the closest known architecture. Then, edit the resulting *makefile.mak* files in the subdirectories under *src/bin* and *src/lib* according to the requirements of your machine. One other hint, if you are compiling on a Solaris (SunOS 5.X) machine using the parallel *make* utility, you may have to add a "-R" option prior to the "-f" option for each of the *make* invocations.

The *bare* option causes the compilation scripts to remove all temporary, backup, core, and object files from the program directories in *src/bin* and the library directories in *src/lib*. The *depend* option causes the compilation scripts to automatically generate source code dependency lists and modify the *makefile.mak* files within the program and library directories. Your C compiler may not have this capability, in which case you may want to generate the dependency lists by hand. The *install* option builds source code dependency lists as needed, compiles all program and library source code files, and installs compiled libraries and programs into their corresponding production directories. Compiled libraries are installed in the top-level directory *lib*, while compiled programs are installed in the top-level directory *bin*.

One other capability, the automatic generation of catalog files, has been incorporated into the hierarchical compilation scripts. A formatted comment header is included at the top of every program and library source code file in the software test-bed. When the *install* option is used, the low-level *makefile.mak* files invoke the C-shell script *bin/catalog.csh*. The script *catalog.csh* extracts all source code headers associated with all the programs or a specific library in the distribution and compiles a *catalog.txt* file. A *catalog.txt* file exists in the subdirectory *src/bin*, and one *catalog.txt* file exists in each of the library directories in *src/lib*. This provides a convenient and quick reference to the source code provided in the distribution.

3. INVOKING TEST-BED PROGRAMS

This section describes how the programs distributed with this software release (with the exception of *mlp*) are invoked and how they were used to generate the supporting data files provided in the test-bed. The invocation of the off-line neural network training program *mlp* is much more involved, and it can be used for pattern recognition problems other than character classification. Therefore, its description is provided separately in Appendix A. On-line documentation is provided for each of these programs in the form of UNIX-style manual pages under the top-level directory *man*.

3.1 *mis2evt* - computing eigenvector basis functions

Both of the NIST standard reference recognition systems, *hsfsys1* and *hsfsys2*, use the Karhunen Loève (KL) transform to extract features for classifying segmented character images. This transform is obtained by projecting a character image onto eigenvectors of the covariance computed from a set of training images. The mathematical details of the KL transform are provided in Reference [14].

The eigenvectors are computed off-line and stored in a basis function file because computing them from a large covariance matrix is very expensive. The recognition systems read the basis function file during their initialization, and then reuse the eigenvectors across all the character images segmented from fields of a specified type (digit, lowercase, uppercase, or Preamble box). The program *mis2evt* compiles a covariance matrix and then computes its eigenvectors from a set of segmented character images and generates a basis function file. The program's main routine is located in the distribution directory *src/bin/mis2evt*. The command line usage of *mis2evt* is as follows:

```
% mis2evt
Usage: mis2evt:
  -n  for 128×128 input, write normed+sheared 32×32 intermediate MIS files
  -v  be verbose - notify completion of each misfile
  <nrequiredevts> <evtfile> <mfs_of_misfiles>
```

Arguments:

- The first argument *nrequiredevts* specifies the number of eigenvectors to be written to the output file. It is also the number of KL features that will ultimately be extracted from each binary image using the associated utilities *mis2pat1* and *mis2pat2*. This integer determines the dimensionality of the feature vectors that are produced for classification. Its upper bound is the image dimensionality (which is $32 \times 32 = 1024$). Typically, this argument is specified to be much smaller than 1024 because the KL transform optimally compacts the representation of the image data into its first few coefficients (features). *Hsfsys1* uses a value of 64, while *hsfsys2* uses 128. Reference [22] documents an investigation of the dependency of classification error on feature vector dimensionality.
- The second argument *evtfile* specifies the name of the output basis function file. The format of this file is documented by the routine *write_basis()* found in *src/lib/nn/basis_io.c*.
- The third argument *mfs_of_misfiles* specifies a text file that lists the names of all the MIS files containing images that will be used to calculate the covariance matrix. This argument is an MFS file with the first line containing an integer indicating the number of MIS files that follow. The remaining lines in the MFS file contain MIS file names, one name per line. The format of an MFS file is documented by the routine *writemfsfile()* found in *src/lib/mfs/writemfs.c*.

Options:

- The option “-n” specifies the storing of intermediate normalized character images. *Mis2evt* can process binary images that are either (128×128) or (32×32). In the case of the former, the program invokes a size normalization utility to produce 32×32 images and then applies a shear transformation to reduce slant variations. If the input images are already 32×32, this flag has no effect. If normalization does occur, the resulting normalized images are stored to MIS files having the same name as those listed in the MFS file, with the

additional extension *32* appended. These intermediate files offer computational gains because usually the same images are used with *mis2pat1* and *mis2pat2*.

- The option “-v” produces messages to standard output signifying the completion of each MIS file and other computation steps.

This program is computationally expensive and may require as long as 60 minutes to compute the eigenvectors for a large set (50,000 characters) of images. The program *mis2evt* was used to generate the basis function files provided with this distribution in the top-level directory *weights* and ending with the extension *evt*. These files contain eigenvectors computed from the images provided in the top-level directory *train*. The MFS files used as arguments to *mis2evt* are also provided in *weights* and end with the extension *ml*. For example, the basis function file *weights/pnn/lower/h46_1.evt* was generated with the following command:

```
% mis2evt -v 64 h46_1.evt h46_1.ml
```

The basis function file *weights/mlp/digit/h6_d.evt* was generated with the following command:

```
% mis2evt -v 128 h6_d.evt h6_d.ml
```

3.2 *mis2pat1* - generating patterns for the PNN classifier

Mis2pat1 is algorithmically equivalent to the program *mis2pat* distributed with the first software release. It takes a set of training images along with the eigenvectors generated by *mis2evt* and computes feature vectors using the KL transform that can be used as prototypes for training the PNN classifier used in *hsfsys1*. Typically, the same images used to compute the eigenvectors are used here to generate prototype vectors. The program *mis2pat1* also builds a kd-tree as described in Reference [20]. The prototypes along with their class assignments and kd-tree are stored in one patterns file, while median vectors computed from the prototype vectors are stored in a separate median vector file. Note that all FORTRAN dependencies have been removed from this release. In doing so, the format of the patterns file generated by *mis2pat1* has changed from that generated by the original program *mis2pat*. The main routine for *mis2pat1* is located in *src/bin/mis2pat1*. The command line usage of *mis2pat1* is as follows:

```
% mis2pat1
Usage: mis2pat1:
  -h  accept hexadecimal class files
  -n  with 128×128 images, write normed+sheared 32×32 intermediate MIS files
  -v  be verbose - notify completion of each misfile
<classset> <evtfile> <outroot> <mfs_of_clsfiles> <mfs_of_misfiles>
```

Arguments:

- The first argument *classset* specifies the name of a text file (MFS file) containing the labels assigned to each class. The integer on the first line of the file indicates the number of classes following, and the remaining lines contains one class label per line. For example, a digit classifier uses ten classes labeled 0 through 9. The format of an MFS file is documented by the routine *writemfsfile()* found in *src/lib/mfs/writemfs.c*.
- The second argument *evtfile* specifies the basis function file containing eigenvectors computed by *mis2evt*. The number of features in each output vector is determined by the number of eigenvectors in this file. The format of this file is documented by the routine *write_basis()* found in *src/lib/nn/basis_io.c*.
- The third argument *outroot* specifies the root file name of the output pattern and median vector files. The name of the output pattern file has extension *pat* while the median vector file has extension *med*. The format of the patterns/kd_tree file is documented by the routine *kdtreewrite()* in *src/lib/nn/kd_io.c* whereas the median vector file format is documented by the routine *writemedianfile()* in *src/lib/nn/med_io.c*.
- The final arguments are the names of text files (MFS files) that contain listings of file names. The argument *mfs_of_clsfiles* lists file names containing class assignments corresponding to the images in the MIS files listed in the argument *mfs_of_misfiles*. Each class assignment file must have the same number of class assignments as there are images in its corresponding MIS file, and the classes assigned must be consistent with those listed in the argument *classset*.

Options:

- The option “-h” specifies that the class labels listed in the *classset* file are to be converted to ASCII character values represented in hexadecimal. All the class assignments in the files listed in the argument *mfs_of_clsfiles* use the convention where [30-39] represent digits, [41-5a] represent uppercase, and [61-7a] represent lowercase. If the *classset* file contains alphabetic representations such as [0-9], [A-Z], and [a-z], then this flag must be used to effect conversion of these labels to their hexadecimal equivalents.
- The option “-n” specifies the storing of intermediate normalized character images. *Mis2pat1* can process binary images that are either (128×128) or (32×32). In the case of the former, the program invokes size and slant normalization utilities to produce 32×32 images. If the input images are already 32×32, this flag has no effect. If normalization does occur, the resulting normalized images are stored to MIS files having the same name as those listed in *mfs_of_misfiles*, with the extension *32* appended.
- The option “-v” produces messages to standard output signifying the completion of each MIS file.

This program was used to generate the patterns files provided with this distribution in the directory *weights/pnn* and ending with the extension *pat* and median vector files ending with extension *med*. The patterns files contain KL feature vectors, their associated classes, and a kd-tree in its *new* format as documented by the routine *kdtreewrite()* found in *src/lib/nn/kd_io.c*. The feature vectors were computed using the eigenvectors found in the same directory and from the images provided in the top-level directory *train*. The MFS files used as arguments to *mis2pat1* are also provided in the *weights/pnn* subdirectories, as are the *classset* files which end with the extension *set*. The class assignment files are listed in files ending with the extension *cl*, whereas the MIS files are listed in files ending with the extension *ml*. For example, the patterns file *weights/pnn/lower/h46_1.pat* and median vector file *weights/pnn/lower/h46_1.med* were generated with the following command:

```
% mis2pat1 -vh l.set h46_1.evt h46_1 h46_1.cl h46_1.ml
```

3.3 *hsfsys1* - running the updated version of the original NIST system

Hsfsys1 is an updated version of the NIST recognition system distributed in the first release of the software. This system is designed to read the handwriting entered on HSF forms like those included in the top-level directory *data*. The most significant changes to this system include more efficient memory usage (improving recognition speed), and all dependencies on FORTRAN-coded subroutines have been removed. A detailed description of the algorithms used in this system is provided in the original user's guide (NISTIR 5469) [1].

The recognition system is run in batch mode with image file inputs and ASCII text file outputs, and the system contains no Graphical User Interface. The command line usage of *hsfsys1* is as follows:

```
% hsfsys1
Usage:
  hsfsys1 [options] <hsf file> <output root>
  -d                process digit fields
  -l                process lowercase fields
  -u                process uppercase fields
  -c nodict         process Constitution field without dictionary
  -c dict           process Constitution field using dictionary
  -m                small memory mode
  -s                silent mode
  -v                verbose mode
  -t                compute and report timings
```

The command line arguments for *hsfsys1* are organized into option specifications, followed by an input file name specification, and an output (root) file name specification. The options can be subgrouped into three categories (field type options, memory control options, and message control options).

Field type options:

- d** designates the processing of the digit fields on an HSF form.
- l** designates the processing of the lowercase field on an HSF form.
- u** designates the processing of the uppercase field on an HSF form.
- c** designates the processing of the Constitution field on an HSF form. This option requires an argument. If the argument *nodict* is specified, then no dictionary-based postprocessing is performed and the raw character classifications and associated confidence values are reported. If the argument *dict* is specified, then dictionary-based postprocessing is performed and matched words from the dictionary are reported without any confidence values.

The options **-dluc** can be used in any combination. For example, use only the **-l** option to process the lowercase field, or use only the **-d** option to process all of the digit fields. If processing both lowercase and uppercase fields, then specify both options **-l** and **-u** (or an equivalent syntax **-lu**). The system processes all of the fields on the form if no field type options are specified, and dictionary-based postprocessing is performed on the Constitution field by default.

Memory control options:

- m** specifies the use of alternative prototype files for classification that have fewer training patterns, so that machines with limited main memory may be able to completely process all the fields on an HSF form. In general, decreasing the number of training prototypes reduces the accuracy of the recognition system's classifier. It is recommended that this option be used only if necessary.

Message control options:

- s** specifies that the silent mode is to be used and all messages sent to standard output and standard error are suppressed except upon the detection of a fatal internal error. Silent mode facilitates silent batch processing and overrides the verbose mode option. By default, the system posts its recognition results to standard output as each field is processed.
- v** specifies that the verbose mode is to be used so that messages providing a functional trace through the system are printed to standard error.
- t** specifies that timing data is to be collected on system functions and reported to a timing file upon system completion.

File name specifications:

- <hsf file>** specifies the binary HSF image in IHead format that is to be read by the system. The IHead file format is documented by the routine *ReadBinaryRaster()* found in *src/lib/image/readrast.c*.
- <output root>** the root file name that is to be appended to the front of each output file generated by the system. Upon completion, the system will create a hypothesis file with the extension *hyp* and a confidence file with the extension *con*. If the **-t** option is specified, a timing file with the extension *tim* will also be created. These text files can be manipulated as FET files, the format of which is documented by the routine *writefetfile()* in *src/lib/fet/writefet.c*.

For example, to run the system in verbose mode on all the HSF fields on the form in *data/f0000_14* and store the system results in the same location with the same root name as the form, the following commands are equivalent (assuming the installation directory is */usr/local/hsfsys2*). In each case, the files *f0000_14.hyp* and *f0000_14.con* will be created by the system in the directory */usr/local/hsfsys2/data/f0000_14*.

```
% hsfsys1 -v /usr/local/hsfsys2/data/f0000_14/f0000_14.pct /usr/local/hsfsys2/data/f0000_14/f0000_14
% hsfsys1 -v /usr/local/hsfsys2/data/f0000_14/f0000_14{.pct,}
% (cd /usr/local/hsfsys2/data/f0000_14; hsfsys1 -v f0000_14.pct f0000_14)
```

To run the system in silent mode on only the digit and uppercase fields on the same form with results including timing data all stored in */tmp* with the root name *foo*, the following command can be used.

```
% hsfsys1 -stdu /usr/local/hsfsys2/data/f0000_14/f0000_14.pct /tmp/foo
```

In this example, the files created by the system will be */tmp/foo.hyp*, */tmp/foo.con*, and */tmp/foo.tim*.

The program *hsfsys1* was used to generate the files with extension *hyp*, *con*, and *tim* located within the form subdirectories under the top-level directory *data*.

3.4 *mis2pat2* - generating patterns for training the MLP classifier

Mis2pat2 takes a set of training images along with the eigenvectors generated by *mis2evt* and computes feature vectors using the KL transform that can be used as prototypes for training the MLP classifier used in *hsfsys2*. Typically, the same images used to compute the eigenvectors are used here to generate prototype vectors. The prototypes along with their class assignments are stored in a patterns file that is of a different format than those generated by *mis2pat1*. The format of the patterns file created by *mis2pat2* is documented in the routine *write_bin_patterns()* found in *src/lib/nn/pat_io.c*. The program's main routine is located in *src/bin/mis2pat2*. The command line usage of *mis2pat2* is as follows:

```
% mis2pat2
Usage: mis2pat2:
  -h  accept hexadecimal class files
  -n  with 128×128 images write normed+sheared 32×32 intermediate MIS files
  -v  be verbose - notify completion of each misfile
<classset> <evtfile> <outfile> <mfs_of_clsfiles> <mfs_of_misfiles>
```

Arguments:

- The first argument *classset* specifies the name of a text file (MFS file) containing the labels assigned to each class. The integer on the first line of the file indicates the number of classes following, and the remaining lines contains one class label per line. For example, a digit classifier uses ten classes labeled 0 through 9. The format of an MFS file is documented by the routine *writemfsfile()* found in *src/lib/mfs/writemfs.c*.
- The second argument *evtfile* specifies the basis function file containing eigenvectors computed by *mis2evt*. The number of features in each output vector is determined by the number of eigenvectors in this file. The format of this file is documented by the routine *write_basis()* found in *src/lib/nn/basis_io.c*.
- The third argument *outfile* specifies the file name of the output patterns file. The format of this patterns file is documented by the routine *write_bin_patterns()* in *src/lib/nn/pat_io.c*.
- The final arguments are the names of text files (MFS files) that contain listings of file names. The argument *mfs_of_clsfiles* lists file names containing class assignments corresponding to the images in the MIS files listed in the argument *mfs_of_misfiles*. Each class assignment file must have the same number of class assignments as there are images in its corresponding MIS file, and the classes assigned must be consistent with those listed in the argument *classset*.

Options:

- The option “-h” specifies that the class labels listed in the *classset* file are to be converted to ASCII character values represented in hexadecimal. All the class assignments in the files listed in the argument *mfs_of_clsfiles* use the convention where [30-39] represent digits, [41-5a] represent uppercase, and [61-7a] represent lowercase. If the *classset* file contains alphabetic representations such as [0-9], [A-Z], and [a-z], then this flag must be used to effect conversion of these labels to their hexadecimal equivalents.
- The option “-n” specifies the storing of intermediate normalized character images. *Mis2pat2* can process binary images that are either (128×128) or (32×32). In the case of the former, the program invokes size and slant normalization utilities to produce 32×32 images. If the input images are already 32×32, this flag has no effect. If normalization does occur, the resulting normalized images are stored to MIS files having the same name as those listed in *mfs_of_misfiles*, with the extension *32* appended.
- The option “-v” produces messages to standard output signifying the completion of each MIS file.

This program was used to generate the patterns files provided with this distribution in the directory *weights/mlp* and ending with the extension *pat*. These patterns files contain KL feature vectors along with their associated classes. The feature vectors were computed using the eigenvectors found in the same directory and from the images provided in the top-level directory *train*. The MFS files used as arguments to *mis2pat2* are also provided in the *weights/*

mlp subdirectories, as are the *classset* files which end with the extension *set*. The class assignment files are listed in files ending with the extension *cl*, whereas the MIS files are listed in files ending with the extension *ml*. For example, the patterns file *weights/mlp/lower/h46_1.pat* was generated with the following command:

```
% mis2pat2 -vh l.set h46_1.evt h46_1.pat h46_1.cl h46_1.ml
```

3.5 *trainreg* - training to register a new form

The new recognition system, *hsfsys2*, uses a generalized method of form registration described in Reference [4]. The technique locates the most dominant left and right, top and bottom lines on the form image and then transforms the image so that these form structures are positioned at registered coordinates. To accomplish this, dominant lines on a new form must be determined and the coordinates of their registered position must be measured and stored. The program *trainreg* does this automatically, storing the resulting x-coordinates of the left and right-most dominant lines on the form and the y-coordinates of the top and bottom-most dominant lines on the form. The program's main routine is located in the distribution directory *src/bin/trainreg*, and its main supporting subroutine is found in *src/lib/hsf/reg-form.c*. The command line usage of *trainreg* is as follows:

```
% trainreg <form_image> <out_points>
```

- The first argument *form_image* specifies the name of the input file containing the image of the new form. This image must be in the binary (black and white) IHead file format which is documented in Reference [1] and by the routine *ReadBinaryRaster()* found in *src/lib/image/readrast.c*.
- The second argument *out_points* specifies the name of the output file to hold the coordinate positions of the detected dominant lines in the image. This is an MFS file, the format of which is documented by the routine *writemfsfile()* found in *src/lib/mfs/writemfs.c*.

This program was used to generate the file *tmplt/hsfgreg.pts*, which is used by *hsfsys2* to register input HSF forms. This file was created with the following command:

```
% trainreg hsfmplt.pct hsfgreg.pts
```

3.6 *hsfsys2* - running the new NIST recognition system

The new recognition system, *hsfsys2*, contains significant technical improvements over its predecessor, *hsfsys1*. It uses new methods of generalized form registration [4], intelligent form removal [5], line isolation within hand-printed paragraphs [6], adaptive character segmentation [7], a new robust MLP-based classifier [8], among other improved techniques which are described in Section 4. *Hsfsys2* is designed to read the handwriting entered on HSF forms like those included in the top-level installation directory *data*, and it is capable of reading every HSF form included in SD19.

The recognition system is run in batch mode with image file inputs and ASCII text file outputs, and the system contains no Graphical User Interface. The command line usage of *hsfsys2* is as follows:

```
% hsfsys2
Usage:
  hsfsys2 [options] <list file>
  -d                process digit fields
  -l                process lowercase fields
  -u                process uppercase fields
  -c nodict         process Constitution field without dictionary
  -c dict           process Constitution field using dictionary
  -s                silent mode
  -v                verbose mode
  -t <time file>   compute and report timings
```

The command line arguments for *hsfsys2* are organized into option specifications and a file containing multiple pairs of input file name and output (root) file name specifications. The options can be subgrouped into two general types (field type options and message control options).

Field type options:

- d** designates the processing of the digit fields on an HSF form.
- l** designates the processing of the lowercase field on an HSF form.
- u** designates the processing of the uppercase field on an HSF form.
- c** designates the processing of the Constitution field on an HSF form. This option requires an argument. If the argument *nodict* is specified, then no dictionary-based postprocessing is performed and the raw character classifications and associated confidence values are reported. If the argument *dict* is specified, then dictionary-based postprocessing is performed and matched words from the dictionary are reported without any confidence values.

The options **-dluc** can be used in any combination. For example, use only the **-l** option to process the lowercase field, or use only the **-d** option to process all of the digit fields. If processing both lowercase and uppercase fields, then specify both options **-l** and **-u** (or an equivalent syntax **-lu**). The system processes all of the fields on the form if no field type options are specified, and dictionary-based postprocessing is performed on the Constitution field by default.

Message control options:

- s** specifies that the silent mode is to be used and all messages sent to standard output and standard error are suppressed except upon the detection of a fatal internal error. Silent mode facilitates silent batch processing and overrides the verbose mode option. By default, the system posts its recognition results to standard output as each field is processed.

- v specifies that the verbose mode is to be used so that messages providing a functional trace through the system are printed to standard error.
- t <time file> specifies that timing data is to be collected on system functions and reported to the specified timing file upon system completion.

File name specification:

<list file> is a two column ASCII file. The first column lists all the binary HSF images in IHead format that are to be read by the system in the current batch. The IHead file format is documented by the routine *ReadBinaryRaster()* found in *src/lib/image/readrast.c*. Along with each input image file is a second argument that specifies the root file name to be appended to the front of each output file generated by the system when processing the corresponding form image. An example of a list file is found in *data/hsfsys2.lis*. Upon completion of each form, the system will create a hypothesis file with extension *hyp* and a confidence file with extension *con*. These output text files can be manipulated as FET files, the format of which is documented by the routine *writefetfile()* in *src/lib/fet/writefet.c*.

Assuming the installation directory is */usr/local/hsfsys2*, the following commands can be used to run the new system in verbose mode on all the fields on all the HSF forms in *data*.

```
% cd /usr/local/hsfsys2/data
% hsfsys2 -v -t hsfsys2.tim hsfsys2.lis
```

The program *hsfsys2* was used to generate the timing file *hsfsys2.ti2* and the output files with extension *hy2* and *co2* located under the top-level directory *data*.

4. ALGORITHMIC OVERVIEW OF NEW SYSTEM HSFSYS2

Reference [2] describes the complexities of integrating various technology components into a successful OCR system. Very little can be found in the literature published on the internal workings of complete systems. Granted, many of the technologies required for successful OCR have been researched and results have been published, but these components are typically tested in isolation and their impact on overall recognition is not measured. Also, many of the algorithms implemented in an end-to-end system are proprietary. Companies disclose research results on pieces of their recognition systems, but no current publications can be found that disclose the details of a completely operational system.

In contrast, NIST has developed a completely open recognition software test-bed for which the components are fully disclosed, and in fact, the source code is publicly available. This software provides a baseline of performance with which new and promising technologies can be compared and evaluated. This section describes the application for which the new system, *hsfsys2*, was designed and provides a high-level description of the algorithms used in each of the system's major components. An algorithmic overview of the updated original recognition system, *hsfsys1*, can be found in Reference [1].

4.1 The Application

It was already noted that the successful application of OCR technology requires more than off-the-shelf system integration. State-of-the-art solutions require customization and tuning to the problem at hand. This being true, an operational system is largely defined by the details of the application it is to solve.

The NIST system is designed to read the handprinted characters written on Handwriting Sample Forms (HSF). An example image of a completed HSF form is displayed in Figure 1 on page 4. This form was designed to collect a large sample of handwriting to support handprint recognition research. A CD-ROM named *NIST Special Database 19* (SD19), containing 3669 completed forms, each filled in by a unique writer, and scanned binary at 11.8 pixels per millimeter (300 pixels per inch), is publicly available [3]. This data set also contains over 800,000 segmented and labeled characters images from these forms. There are 10 completed HSF forms provided with this software test-bed. In addition, there is one blank form provided both in Latex and PostScript formats that can be printed, filled in, scanned, and then recognized. For additional HSF forms, SD19 may be purchased by contacting:

Standard Reference Data Program
National Institute of Standards and Technology
NIST North (820), Room 113
Gaithersburg, MD 20899
voice: (301) 975-2208
FAX: (301) 926-0416
email: srdata@enh.nist.gov

The new NIST system is designed to read all but the top line of fields on the form. The system processes the 28 digit fields and the randomly ordered lowercase and uppercase alphabet fields along with the handprinted paragraph of the Preamble to the U.S. Constitution at the bottom of the form.

4.2 System Components

Figure 4 contains a diagram that illustrates the organization of the functional components within the new system *hsfsys2*. Generally speaking, each one of these components has many possible algorithmic solutions. Therefore, the new system is designed in a modular fashion so that different methods can be evaluated and compared within the context of an end-to-end system. This section provides a brief description of the most recent techniques developed by NIST for each of these tasks. Listed with each component subheading is a path name followed by a subroutine name. These listings are provided to guide the reader to specific areas of the software test-bed for further study and investigation.

NIST Form-Based Recognition System *Hsfsys2*

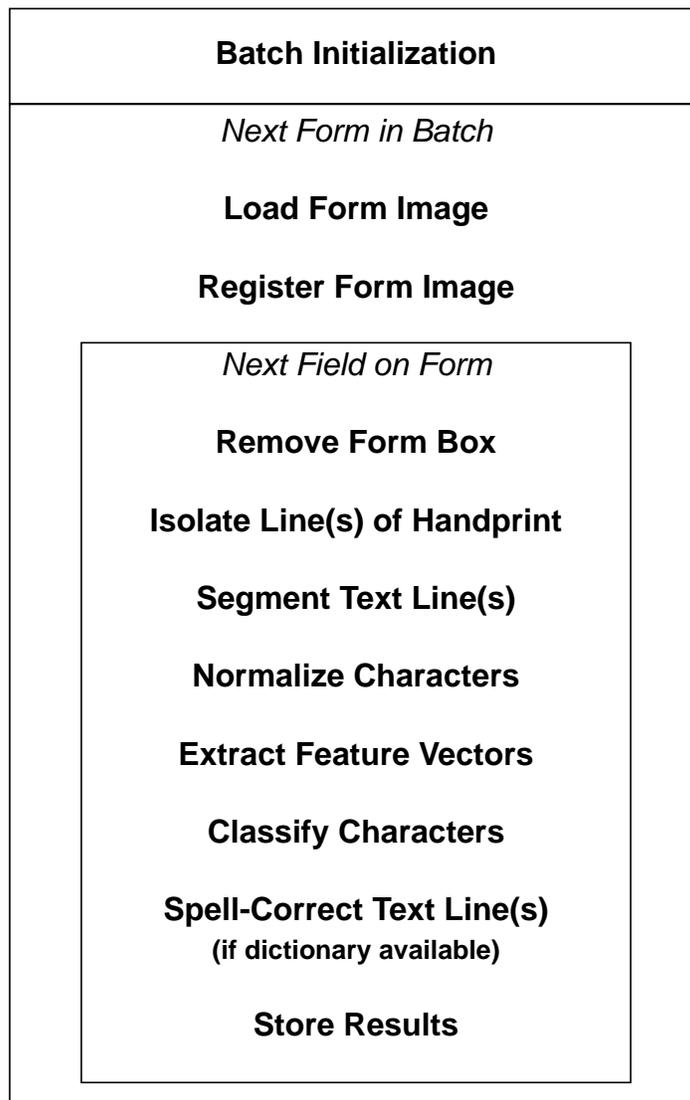


Figure 4. Organization of the functional components within the new recognition system.

4.2.1 Batch Initialization; `src/lib/hsf/run.c; init_run()`

The new system is a non-interactive batch processing system designed to process one or more images of completed HSF forms with each invocation. The first step loads all the precomputed items required to process a particular type of form (in this case HSF forms). These items include a list of the image files to be processed in a batch, prototypical coordinate locations of dominant form structures used for form registration, a spatial template containing the prototypical coordinate location of each field on the form, basis functions used for feature extraction, neural network weights for classification, and dictionaries for spelling correction. There are four types of fields on the HSF form: numeric, lowercase, uppercase, and the Preamble paragraph. Each type of field requires a separate set of basis functions and neural network weights. Only the Preamble paragraph has a dictionary available. The use of these items will be explained in more detail later.

Because the new system only processes HSF forms, form identification is not utilized. Form identification can be avoided for any application when it is economically feasible to sort forms (whether automatically or manually) into homogeneous batches. Unfortunately, this is not practical for all applications.

4.2.2 Load Form Image; src/lib/image/readrast.c; ReadBinaryRaster()

The new system is strictly an off-line recognition system, meaning that the time at which images are scanned is independent from when recognition takes place. This is typical of large-scale OCR applications where operators work in shifts running high-speed scanners that archive images of forms to mass-storage devices for later batch conversion. For each form in the batch, the new system reads a CCITT Group 4 compressed binary raster image from a file on disk, decompresses the image in software, and passes the bitmap along with its attributes on to subsequent components.

4.2.3 Register Form Image; src/lib/hsf/regform.c; genregform8()

A considerable amount of image processing must take place in order to reliably isolate the handprint on a form. The form must be registered or aligned so that fields in the image correspond with the prototypical template of fields (or zones). The new system uses a generalized method of form registration that automatically estimates the amount of rotation and translation in the image without any detailed knowledge of the form [4].

To measure rotational distortion, a technique similar to the one invented by Postl is used [21]. This technique traces parallel rays across the image accumulating the number of black pixels along each ray using a non-linear function. A range of ray angles are sampled, and the angle producing the maximum response is used to estimate the rotational skew. The image is rotated based on this estimate, and it is then analyzed to detect any translational distortion. This step capitalizes on the fact that most forms contain a fixed configuration of vertical and horizontal lines. Once the rotational skew is removed, these lines correspond well with the raster grid of the image. A run-based histogram is computed to detect the top and bottom, left and right, dominant lines in the image.

For example, to locate the top and bottom-most dominant lines, the horizontal runs in the image are computed. The n -longest runs (in the new system, $n=3$) on each scanline of the image are accumulated into a histogram bin. These bins are then analyzed for relative maxima as described in Reference [4]. The accumulation of the n -longest runs effectively suppresses regions of the form containing handwriting and noise, and accentuates the lines on the form. The same analysis is conducted on vertically-oriented runs to locate the left and right-most dominant lines. Given the locations of these lines, translation estimates in x and y are computed with respect to the coordinates of prototypical lines, and the image is translated accordingly. At this point, fields in the image correspond to the coordinates of the prototypical spatial template.

By using this general registration technique, new form types can be trained automatically. A prototypical form is scanned, its rotational distortion is automatically measured and removed, and the position of the detected dominant lines are stored for future registrations. The results of registering 500 HSF forms is shown in Figure 5. The image displayed is the result of logically ORing corresponding pixels across a set of 500 registered images. Notice the tight correspondence of the boxes and the printed instructions.

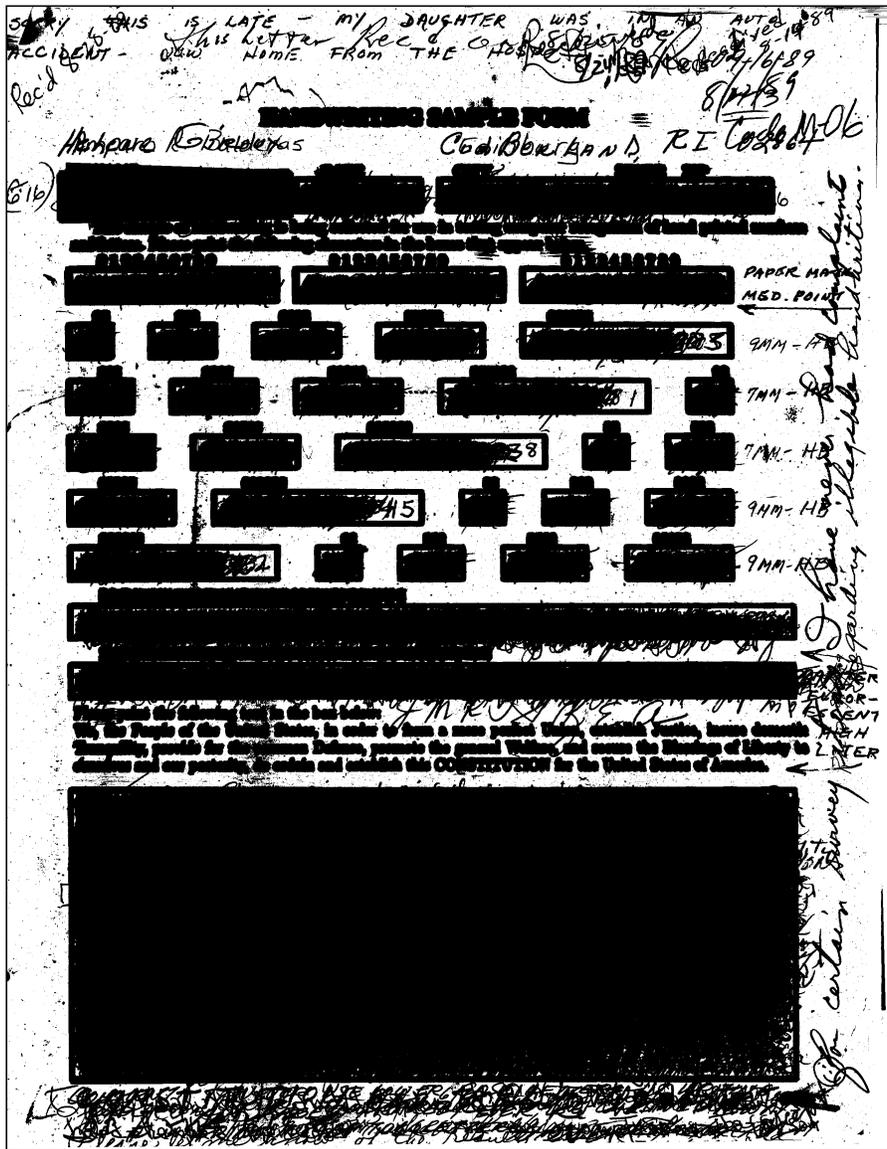
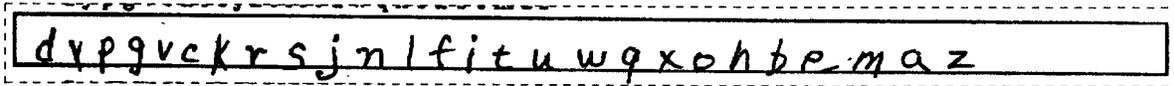


Figure 5. Composite image of 500 registered HSF forms logically ORed together.

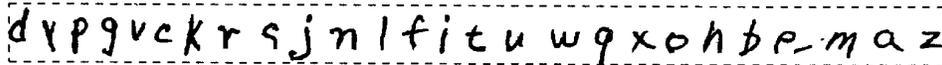
4.2.4 Remove Form Box; src/lib/rmline/remove.c; rm_long_hori_line()

Upon registration, a spatial template is used to extract a subimage of each field on the form. Fields are extracted and processed one at a time. Given a field subimage, black pixels corresponding to the handwriting must be separated from the black pixels corresponding to the form. This is a difficult task because a black pixel can represent handwriting, the form, or an overlap of both. As all the fields on the HSF form are represented by boxes, the new system uses a general algorithm that locates the box within the field subimage, and intelligently removes the sides so as to preserve overlapping characters [5]. The sides of the box are detected using a run-based technique that tracks the longest runs across the subimage. Then, by carefully analyzing the width of the sides of the box, overlapping character strokes are identified using spatial cues, and only pixels that are distinctly part of the form's box are removed. This way, descenders of lowercase characters, for example, are not unnecessarily truncated. Figure 6 shows two fields before and after form removal.

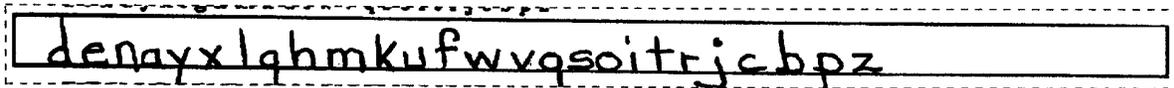
Field Subimage (A)



Isolated Handprint



Field Subimage (B)



Isolated Handprint

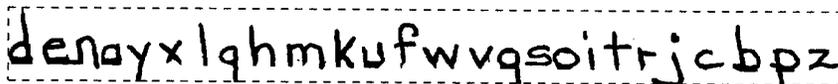


Figure 6. Results of form box removal.

4.2.5 Isolate Line(s) of Handprint; `src/lib/phrase/phrasemap.c; phrases_from_map()`

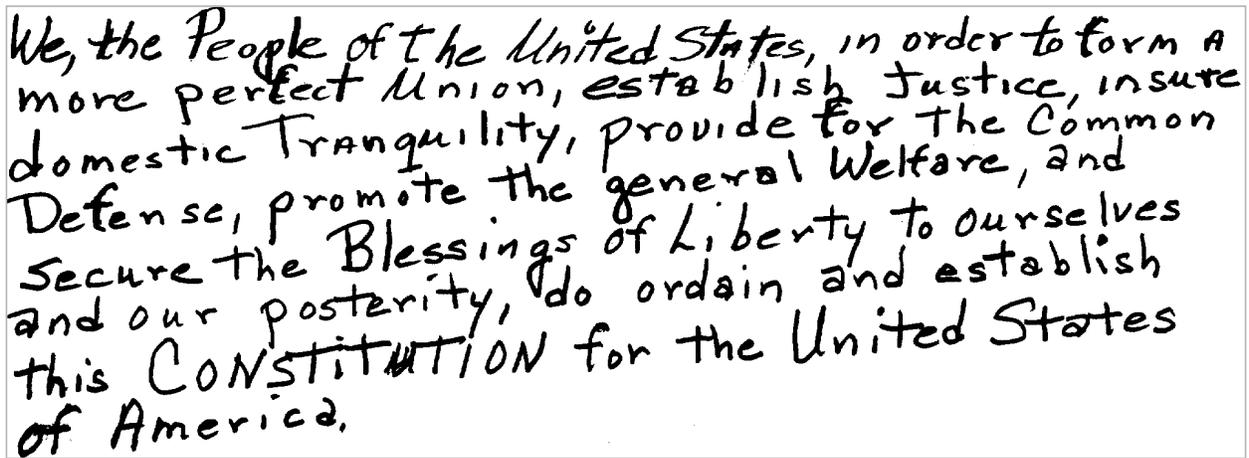
The numeric and alphabetic fields on an HSF form are written as single-line responses. After the box is removed, the handprint contained in a field is isolated (or lifted out) by simply extracting all the connected components that overlap the interior region of the detected box. A connected component is defined as the largest set of black pixels where each pixel is a direct neighbor of at least one other black pixel in the component. Single isolated black pixels are also considered components.

Line isolation is much more difficult for multiple-line responses such as the handprinted paragraph of the Preamble at the bottom of the HSF form. There are no lines provided within this paragraph box to guide a writer, nor are there any instructions of how many words should be written on a line. The handwriting is relatively unconstrained, and as a result, the baselines of the writing at times significantly fluctuate. This, along with the fact that the paragraph contains handprinted punctuation marks, makes tracking the lines of handprint difficult. Histogram projections (used extensively for isolating lines of machine printed characters) are rendered unreliable in this case.

The new system uses a bottom-up approach to isolate the lines of handprint within a paragraph. This technique starts by decomposing a paragraph into a set of connected components. Each component is represented by its geometric center. To reconstruct the handprinted lines of text, a nearest neighbor search is performed left-to-right and top-to-bottom through the system of 2-dimensional points [9]. The search is horizontally biased and links sequences of points into piecewise-linear segments. Simple statistics are used to sort components into categories of too small, too tall, problematic, and normal. Only those components determined to be normal are linked together by the search.

Given these piecewise-linear trajectories, the tops and bottoms of linked components are interpolated and smoothed forming line bands. An example of these bands is shown in Figure 7. These bands form a spatial map, and

all the components in the image are sorted into their respective lines in correct reading order according to their overlap and/or proximity to these bands [6]. At this point, the handwriting in the paragraph is isolated into individual text lines.



We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the Common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.

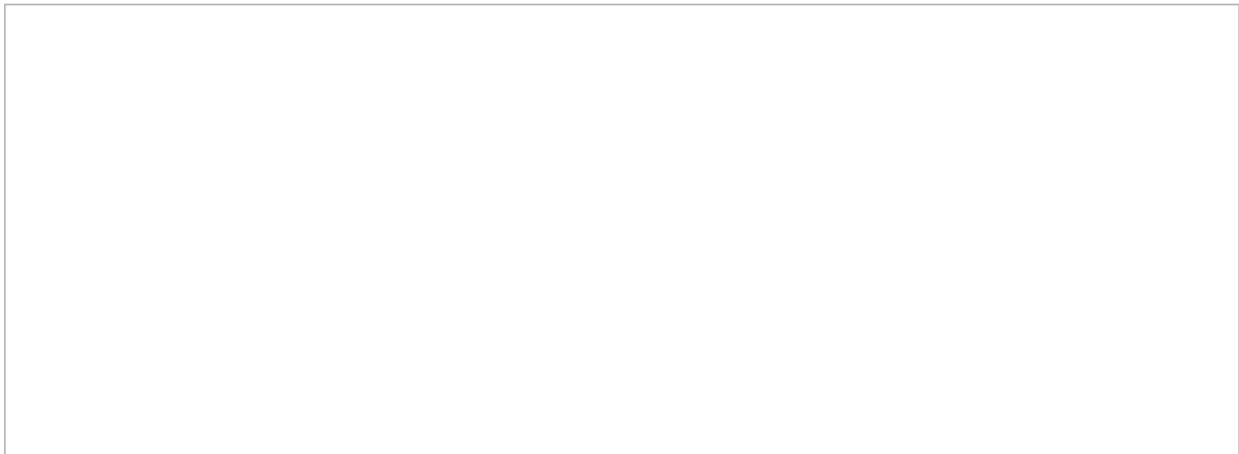


Figure 7. Line-bands computed from the paragraph image above.

4.2.6 Segment Text Line(s); src/lib/adseg/segchars.c; blobs2chars8()

Connected components are used as first-order approximations to single and complete characters. Connected components frequently represent single characters and are computed very quickly. On the other hand, their direct use as character segments is prone to error. Errors occur when characters touch one another and when characters are written with disconnected strokes (naturally occurring with dotted letters). The new system was initially designed to process the numeric fields on HSF forms. Numeric fields typically do not have any linguistic context; therefore, the utility of oversegmentation schemes is severely compromised in this case.

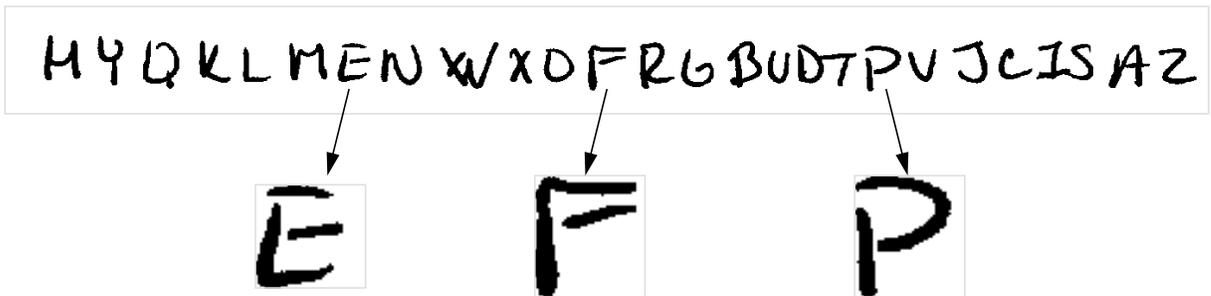
Building upon the utility of connected components, the new system utilizes a method of handprint character segmentation that uses a simple adaptive model of writing style [7]. Using this model, fragmented characters are reconstructed, multiple characters are split, and noise components are identified and discarded. Visual features are measured (the width of the pen stroke and the height of the characters) and used by fuzzy rules, making the method robust. Examples of segmentation results are illustrated in Figure 8 and Figure 9. The segmentor performs best when applied to single-line responses, and then even better when the fields are numeric.

With minor modification, the same method is used to segment the isolated lines extracted from paragraphs of handprinted text as described in Reference [6].

Broken Characters



Detached Strokes

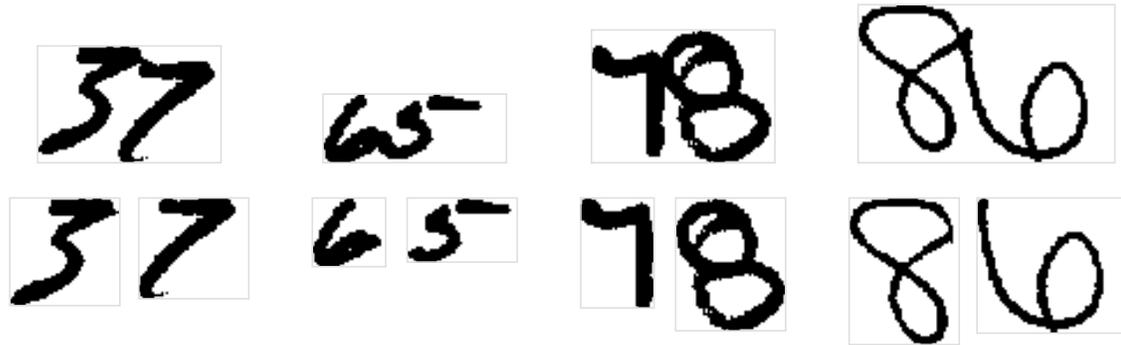


Dotted Characters

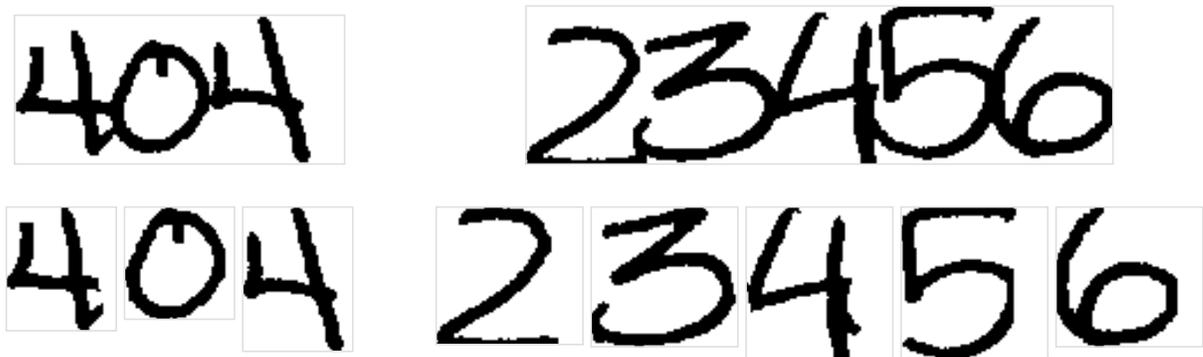


Figure 8. Segmentor results of merging components together.

Two Characters Touching



More Than Two Characters Touching



Uppercase Characters Touching



Figure 9. Segmentor results of splitting components apart.

4.2.7 Normalize Characters; src/lib/hst/norm8.c; norm_2nd_gen_blobs8()

The recognition technique used by the new system falls under the category of feature-based pattern classification. The segmented character images vary greatly in size, slant, and shape. Image normalization is performed to deal with the size and slant of writing, leaving the recognition process primarily the task of differentiating characters by variation in shape.

The segmented character images are size-normalized by scaling the image either up or down so that the character tightly fits within a 20×32 pixel region. The stroke width is also normalized using simple morphology: if the pixel content of the character image is too high, it is eroded (strokes are thinned), and if too low, it is dilated (strokes are widened).

Slant is removed by interpolating a line between the top left-most black pixel in the scaled image and the bottom left-most black pixel. The line (centered on the image) is used as a horizontal shear function. The slant of the character is removed as horizontal rows of pixels in the image are increasingly shifted (left or right) outwards from the center of the image. Upon normalization, each character is centered in a 32×32 pixel image. Size and slant normalization are discussed in greater detail in Reference [1].

4.2.8 Extract Feature Vectors; `src/lib/nn/kl.c`; `kl_transform()`

At this point, characters are represented by 1024 binary pixel values. The Karhunen Loève (KL) transform is applied to these binary pixel vectors in order to reduce dimensionality, suppress noise, and produce optimally compact features (in terms of variance) for classification [14].

A training set of normalized character images is used to compute a covariance matrix which is diagonalized using standard linear algebra routines, producing eigenvalues and corresponding eigenvectors. This computation is relatively expensive, but is done once off-line, and the top-*n* ranked eigenvectors are stored as basis functions and used subsequently for feature extraction. Feature vectors of length 128 are used in the new system, and each coefficient in the vector is the dot product of a subsequent eigenvector with the 1024 pixel vector of the character being classified.

4.2.9 Classify Characters; `src/lib/mlp/runmlp.c`; `mlphyscons()`

Once segmented characters are represented as feature vectors, a whole host of different pattern classification techniques can be applied. NIST has conducted extensive research on classification methods that utilize machine learning, and most of these have been various types of neural networks. In previous work, the Probabilistic Neural Network (PNN) was shown to provide better zero-reject error performance on character classification problems than Radial Basis Function (RBF) and Multi-Layer Perceptron (MLP) neural network methods [22]. Later work demonstrated that various combined neural networks could provide performance equal to PNN and substantially better error-reject performance. However, these systems were very expensive to train and were much slower and less memory efficient than MLP-based systems [23].

NIST has developed a robust training method that produces MLP networks with performance equal to or better than PNN for character recognition [8]. This is achieved with a single three-layer network by integrating fundamental changes in the network optimization strategy. These changes are: 1) Sinusoidal neuron activation functions are used which reduce the probability of singular Jacobians; 2) Successive regularization is used to constrain the volume of the weight space; 3) Boltzmann pruning is used to constrain the dimension of the weight space [24]. All three of these changes are made in the inner loop of a conjugate gradient optimization iteration [25] and are intended to simplify the training dynamics of the optimization. On handprinted digit classification problems, these modifications improve error-reject performance by factors between 2 and 4 and reduce network size by 40% to 60%.

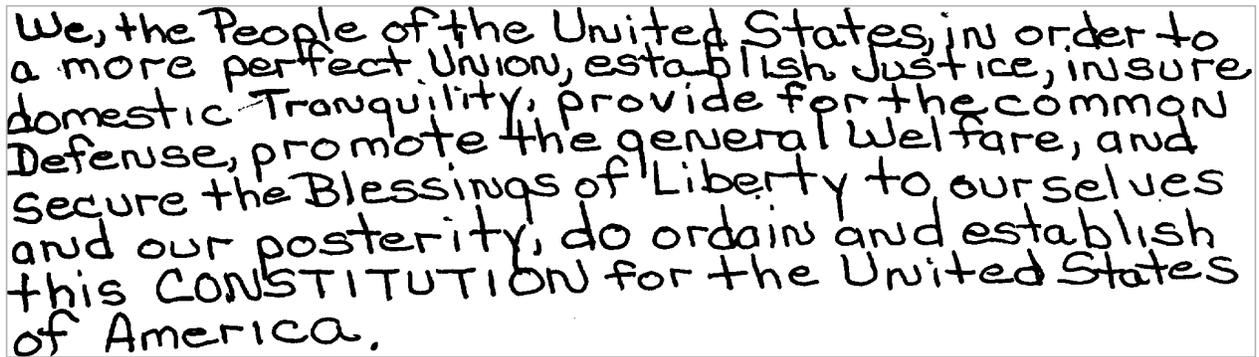
To classify a character, the appropriate eigenvectors (or basis functions) and MLP weight matrices must be loaded into memory. As mentioned earlier, this is accomplished during batch initialization. Using the eigenvectors, the normalized image is transformed into a feature vector. The feature vector is then presented to the MLP network. The result is an assigned classification along with a confidence value.

4.2.10 Spell-Correct Text Line(s); `src/lib/phrase/spellphr.c`; `spell_phrases_Rel2()`

The only field on the HSF form that has any linguistic information that can be applied is the Preamble field. The Preamble is comprised of 38 unique words which are used to form a field-specific dictionary.

The dictionary-based processing conducted by the new system is somewhat different than other correction techniques [26][27]. Up to this point, segmented character images have been extracted from the handprinted paragraph, sorted into reading order line by line, and classified. This results in one long contiguous character stream for each line in the paragraph. The MLP weights used to process the Preamble paragraph were trained to map lowercase and uppercase instances of the same letter into the same class, making the output of the classifier case-invariant. There are also no interword gaps identified by the system at this point. Figure 10 shows an example of these raw classifications.

Words are parsed from each line of raw classifications by applying the preloaded dictionary as described in reference [9]. This process identifies words within the character stream while simultaneously compensating for errors due to wrong segmentations and classifications. The limited size of the dictionary helps offset the burden placed on this process.



Raw Classifications

WEJTHEPEOPIEOPTHEUNITEASTATFSJLNORDERTO
 AMOREPQRFKTUNIONJEBTAEIBHJUSTICEJINSURE
 DOMDLCITRONGUIIJTYIPROVIDEFPRTHFCOMMQN
 DEFENBELPROMOTETHEGENEMIWELNRELAND
 SECURETHEBLCSSINPOFLIBBHYTOOOURSELUES
 ANDOURPOSTERLTYIDOORBINANDQDADLISH
 THISCONETITUTIBNFORTHEUNIFEDSBTES
 OFAMERICA

Spell-Corrected Words

WE THE PEOPLE THE UNITED A STATES ORDER TO
 A MORE UNION THE JUSTICE INSURE
 DO TRANQUILITY PROVIDE FOR THE COMMON
 DEFENSE PROMOTE THE GENERAL WELFARE AND
 SECURE THE BLESSINGS OF LIBERTY TO OURSELVES
 AND OUR POSTERITY DO FOR IN AND A
 THIS CONSTITUTION FOR THE UNITED STATES
 OF AMERICA

Figure 10. Results from processing the top paragraph image.

Hypothesized words are constructed from sequences of the classifier output and then matched to the dictionary. When there is a sufficiently good match, the dictionary word is output, the process resynchronizes itself in the

character stream, and parsing resumes. The matching criterion takes into account the number of errors in the word relative to the length of the word. This way longer words are permitted to tolerate more errors.

4.2.11 Store Results; src/lib/fet/writfet.c; writfetfile()

When processing the Preamble paragraph, the system produces a sequence of spell-corrected words as output. Results of spell-correcting the paragraph image in Figure 10 are listed at the bottom of the figure. Shorter words such as articles and prepositions tend to be frequently deleted and in other places inserted, while the system does a reasonable job of recognizing longer words. This type of dictionary processing is better suited to word-spotting than to full OCR transcription.

For the numeric and randomly ordered alphabet fields, the new system outputs for each segmented character an assigned class and its associated confidence as determined by the MLP classifier. Example output files from the recognition system can be found under the top-level directory *data*.

5. PERFORMANCE EVALUATION AND COMPARISONS

This section evaluates and compares the performances of three recognition systems. The first release of the recognition system (based on the PNN classifier) is named *HSFSYS*, the updated version of the original system distributed with this (the second) release is named *HSFSYS1*, and the new system (based on the MLP classifier) is *HSFSYS2*. Each of these systems was designed to process the HSF forms distributed in *NIST Special Database 19 (SD19)* [3]. *HSFSYS* and *HSFSYS1* are capable of processing the forms in SD19 partitions hsf_0, 1, 2, & 3, whereas the new system, *HSFSYS2*, is capable of processing *every* one of the 3669 forms in the database. This section presents comprehensive results primarily for *HSFSYS1* and *HSFSYS2* across SD19. Statistics are provided on accuracy, error versus reject performance, timings, and memory usage.

5.1 Accuracies and Error Rates

In order to compile statistics on accuracy and error rates, each system was run across the forms in SD19 and recognition results were stored to file. Recognition system classifications were stored to *hypothesis* files, and their associated confidence values were stored to *confidence* files. Once generated, these files were processed using the NIST Scoring Package [28], and performance statistics were compiled at the character, word, and field levels.

Table 3 lists the digit recognition results of running *HSFSYS1* on the first 2,100 forms (partitions hsf_0 to hsf_3) in SD19. The forms in the remaining partitions differ enough that the method of form registration used in *HSFSYS1* fails. The top portion of the table reports character-level statistics, and the bottom reports field-level accuracies. 33 of the 2,100 forms were rejected due to form registration failures and their characters are not included in the table. It was determined that a majority of these failures occurred due to writing outside the provided boxes with continued responses or annotations. A small number (about 5) failed registration due to spurious noise in critical areas on the form. *HSFSYS1* is an implementation improvement over the originally released system, *HSFSYS*. The same methods are applied in both, only *HSFSYS1* has been improved in terms of its memory usage, more efficient execution, and there is no longer any dependence on FORTRAN subroutines. Both systems use the PNN classifier. As was expected, the results in Table 3 are very similar to those reported for the original system in Reference [1].

The performance statistics in Table 3 can be compared to those listed in Table 4. The second table reports the digit recognition results from the new MLP-based system, *HSFSYS2*. These two systems use significantly different algorithms for more than just classification, and as can be seen, *HSFSYS2* performs significantly better than *HSFSYS1*. In terms of digit accuracy, *HSFSYS2* is 3.2% more accurate at 96.3% and it recognizes 86% of the digit fields entirely correctly (6% more than *HSFSYS1*). This difference in accuracy is primarily attributed to the different segmentation methods used in the systems, not to the different classifiers. Studies have shown that at zero-rejection, PNN and the new MLP classifier have similar accuracy [8]. Looking at deletion errors, *HSFSYS2* cuts them by 80% which confirms the improved performance of the system's statistically adaptive segmentor [7]. In addition, *HSFSYS2* is capable of registering every form in SD19, with only 10 fields (6 digit fields, 1 lowercase field, and 3 Preamble fields) rejected due to poor image quality. The characters from these 10 fields have been tallied into the reported statistics as deletions.

HSFSYS1 DIGIT RECOGNITION

		hsf_0	hsf_1	hsf_2	hsf_3	Total
Characters	Correct	92.9% 59288	93.0% 59868	92.6% 58258	93.9% 72872	93.1% 250286
	Substituted	3.9% 2481	3.9% 2531	4.1% 2578	3.6% 2782	3.9% 10372
	Inserted	0.6% 398	0.7% 475	0.6% 385	0.7% 539	0.7% 1797
	Deleted	3.2% 2061	3.0% 1951	3.3% 2084	2.5% 1956	3.0% 8052
	Total	63830	64350	62920	77610	268710
Fields	Correct	79.0% 10865	79.5% 11012	79.1% 10717	81.7% 13662	79.9% 46256
	Total	13748	13860	13552	16716	57876

Table 3. HSFSYS1 accuracies and error rates for digit fields across the first part of SD19.

HSFSYS2 DIGIT RECOGNITION

		hsf_0	hsf_1	hsf_2	hsf_3	hsf_4	hsf_6*	hsf_7	hsf_8	Total
Characters	Correct	96.6% 62772	96.5% 62731	96.1% 62486	97.2% 75804	93.7% 60933	97.3% 63144	96.3% 62615	96.9% 8817	96.3% 459302
	Substituted	2.8% 1816	2.9% 1871	3.0% 1972	2.4% 1891	5.5% 3575	2.1% 1367	3.0% 1920	2.5% 229	3.1% 14641
	Inserted	0.7% 454	0.7% 487	0.7% 425	0.7% 571	0.8% 489	0.5% 318	0.6% 422	0.3% 25	0.7% 3191
	Deleted	0.6% 412	0.6% 398	0.8% 542	0.4% 305	0.8% 492	0.6% 359	0.7% 465	0.6% 54	0.6% 3027
	Total	65000	65000	65000	78000	65000	64870	65000	9100	476970
Fields	Correct	86.3% 12084	86.7% 12139	86.2% 12068	88.6% 14881	77.5% 10855	89.6% 12517	86.5% 12105	88.2% 1728	86.0% 88377
	Total	14000	14000	14000	16800	14000	13972	14000	1960	102732

Table 4. HSFSYS2 accuracies and error rates for digit fields across SD19.

(* Segmented character images from the writers in this partition were used to train the neural network classifiers.)

The results of uppercase recognition can be compared between Table 5 and Table 6. HSFSYS2 recognizes uppercase characters at nearly 90% (4.6% higher than HSFSYS1). Again, the difference in performance can be primarily attributed to the segmentation methods used. With HSFSYS2, insertion errors are reduced by 46% and deletion errors by 58%.

HSFSYS1 UPPERCASE RECOGNITION

		hsf_0	hsf_1	hsf_2	hsf_3	Total
Characters	Correct	84.7% 10808	85.5% 11004	84.7% 10661	86.2% 13377	85.3% 45850
	Substituted	12.8% 1636	11.8% 1524	12.8% 1609	12.0% 1858	12.3% 6627
	Inserted	4.7% 599	4.3% 556	5.5% 687	4.6% 717	4.8% 2559
	Deleted	2.5% 322	2.7% 342	2.5% 314	1.8% 287	2.4% 1265
	Total	12766	12870	12584	15522	53742

Table 5. HSFSYS1 accuracy and error rates for uppercase fields across the first part of SD19.

HSFSYS2 UPPERCASE RECOGNITION

		hsf_0	hsf_1	hsf_2	hsf_3	hsf_4*	hsf_6*	hsf_7	hsf_8	Total
Characters	Correct	89.1% 11587	89.3% 11603	89.2% 11591	89.9% 14029	90.3% 11740	93.0% 12062	88.9% 11563	89.5% 1629	89.9% 85804
	Substituted	9.7% 1256	9.3% 1210	9.4% 1223	9.1% 1425	9.1% 1187	6.5% 847	10.4% 1347	8.8% 161	9.1% 8656
	Inserted	2.3% 294	2.3% 300	2.5% 320	2.7% 426	3.4% 436	2.4% 315	2.6% 336	1.4% 25	2.6% 2452
	Deleted	1.2% 157	1.4% 187	1.4% 186	0.9% 146	0.6% 73	0.5% 65	0.7% 90	1.6% 30	1.0% 934
	Total	13000	13000	13000	15600	13000	12974	13000	1820	95394

Table 6. HSFSYS2 accuracy and error rates for uppercase fields across SD19.

(* Segmented character images from the writers in this partition were used to train the neural network classifiers.)

Lowercase statistics are listed in Table 7 and Table 8. HSFSYS2 correctly recognizes not quite 80% of the lowercase characters in SD19. Not only does the new system employ a new segmentor, it also conducts intelligent line removal that preserves character stroke data that overlaps with the form and extends beyond the immediate limits of the field. An independent study [5] shown that one can expect up to a 3% improvement in lowercase accuracy when using this method of line removal. The difference between HSFSYS1 and HSFSYS2 is 2.8%, some of which can be directly attributed to the line removal. Adaptive character segmentation is also contributing, as insertion errors are reduced by 70%. This demonstrates the segmentor’s ability to compose characters from multiple connected components, as unattached fragments contribute to insertion errors. On the other hand, the number of deletion errors increases with HSFSYS2. This leads one to conclude that the adaptive segmentor may be over-aggressive in merging components, and not aggressive enough when it comes to splitting touching characters. An independent study has shown that the general segmentation method used in HSFSYS2 can benefit from further refinement for lowercase characters [6].

HSFSYS1 LOWERCASE RECOGNITION

		hsf_0	hsf_1	hsf_2	hsf_3	Total
Characters	Correct	75.1% 9593	76.8% 9890	76.5% 9625	78.7% 12217	76.9% 41325
	Substituted	22.9% 2927	21.4% 2748	22.3% 2804	19.9% 3092	21.5% 11571
	Inserted	3.2% 405	2.6% 336	3.0% 381	3.2% 500	3.0% 1622
	Deleted	1.9% 246	1.8% 232	1.2% 155	1.4% 213	1.6% 846
	Total	12766	12870	12584	15522	53742

Table 7. HSFSYS1 accuracy and error rates for lowercase fields across the first part of SD19.

HSFSYS2 LOWERCASE RECOGNITION

		hsf_0	hsf_1	hsf_2	hsf_3	hsf_4*	hsf_6*	hsf_7	hsf_8	Total
Characters	Correct	77.2% 10042	78.6% 10218	77.8% 10109	80.4% 12541	82.6% 10739	82.9% 10756	79.1% 10280	75.5% 1374	79.7% 76059
	Substituted	20.4% 2654	19.2% 2495	20.3% 2645	17.8% 2781	15.1% 1967	15.2% 1966	18.4% 2397	20.5% 373	18.1% 17278
	Inserted	0.9% 119	0.8% 102	0.8% 110	1.1% 168	0.8% 110	0.6% 75	0.9% 114	1.0% 18	0.9% 816
	Deleted	2.3% 304	2.2% 287	1.9% 246	1.8% 278	2.3% 294	1.9% 252	2.5% 323	4.0% 73	2.2% 2057
	Total	13000	13000	13000	15600	13000	12974	13000	1820	95394

Table 8. HSFSYS2 accuracy and error rates for lowercase fields across SD19.

(* Segmented character images from the writers in this partition were used to train the neural network classifiers.)

The last pair of tables (Table 9 and Table 10) lists the results of recognizing words across SD19's Preamble fields. SD19 has completed Preamble paragraphs only in its first 4 partitions. These word-level statistics were computed by tokenizing each word in the system output. The NIST Scoring Package [28] was used to align the word tokens with the known Preamble text, and statistics were accumulated. Much effort was spent in improving the line isolation algorithm used in HSFSYS2 [6]. Even so, overall word accuracy only improved 2.3% (61.6% to 63.9%). Considerable work still remains in improving the segmentation of vertically and horizontally touching characters, the detection of punctuation marks, and dictionary-based spelling correction.

HSFSYS1 PREAMBLE RECOGNITION

		hsf_0	hsf_1	hsf_2	hsf_3	Total
Words	Correct	60.3% 15403	59.8% 15387	60.5% 15237	65.0% 20166	61.6% 66193
	Substituted	15.2% 3871	14.3% 3676	14.0% 3525	12.7% 3943	14.0% 15015
	Inserted	1.1% 283	0.9% 224	1.1% 270	1.1% 335	1.0% 1112
	Deleted	24.5% 6258	25.9% 6677	25.5% 6406	22.3% 6935	24.4% 26276
	Total	25532	25740	25168	31044	107484

Table 9. HSFSYS1 accuracy and error rates for Preamble fields across SD19.

HSFSYS2 PREAMBLE RECOGNITION

		hsf_0	hsf_1	hsf_2	hsf_3	Total
Words	Correct	62.6% 16276	62.4% 16231	62.7% 16304	67.2% 20961	63.9% 69772
	Substituted	15.4% 4012	15.2% 3958	14.7% 3833	12.6% 3940	14.4% 15743
	Inserted	1.6% 405	1.0% 260	1.1% 287	1.3% 418	1.3% 1370
	Deleted	22.0% 5712	22.4% 5811	22.6% 5863	20.2% 6299	21.7% 23685
	Total	26000	26000	26000	31200	109200

Table 10. HSFSYS2 accuracy and error rates for Preamble fields across SD19.

As a final note on these accuracy statistics, realize that results are reported with HSFSYS2 having processed the entire set of forms in SD19. This is one of the largest published experiments of its kind, and it is reproducible by purchasing the SD19 database from NIST. In all, sample handwriting from 3669 writers was tested and a total of 109,200 words and 667,758 characters were recognized and scored. As SD19 is our only handprint database, training samples were extracted from specific writer partitions and used to train the PNN and MLP character classifiers off-line. From the 667,758 characters, 109,719 were used in training. In the case of digits, the writers in hsf_6 (61,094 characters) were used in the training set, and in the case of upper and lowercase, writers in both hsf_4 and hsf_6 (totalling 24,420 uppercase characters and 24,205 lowercase characters) were used. It is worth pointing out that the machine processes used to isolate the character training samples were different, as they predate the technology used in HSFSYS2.

Comparing the HSFSYS2 results on hsf_6 to other partitions, it is interesting to see that the inclusion of hsf_6 in the classifier training does have some influence, however the influence is small. With digits, HSFSYS2 is 97.3% correct on hsf_6 whereas the results on hsf_3 are almost as good at 97.2%, and the other partitions (with the exception of hsf_4) range between 96% and 97%. The writers in hsf_4 are from a different population and are known to be statistically more difficult to recognize [29]. The influence of training is a bit more pronounced with the results on upper and lowercase fields. On uppercase, HSFSYS2 is 93% correct on hsf_6, and the other partitions range between 89% to 90%. For lowercase, HSFSYS2 is 83% correct on hsf_6, and other partitions range between 77% to 80%. These small differences (particularly for the digits) demonstrate that the MLP character classifier is doing a reasonably good job at generalizing on writers it hasn't seen during its off-line training. The MLP-based system doesn't generalize as well on upper and lowercase recognition in part because fewer training samples were used than for digits.

5.2 Error versus Rejection Rate

The advantages of using a machine for OCR in many ways complement the performance of humans [2]. Machines are very efficient in doing tasks that are primarily repetitive and reflexive, whereas humans quickly fatigue under these conditions. Humans, on the other hand, are very adept at performing tasks requiring higher-level reasoning, and as a result, provide more robust but much slower solutions to complex problems. Accounting for these differences, successful recognition systems allow a machine to perform the bulk of the work, and on an exception basis, humans can be used to resolve ambiguities and potential errors. This is accomplished through rejection mechanisms that automatically route low-confidence machine decisions to humans for verification. This section compares the ability of the NIST recognition systems to effectively reject low-confidence character classifications.

The graph in Figure 11 plots error versus rejection rates with error plotted on a logarithmic scale. The results plotted were computed from the first 500 writers (partition *hsf_0*) in SD19. Results are shown for both HSFSYS1 and the new system HSFSYS2, and they are broken out by digit, upper, and lowercase recognition. In general, as the number of rejected character classifications increases, the error rate on the remaining accepted (or non-rejected) classifications decreases, and accuracy improves. Also, the impact of rejection on accuracy tapers off as more and more characters are rejected. In the figure, the bottom two curves represent the performance of the new and old systems on recognizing characters in the numeric fields on the HSF forms. With no rejection, HSFSYS2 has an error rate near 4%, and HSFSYS1 has an error rate over 7.5%. As the number of rejected digit classifications is increased, the error rate proceeds to drop, only HSFSYS2 falls at a significantly faster rate than does HSFSYS1. The difference in the slope of the two digit curves confirms the robustness of the MLP classifier used in HSFSYS2 over the PNN classifier used in HSFSYS1. The digit error rate of HSFSYS2 continues to drop to nearly 1.2% at 15% rejection. One concludes from these results, that in terms of recognizing numeric fields, the new NIST recognition system is more than twice as good as the original system.

The differences between the two systems are less dramatic with upper and lowercase recognition. The middle two curves in Figure 11 correspond to the results of recognizing the uppercase alphabet fields on the HSF forms. The HSFSYS2 curve does fall off slightly faster than does HSFSYS1's, but the distance between the curves is not as large as that of the digit curves. With no rejection, HSFSYS2 has an error rate of almost 13% and HSFSYS1 is just over 19%. The two lowercase curves are even closer to each other, and their distance only slightly increases across the range of rejections plotted. This emphasizes that lowercase recognition is still the most difficult for the NIST systems. The minimal increase in separation between these two curves can be attributed to a combination of two factors. First, the decision surfaces trained within the MLP classifier for lowercase are much more complex than those of uppercase, and the decision surfaces for uppercase are more complex than those of digits [8]. Second, the challenges remaining in the system that are impacting accuracy lie primarily in components other than the classifier. Otherwise, the relative slopes in the upper and lowercase curves would more closely resemble those of the digit classifications.

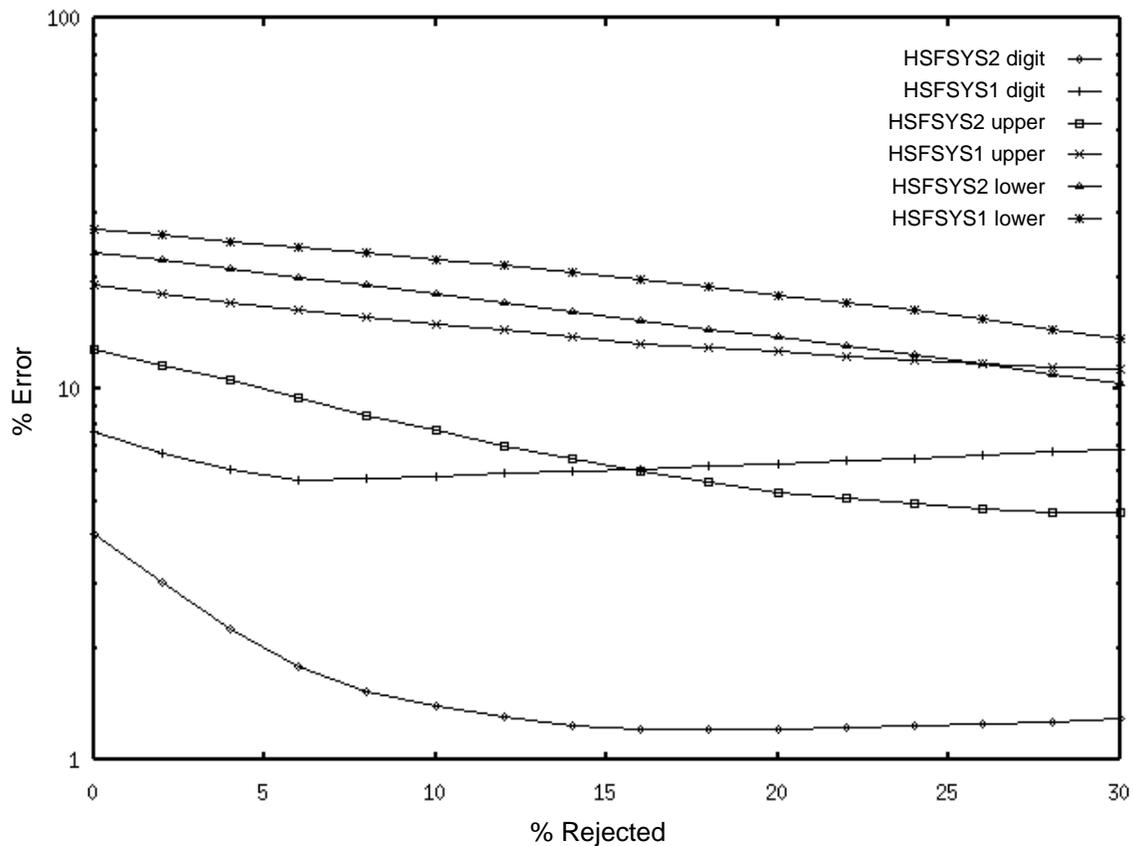


Figure 11. Error versus rejection rates for digit, upper, and lowercase recognition between HSFYSYS1 and HSFYSYS2.

5.3 Timing and Memory Statistics

Over the two years following the first system's release, a number of significant improvements were made to the existing source code so that a modified version of the old system now runs faster and uses memory more efficiently. As new methods were developed to improve the NIST system, the focus was primarily on improving system accuracy, although considerable effort was made to ensure that the resulting implementations were time-efficient. This section first compares the timing results between the original system (HSFYSYS) as it was distributed in the first release, the augmented original system (HSFYSYS1) as it is distributed in this release, and the new system (HSFYSYS2).

Table 11 lists timing statistics (in seconds) for each of the major components in the original and augmented versions of the old recognition system. The timings reported in Table 11 and Table 12 were generated on a Sun Microsystems SPARCstation 2 with a Weitek CPU upgrade, and the reported user times are an average over the 10 HSF forms included in the test-bed. The first pair of data columns in Table 11 lists results from the implementation distributed in the first release, and the second pair of columns lists results from the augmented version distributed with this release. Virtually the same algorithms are used throughout, only memory usage has been made more efficient, and in several places, the source code was modified to execute more quickly. The most significant change is in spelling correction, where in the development of the new release, it was discovered that memory was being allocated and then deallocated every time a word was being matched to the dictionary. By simply allocating a matrix only once, the average time required to spell-correct a handprinted Preamble paragraph dropped by almost a factor of 9. Another time improvement to note is that the PNN character classifier was modified to use new internal data structures and now runs 24% faster. This is slightly offset by field initialization taking 2 seconds longer due to a new PNN-supporting file format. Also note that the time required to compute the KL transform (within character feature extraction) has been cut in half. As a result of implementation changes, the component taking the most time in HSFYSYS1 is now the PNN clas-

sifier, requiring nearly a quarter of the time. Form registration, field initialization, then segmentation also require significant amounts of time.

Task	HSFSYS		HSFSYS1	
form init	1.1	1.4%	0.9	2.0%
form register	9.8	12.3%	9.4	20.6%
form remove	1.0	1.3%	0.8	1.8%
field init	5.5	7.0%	7.7	16.8%
field isolate	1.0	1.3%	1.0	2.3%
field segment	7.0	8.8%	6.8	14.9%
chr normalize	0.9	1.2%	0.9	1.9%
chr shear	0.3	0.4%	0.3	0.6%
chr feature	5.6	7.0%	2.7	6.0%
chr classify	13.8	17.4%	10.5	23.1%
chr sort	0.2	0.3%	0.2	0.5%
field spell	33.2	41.7%	3.8	8.3%
total	79.6	100.0%	45.6	100.0%

Table 11. Timing statistics in seconds between original and augmented versions of old system.

Task	HSFSYS2	
batch init	1.7	5.1%
form load	1.5	4.6%
form register	5.6	17.1%
field isolate	8.9	27.2%
field segment	1.1	3.2%
chr normalize	0.9	2.7%
chr feature	5.4	16.4%
chr classify	4.0	12.2%
field spell	3.8	11.4%
total	32.9	100.0%

Table 12. Timing statistics in seconds for new version of recognition system.

Table 12 reports the average user times required to run the new system, HSFSYS2, across the same set of 10 HSF forms. First, notice that the overall time required has decreased by nearly 28% when compared to HSFSYS1. The speed increase can be explained in part by using the new MLP classifier in place of the PNN. The MLP character classifier is a factor of 2.6 faster, and using the MLP does not require field initialization. All the MLP weights for all the types of fields on the HSF form can be held in memory simultaneously, so they are read from file once during batch initialization. The time for this initialization is factored across the number of forms processed within the batch. As the number of forms increases, the percentage of time required for the reading of the weights becomes negligible. The PNN classifier requires much more memory, and it becomes infeasible to hold all of its weights (training prototypes) in memory at once. Every time the PNN-based system begins processing a new type of field, a large number of new field-specific prototypes must be read from file.

In order to analyze memory usage, the SunOS/UNIX routine *mallinfo()* was used to measure the maximum arena size of the various NIST recognition systems. During the execution of the original system, HSFSYS grew to

require a total arena size of 33.7Mb; the more efficient implementation (HSFSYS1) required only 21.9Mb; and the new MLP-based HSFSYS2 required 25.1 Mb. The number of floating point values required by the PNN in HSFSYS1 to classify digits is over 4 million (61,094 prototypes \times 64 KL coefficients), whereas the MLP digit weights in HSFSYS2 contain about 18,000 floating point values (a 128 \times 128 \times 10 network). One would expect this dramatic difference in the required weight size to be reflected in the overall arena sizes between the two systems, but instead, HSFSYS2 actually uses more memory than HSFSYS1.

This is primarily due to the internal representation of image used in the two systems. While HSFSYS1 attempts to maintain a general binary image representation with eight pixels packed in one byte, HSFSYS2 expands images to be one pixel per byte. This makes arbitrary pixel addressing less expensive, but it does utilize 8 times more memory. A full page 11.8 pixel/millimeter (300 pixels/inch) binary image requires about 1Mb of memory when pixels are packed 8 per byte. Expanded, the same image with one pixel per byte requires over 8 Mb. In general, an image transformation on the expanded image will result in two images in memory (the source image and the resulting image). These two images now require a total of 16Mb, whereas the 8 pixels per byte representation would require only 2Mb. In effect, the 16Mb of PNN weights (4 million floats \times 4 bytes/float) replaced by the 18,000 MLP weights is offset (and then some) by the 16Mb of additional image representation. Thus the new MLP-based system uses slightly more memory than the PNN-based one.

As part of the testing of the second release, the software was installed and executed, and results were analyzed on a number of different UNIX platforms. These systems included computers manufactured by Digital Equipment Corporation, Hewlett Packard, IBM, Silicon Graphics Incorporated, and Sun Microsystems. Times are listed in Table 13 for both the augmented original system (HSFSYS1) and for the new system (HSFSYS2). The times reported are the average user times required to process an HSF form, and the statistics were computed across the 10 HSF forms provided with this distribution. On all the machines, HSFSYS2 processed the 10 forms faster than did HSFSYS1. Two computers, the SGI Challenge and the Sun SPARCstation 10, have multiple processors. However, the recognition systems were compiled serially on these machines and run on single processors, so no parallel processing was employed. The range of user times varies by a factor of 3 to 4 over the set of machines tested. On the faster machines, HSF forms are processed 10 to 15 seconds a page.

Man.	Model	O.S.	RAM	HSFSYS1	HSFSYS2
DEC	Alpha 3000/400	OSF/1 V1.3	32 Mb	10.4	10.0
SGI	Indy (IP22)	IRIX 5.3	128 Mb	13.2	10.3
IBM	RS6000 Model 370	AIX 4.1	128 Mb	17.1	15.4
SGI	Challenge (8-IP19's)*	IRIX 5.3	512 Mb	17.5	14.3
HP	9000/735	HP-UX A.09.05	32 Mb	18.2	14.1
Sun	SPARCstation 10 (2-CPU's)*	SunOS 5.4 (Solaris)	128 Mb	34.3	24.4
Sun	SPARCstation 2 (Weitek 80MHz CPU)	SunOS 4.1.3	64 Mb	45.6	32.9

Table 13. Table of timings in seconds from the different UNIX computers tested.

(*Those computers with multiple processors were compiled and tested serially.)

6. IMPROVEMENTS TO THE TEST-BED

With the new technologies released in this distribution, the NIST recognition software test-bed is much closer to being a usable product than its first release. There should be significantly less effort commercializing the new technology within the framework provided. Nonetheless, the new system is a technology transfer rather than a shrink-wrapped product. One item missing from the test-bed, which may be required for its commercialization, is a forms identification component so that the system can process more than one type of form within a batch. Also missing is an integrated work flow for routing and correcting rejected classifications with human key operators. The modular system architecture does however provide handles to support this work flow. Finally, a formalized form definition utility is needed, supported by an interactive interface that sets up the recognition system to process new types of forms.

6.1 Processing New Forms with the HSFSYS2

To set up the new recognition system, *hsfsys2*, to process a new type of form, *trainreg* must be run on a prototypical form and the output coordinates stored. In addition, the fields or zones on a registered version of the new form must be measured manually with an interactive image display tool (not provided with this distribution), and the coordinates of each field must be stored to an MFS file. The file *tmplt/hsfsmplt.pts* contains the field coordinates for a registered HSF form. The source code must be modified to load these two new files (the registration coordinate file and the field coordinate file).

If a new form contains fields different than those on an HSF form, then the MLP classifier will need to be retrained and the resulting weights files will need to be loaded into the system. For example, if a new form contains money fields that include dollar signs, commas, and decimal points, then it will be desirable to train the MLP network to classify these three new characters in addition to the ten numeric characters. A future improvement to *hsfsys2* would be to develop and incorporate a forms definition tool that locates the zones on a new form (preferably automatically) and then systematically prompts an operator to identify the types of each field on the form so that the appropriate classifier weights, form removal, syntax checking, and field-specific dictionaries can be automatically applied by the system. Currently, this must all be done manually through the coding of a new application-specific main program.

7. FINAL COMMENTS

A number of NIST Internal Reports (NISTIR's) have been referenced in this document. These reports are provided in PostScript format in the top-level directory *doc*. The file *doc/hfsys2.ps* contains this specific document. These reports along with many other NIST Visual Image Processing Group publications are available in PostScript format over the Internet via anonymous FTP on *sequoyah.nist.gov* or via the World Wide Web at <http://www.nist.gov/itl/div894/894.03>. To request a paper copy of any of these NISTIRs, please contact:

ITL Publications
National Institute of Standards and Technology
Building 225, Room B216
Gaithersburg, MD 20899
voice: (301) 975-2832

This report documents the second release of the NIST standard reference recognition software test-bed in terms of its installation, organization, and functionality. The software has been successfully compiled and tested on a number of different vendors' UNIX workstations. If necessary, it is the responsibility of the distribution recipient to port the software to their specific computer architecture. The source code is written entirely in C and is organized into 15 libraries. In all, there are approximately 39,000 lines of code supporting more than 725 subroutines. Source code is provided for a wide variety of utilities that have application to many other types of problems.

Approximately 25 person-years have been invested by NIST in the development of this software test-bed, and it can be obtained free of charge on CD-ROM by sending a letter of request via postal mail or FAX to the primary author. Requests for distribution made by electronic mail will not be accepted; however, electronic mail is encouraged for technical questions once the distribution has been received. Any portion of this test-bed may be used without restrictions because it was created with U.S. government funding. Redistribution of this standard reference software is strongly discouraged as any subsequent corrections or updates will be sent to registered recipients only. This software was produced by NIST, an agency of the U.S. government, and by statute is not subject to copyright in the United States. Recipients of this software test-bed assume all responsibilities associated with its operation, modification, and maintenance.

8. REFERENCES

- [1] M. D. Garris, J. L. Blue, G. T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C. L. Wilson, "NIST Form-Based Handprint Recognition System," NIST Internal Report 5469 and CD-ROM, July 1994.
- [2] C. L. Wilson, J. Geist, M. D. Garris, and R. Chellappa, "Design, Integration, and Evaluation of Form-Based Handprint and OCR Systems," NIST Internal Report 5932, December 1996.
- [3] P. J. Grother, "Handprinted Forms and Characters Database, *NIST Special Database 19*," NIST Technical Report and CD-ROM, March 1995.
- [4] M. D. Garris and P. J. Grother, "Generalized Form Registration Using Structure-Based Techniques," NIST Internal Report 5726 and in Proceedings of the *Fifth Annual Symposium on Document Analysis and Information Retrieval*, pp. 321-334, UNLV, April 1996.
- [5] M. D. Garris, "Method and Evaluation of Character Stroke Preservation of Handprint Recognition," NIST Internal Report 5687, July 1995, and in Proceedings of *Document Recognition III*, Vol. 2660, pp. 321-332, SPIE, San Jose, February 1996.
- [6] M. D. Garris, "Teaching Computers to Read Handprinted Paragraphs," NIST Internal Report 5894, September 1996.
- [7] M. D. Garris, "Component-Based Handprint Segmentation Using Adaptive Writing Style Model," NIST Internal Report 5843, June 1996.
- [8] C. L. Wilson, J. L. Blue, O. M. Omidvar, "The Effect of Training Dynamics on Neural Network Performance," NIST Internal Report 5696, August 1995.
- [9] M. D. Garris, "Unconstrained Handprint Recognition Using a Limited Lexicon," NIST Internal Report 5310, December 1993, and in Proceedings of *Document Recognition*, Vol. 2181, pp. 36-46, SPIE, San Jose, February 1994.
- [10] Department of Defense, "Military Specification - Raster Graphics Representation in Binary Format, Requirements for, MIL-R-28002," 20 Dec 1988.
- [11] CCITT, "Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus, Fascicle VII.3 - Rec. T.6," 1984.
- [12] C. R. Wyle, *Advanced Engineering Mathematics*, Second Edition, pp. 175-179, McGraw-Hill, New York, 1960.
- [13] A. K. Jain, *Fundamentals of Digital Image Processing*, pp. 384-389, Prentice-Hall, New Jersey, 1989.
- [14] P. J. Grother, "Karhunen Loève Feature Extraction for Neural Handwritten Character Recognition," NIST Internal Report 4824, April 1992, and in Proceedings of *Applications of Artificial Neural Networks III*, Vol. 1709, pp. 155-166. SPIE, Orlando, April 1992.
- [15] D. F. Specht, "Probabilistic Neural Networks." *Neural Networks*, Vol. 3(1), pp 109-119, 1990.
- [16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel Distributed Processing, Volume 1: Foundations*, edited by D. E. Rumelhart, J. L. McClelland, et. al., MIT Press, Cambridge, pp. 318-362, 1986.
- [17] H. G. Zwakenberg, "Inexact Alphanumeric Comparison," *The C Users Journal*, pp. 127-131, May 1991.
- [18] ISO-9660, "Information Processing - Volume and File Structure of CD-ROM for Information Interchange," Standard by the International Organization for Standardization, 1998.
- [19] J. Geist, R. A. Wilkinson, S. Janet, P. J. Grother, B. Hammond, N. W. Larsen, R. M. Klear, M. J. Matsko, C. J. C. Burges, R. Creecy, J. J. Hull, T. P. Vogl, C. L. Wilson, "The Second Census Optical Character Recognition Systems Conference," NIST Internal Report 5452, May 1994.
- [20] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, Vol. 18, pp. 509-517, 1975.
- [21] W. Postl, "Method for Automatic Correction of Character Skew in the Acquisition of a Text Original in the Form of Digital Scan Results," United States Patent Number 4,723,297, February 1988.
- [22] J. L. Blue, G. T. Candela, P. J. Grother, R. Chellappa, and C. L. Wilson, "Evaluation of Pattern Classifiers for Fingerprint and OCR Applications," *Pattern Recognition*, Vol. 27, No. 4, pp. 485-501, 1994.
- [23] C. L. Wilson, P. J. Grother, and C. S. Barnes, "Binary Decision Clustering for Neural Network Based Optical Character Recognition," NIST Internal Report 5542, December 1994, and in *Pattern Recognition*, Vol. 29, No. 3, pp. 425-437, 1996.

- [24] O. M. Omidvar and C. L. Wilson, "Information Content in Neural Net Optimization," NIST Internal Report 4766, February 1992, and in *Journal of Connection Science*, 6:91-103, 1993.
- [25] J. L. Blue and P. J. Grother, "Training Feed Forward Networks Using Conjugate Gradients," NIST Internal Report 4776, February 1992, and in Conference on *Character Recognition and Digitizer Technologies*, Vol. 1661, pp. 179-190, SPIE, San Jose, February 1992.
- [26] M. J. Ganzberger, R. Rovner, A. M. Gillies, D. J. Hepp, and P. D. Gader, "Matching Database Records to Hand-written Text," in Proceedings on *Document Recognition*, Vol. 2181, pp. 66-75, SPIE, San Jose, February 1994.
- [27] G. L. Martin and J. A. Pittman, "Recognizing Hand-Printed Letters and Digits," *Neural Networks*, Vol. 3, pp. 258-267, 1991.
- [28] S. A. Janet, "NIST Scoring Package User's Guide, Release 2.0," NIST Technical Report and Software, to be published.
- [29] P. J. Grother, "Cross Validation Comparison of NIST OCR Databases," NIST Internal Report 5123, January 1993, and in Proceedings of *Character Recognition Technologies*, Vol. 1906, pp. 296-307, SPIE, San Jose, February 1993.
- [30] D. Liu and J. Nocedal, "On the Limited Memory BFGS Method for Large Scale Optimization," *Mathematical Programming B*, Vol. 45, 503-528, 1989.

Appendix A. TRAINING THE MULTI-LAYER PERCEPTRON (MLP) CLASSIFIER OFF-LINE

The program *mlp* trains a 3-layer feed-forward linear perceptron [16] using novel methods of machine learning that help control the learning dynamics of the network. As a result, the derived minima are superior, the decision surfaces of the trained network are well-formed, the information content of confidence values is increased, and generalization is enhanced. Trained MLP networks are used in the new recognition system, *hsfsys2*. As a classifier, this new MLP is superior to the PNN classifier used in *hsfsys1* in terms of its memory requirements, classification speed, and superior confidence values for rejecting confusions. The theory behind the machine learning techniques used in this program is discussed in Reference [8]. The main routine for this program is found in *src/bin/mlp/mlp.c* and the majority of its supporting subroutines is located in the library *src/lib/mlp*.

Machine learning is controlled through a batch-oriented iterative process of training the MLP on a set of prototype feature vectors, and then evaluating the progress made by running the MLP (in its current state) on a separate set of feature vectors. Training on the first set of patterns then resumes for a predetermined number of passes through the training data, and then the MLP is tested again on the evaluation set. This process of training and then testing continues until the MLP has been determined to have satisfactorily converged. The command line invocation of *mlp* is as follows:

```
% mlp [-c] [specfile]
```

- The first optional argument *-c* performs checking of the specfile only: scan it; write any applicable warnings or error messages to the standard error output; then exit.
- The second optional argument *specfile* is the name of the specification file to be processed by *mlp*. If this argument is omitted, the specfile is assumed to be the file *spec* in the current working directory. The format of the specfile is documented in the routine *scanspec()* found in *src/lib/mlp/scanspec.c*.

This command trains or tests an MLP neural network suitable for use as a classifier or as a function-approximator. The network has an input layer, a hidden layer, and an output layer, each layer comprising a set of nodes. The input nodes are feed-forwardly connected to the hidden nodes, and the hidden nodes to the output nodes, by connections whose weights (strengths) are trainable. The activation function used for the hidden nodes can be chosen to be sinusoid, sigmoid (logistic), or linear, as can the activation function for the output nodes. Training (optimization) of the weights is done using either a Scaled Conjugate Gradient (SCG) algorithm [25], or by starting out with SCG and then switching to a Limited Memory Broyden Fletcher Goldfarb Shanno (LBFGS) algorithm [30]. Boltzmann pruning [24], i.e. dynamic removal of connections, can be performed during training if desired. Prior weights can be attached to the patterns (feature vectors) in various ways.

A.1 Training and Testing Runs

When *mlp* is invoked, it performs a sequence of *runs*. Each run does either training, or testing:

training run: A set of *patterns* is used to train (optimize) the weights of the network. Each pattern consists of a *feature vector*, along with either a *class* or a *target vector*. A feature vector is a tuple of floating-point numbers, which typically has been extracted from some natural object such as a handwritten character. A class denotes the actual class to which the object belongs, for example the character which a handwritten mark is an instance of. The network can be trained to become a classifier: it trains using a set of feature vectors extracted from objects of known classes. Or, more generally, the network can be trained to learn, again from example input-output pairs, a function whose output is a vector of floating-point numbers, rather than a class; if this is done, the network is a sort of interpolator or function-fitter. A training run finishes by writing the final values of the network weights as a file. It also produces a summary file showing various information about the run, and optionally produces a longer file that shows the results the final (trained) network produced for each individual pattern.

testing run: A set of patterns is sent through a network, after the network weights are read from a file. The output values, i.e. the hypothetical classes (for a classifier network) or the produced output vectors (for a fitter network), are compared with target classes or vectors, and the resulting error rate is computed. The program can produce a table showing the correct classification rate as a function of the rejection rate.

A.2 Specification (Spec) File

This is a file produced by the user, which sets the parameters (henceforth “parms”) of the run(s) that *mlp* is to perform. It consists of one or more *blocks*, each of which sets the parms for one run. Each block is separated from the next one by the word “newrun” or “NEWRUN”. Parmes are set using name-value pairs, with the name and value separated by non-newline white space characters (blanks or tabs). Each name-value pair is separated from the next pair by newline(s) or semicolon(s). Since each parm value is labeled by its parm name, the name-value pairs can occur in any order. Comments are allowed; they are delimited the same way as in C language programs, with /* and */. Extraneous white space characters are ignored. The specfiles used to train the MLP in *hsfsys2* are provided in the *weights/mlp* subdirectories and end with the extension *spc*.

When *mlp* is run, it first scans the entire specfile, to find and report any (fatal) errors (e.g. omitting to set a necessary parm, or using an illegal parm name or value) and also any conditions in the specfile which, although not fatally erroneous, are worthy of warnings (e.g. setting a superfluous parm). *Mlp* writes any applicable warning or error messages; then, if there are no errors in the specfile, it starts to perform the first run. Warnings do not prevent *mlp* from starting to run. (The motivation for having *mlp* check the entire specfile before it starts to perform even the first run, is that this will prevent an *mlp* instance that runs a multi-run specfile from failing, perhaps many hours, or days, after it was started, because of an error in a block far into the specfile: such errors will be detected up front and presumably fixed by the user, because that is the only way to cause *mlp* to get past its checking phase.) To cause *mlp* only to check the specfile without running it, use the *-c* option.

The following listing describes all the parms that can be set in a specfile. There are four types of parms: *string* (value is a filename), *integer*, *floating-point*, and *switch* (value must be one of a set of defined names, or may be specified as a code number). A block of the specfile, which sets the parms for one run, often can omit to set the values of several of the parms, either because the parm is unneeded (e.g., a training “stopping condition” when the run is a test run; or, **temperature** when **boltzmann** is **no_prune**), or because it is an *architecture parm* (**purpose**, **ninps**, **nhids**, **nouts**, **acfunc_hids**, or **acfunc_outs**), whose value will be read from **wts_infile**. The descriptions below indicate which of the parms are needed only for training runs (in particular, those described as stopping conditions). Architecture parms should be set in a specfile block only if its run is to be a training run that generates random initial network weights: a training run that reads initial weights from a file (typically, final weights produced by a previous training session), or a test run (must read the network weights from a file), does not need to set any of the architecture parms in its specfile block, because their values are stored in the weights file that it will read. (Architecture parms are ones whose values it would not make sense to change between training runs of a single network that together comprise a training “meta-run”, nor between a training run for a network and a test run of the finished network.) Setting unneeded parms in a specfile block will result in warning messages when *mlp* is run, but not fatal errors; the unneeded values will be ignored.

If a parm-name/parm-value pair occurring in a specfile has just its value deleted, i.e. leaving just a parm name, then the name is ignored by *mlp*; this is a way to temporarily unset a parm while leaving its name visible for possible future use.

A.2.1 String (Filename) Parmes

short_outfile: This file will contain summary information about the run, including a history of the training process if a training run. The set of information to be written is controlled, to some extent, by the switch parms **do_confuse** and **do_cvr**. See Section A.4.

long_outfile: This optionally produced file will have two lines of header information followed by a line for each pattern. The line will show: the sequence number of the pattern; the correct class of the pattern (as a number in the range 1 through **nouts**); whether the hypothetical class the network produced for this pattern was right (R) or wrong (W); the hypothetical class (number); and the **nouts** output-node activations the network produced for the pattern. (See the switch parm **show_acs_times_1000** below, which controls the formatting of the activations.) In a testing run, *mlp* produces this file for the result of running the patterns through the network

whose weights are read from **wts_infile**; in a training run, *mlp* produces this file only for the final network weights resulting from the training session. This is often a large file; to save disk space by not producing it, just leave the parm unset.

patterns_infile: This file contains *patterns* upon which *mlp* is to train or test a network. A pattern is either a feature-vector and an associated class, or a feature-vector and an associated target-vector. The file must be in one of the two supported patterns-file formats, i.e. ASCII and (FORTRAN-style) binary; the switch parm **pats-file_ascii_or_binary** must be set to tell *mlp* which of these formats is being used.

wts_infile: This optional file contains a set of network weights. *Mlp* can read such a file at the start of a training run - e.g., final weights from a preceding training run, if one is training a network using a sequence of runs with different parameter settings (e.g., decreasing values of **regfac**) - or, in a testing run, it can read the final weights resulting from a training run. This parm should be left unset if *random* initial weights are to be generated for a training run (see the integer parm **seed**).

wts_outfile: This file is produced only for a training run; it contains the final network weights resulting from the run.

lcn_scn_infile: Each line of this optional file should consist of a long class-name (as shown at the top of **patterns_infile**) and a corresponding short class-name (1 or 2 characters), with the two names separated by white space; the lines can be in any order. This file is required only for a run that requires short class-names, i.e. only if **purpose** is **classifier** and (1) **priors** is **class** or **both** (these settings of **priors** require class-weights to be read from **class_wts_infile**, and that type of file can be read only if the short class-names are known) or (2) **do_confuse** is **true** (proper output of confusion matrices requires the short class-names, which are used as labels).

class_wts_infile: This optional file contains class-weights, i.e. a “prior weight” for each class. (See switch parm **priors** to find out how *mlp* can use these weights.) Each line should consist of a short class-name (as shown in **lcn_scn_infile**) and the weight for the class, separated by white space; the order of the lines does not matter.

pattern_wts_infile: This optional file contains pattern-weights, i.e. a “prior weight” for each pattern. (See switch parm **priors** to find out how *mlp* can use these weights.) The file should be just a sequence of floating-point numbers (ascii) separated from each other by white space, with the numbers in the same order as the patterns they are to be associated with.

A.2.2 Integer Parms

npats: Number of (first) patterns from **patterns_infile** to use.

ninps, **nhids**, **nouts**: Specify the number of input, hidden, and output nodes in the network. If **ninps** is smaller than the number of components in the feature-vectors of the patterns, then the first **ninps** components of each feature-vector are used. If the network is a **classifier** (see **purpose**), then **nouts** is the number of classes, since there is one output node for each class. If the network is a **fitter**, then **ninps** and **nouts** are the dimensionalities of the input and output real vector spaces. These are architecture parms, so they should be left unset for a run that is to read a network weights file.

seed: For the UNI random number generator, if initial weights for a training run are to be randomly generated. Its values must be positive. Random weights are generated only if **wts_infile** is not set. (Of course, the seed value can be reused to generate identical initial weights in different training runs; or, it can be varied in order to do several training runs using the same values for the other parameters. It is often advisable to try several seeds, since any particular seed may produce atypically bad results (training may fail). However, the effect of varying the seed is minimal if Boltzmann pruning is used.)

niter_max: A stopping condition: maximum number of iterations a training run will be allowed to use.

nfreq: At every **nfreq**'th iteration during a training run, the **errdel** and **nokdel** stopping conditions are checked and a pair of status lines is written to the standard error output and to **short_outfile**.

nokdel: A stopping condition: stop if the number of iterations used so far is at least **kmin** and, for each of the most recent NNOT (defined in *src/lib/mlp/optchk.c*) sequences of **nfreq** iterations, the number right and the number right minus number wrong have both failed to increase by at least **nokdel** during the sequence.

lbfgs_mem: This value is used for the **m** argument of the LBFGS optimizer (if that optimizer is used, i.e. only if there is no Boltzmann pruning). This is the number of corrections used in the bfgs update. Values less than 3 are not recommended; large values will result in excessive computing time, as well as increased memory usage. Values in the range 3 through 7 are recommended; value must be positive.

A.2.3 Floating-Point Parm

regfac: Regularization factor. The error value that a training run attempts to minimize, contains a term consisting of **regfac** times half the average of the squares of the network weights. (The use of a regularization factor often improves the generalization performance of a neural network, by keeping the size of the weights under control.) This parm must always be set, even for test runs (since they also compute the error value, which always uses **regfac**); however, its effect can be nullified by just setting it to 0.

alpha: A parm required by the **type_1** error function: see Section A.4.2.2.2.

temperature: For Boltzmann pruning: see the switch parm **boltzmann**. A higher temperature causes more severe pruning.

egoal: A stopping condition: stop when error becomes less than or equal to **egoal**.

gwgoal: A stopping condition: stop when $|\mathbf{g}| / |\mathbf{w}|$ becomes less than or equal to **gwgoal**, where **w** is the vector of network weights and **g** is the gradient vector of the error with respect to **w**.

errdel: A stopping condition: stop if the number of iterations used so far is at least **kmin** and the error has not decreased by at least a factor of **errdel** over the most recent block of **nfreq** iterations.

oklvl: The value of the highest network output activation produced when the network is run on a pattern (the position of this highest activation among the output nodes is the hypothetical class) can be thought of as a measure of confidence. This confidence value is compared with the threshold **oklvl**, in order to decide whether to classify the pattern as belonging to the hypothetical class, or to reject it, i.e. to consider its class to be unknown because of insufficient confidence that the hypothetical class is the correct class. The numbers and percentages of the patterns that *mlp* reports as *correct*, *wrong*, and *unknown*, are affected by **oklvl**: a high value of **oklvl** generally increases the number of unknowns (a bad thing) but also increases the percentage of the accepted patterns that are classified correctly (a good thing). If no rejection is desired, set **oklvl** to 0. (*Mlp* uses the single **oklvl** value specified for a run; but if the switch parm **do_cvr** is set to **true**, then *mlp* also makes a full *correct vs. rejected* table for the network (for the finished network if a training run). This table shows the (number correct) / (number accepted) and (number unknown) / (total number) percentages for each of several standard **oklvl** values.)

trgoff: This number sets how mildly the target values for network output activations vary between their “low” and “high” values. If **trgoff** is 0 (least mild, i.e. most extreme, effect), then the low target value is 0 and the high, 1; if **trgoff** is 1 (most mild effect), then low and high targets are both (1 / **nouts**); if **trgoff** has an intermediate value between 0 and 1, then the low and high targets have intermediately mild values accordingly.

scg_earlystop_pct: This is a percentage that controls how soon a hybrid SCG/LBFGS training run (hybrid training can be used only if there is to be no Boltzmann pruning) switches from SCG to LBFGS. The switch is done

the first time a check (checking every **nfreq**'th iteration) of the network results finds that every class-subset of the patterns has at least **scg_earlystop_pct** percent of its patterns classified correctly. A suggested value for this parm is 60.0.

lbfgs_gtol: This value is used for the **gtol** argument of the LBFGS optimizer. It controls the accuracy of the line search routine **mcsrch**. If the function and gradient evaluations are inexpensive with respect to the cost of the iteration (which is sometimes the case when solving very large problems) it may be advantageous to set **lbfgs_gtol** to a small value. A typical small value is 0.1. **Lbfgs_gtol** must be greater than 1.e-04.

A.2.4 Switch Parms

Each of these parms has a small set of allowed values; the value is specified as a string, or less verbosely, as a code number (shown in parentheses after string form):

train_or_test:

train (0): Train a network, i.e. optimize its weights in the sense of minimizing an error function, using a training set of patterns.

test (1): Test a network, i.e. read in its weights and other parms from a file, run it on a test set of patterns, and measure the quality of the resulting performance.

purpose:

Which of two possible kinds of engine the network is to be. This is an architecture parm, so it should be left unset for a run that is to read a network weights file. The allowed values are:

classifier (0): The network is to be trained to map any feature vector to one of a small number of classes. It is to be trained using a set of feature vectors and their associated correct classes.

fitter (1): The network is to be trained to approximate an unknown function that maps any input real vector to an output real vector. It is to be trained using a set of input-vector/output-vector pairs of the function.
NOTE: this is not currently supported.

errfunc:

Type of error function to use (always with the addition of a regularization term, consisting of **regfac** times half the average of the squares of the network weights). See the formulas under "**Err, Ep, Ew**" in Section A.4.2.2.2 for the definitions of these error functions.

mse (0): Mean-squared-error between output activations and target values, or its equivalent computed using classes instead of target vectors. This is the recommended error function.

type_1 (1): Type 1 error function; requires floating-point parm **alpha** be set. (Not recommended.)

pos_sum (2): Positive sum error function. (Not recommended.)

boltzmann:

Controls whether Boltzmann pruning of network weights is to be done and, if so, the type of threshold to use:

no_prune (0): Do no Boltzmann pruning.

abs_prune (2): Do Boltzmann pruning using threshold $\exp(-|w|/T)$, where w is a network weight being considered for possible pruning and T is the Boltzmann **temperature**.

square_prune (3): Do Boltzmann pruning using threshold $\exp(-w^2/T)$, where w and T are as above.

acfunc_hids, acfunc_outs:

The types of *activation functions* to be used on the hidden nodes and on the output nodes (separately settable for each layer). These are architecture parms, so they should be left unset for a run that is to read a network weights file. The allowed values are:

sinusoid (0): $f(x) = .5 * (1 + \sin(.5 * x))$

sigmoid (1): $f(x) = 1 / (1 + \exp(-x))$ (Also called logistic function.)

linear (2): $f(x) = .25 * x$

priors:

What kind of prior weighting to use to set the final pattern-weights, which control the relative amounts of impact the various patterns have when doing the computations. These final pattern-weights remain fixed for the duration of a training run, but of course they can be changed between training runs.

allsame (0): Set each final pattern-weight to $(1 / \mathbf{npats})$. (The simplest thing to do; appropriate if the set of patterns has a natural distribution.)

class (1): Set each final pattern-weight to the class-weight of the class of the pattern concerned divided by **npats**; read the class-weights from **class_wts_infile**. (Appropriate if the frequencies of the several classes, in the set of patterns, are not approximately equal to the natural frequencies (prior probabilities), so as to compensate for that situation.)

pattern (2): Set the final pattern-weights to values read from **pattern_wts_infile** divided by **npats**. (Appropriate if none of the other settings of **priors** does satisfactory calculations (one can do whatever calculations one desires), or if one wants to dynamically change these weights between sessions of training.)

both (3): Set each final pattern-weight to the class-weight of the class of the pattern concerned, times the provided pattern-weight, and divided by **npats**; read the class-weights and pattern-weights from files **class_wts_infile** and **pattern_wts_infile**. (Appropriate if one wants to both adjust for unnatural frequencies, and dynamically change the pattern weights.)

patsfile_ascii_or_binary:

Tells *mlp* which of two supported formats to expect for the patterns file that it will read at the start of a run. (If much compute time is being spent reading ascii patsfiles, it may be worthwhile to convert them to binary format: that causes faster reading, and the binary-format files are considerably smaller.)

ascii (0): **patterns_infile** is in ascii format.

binary (1): **patterns_infile** is in binary (FORTRAN-style binary) format.

do_confuse:

true (1): Compute the confusion matrices and miscellaneous information as described in Section A.4.2.3, and include them in **short_outfile**.

false (0): Do not compute the confusion matrices and miscellaneous information.

show_acs_times_1000:

This parm need be set only if the run is to produce a **long_outfile**.

true (1): Before recording the network output activations in **long_outfile**, multiply them by 1000 and round to integers.

false (0): Record the activations as their original floating-point values.

do_cvr: (See the notes on **oklvl**.)

true (1): Produce a correct-vs.-rejected table and include it in **short_outfile**.

false (0): Do not produce a correct-vs.-rejected table.

A.3 Training the MLP in *hsfsys2*

Output files generated from *mlp* are provided in 4 subdirectories under *weights/mlp*: *digit* (containing output files from training on segmented numeric character images), *lower* (output files from training on lowercase characters), *upper* (output files from training on uppercase characters), and *const* (output files from training on both lower and uppercase characters). For example, the *digit* directory contains the input file (*h6_d.ml*) and output file (*h6_d.evt*) from running *mis2evt*, the input files (*d.set*, *h6_d.evt*, *h6_d.cl*, and *h6_d.ml*) and the output file (*h6_d.pat*) from *mis2pat2*, a second set of input files (*d.set*, *h6_d.evt*, *h7_d.cl*, and *h7_d.ml*) and the output file (*h7_d.pat*) from *mis2pat2*, and input and output files from running the program *mlp*.

The specfile used by *mlp* to train the classifier on digit images is *d.spc*. This specfile requires the input files *d.scn*, the training set *h6_d.pat*, and the testing set *h7_d.pat*, and it invokes 7 sequential pairs of *mlp* training/testing sessions. Three files are generated from each training/testing session. The following files are created from the first session: *trn_0.err* (a report of the progressive error rates achieved on the training set), *trn_0.wts* (the resulting weights trained in the session), and *tst_0.err* (a report of the error rate achieved on the testing set using the most recent set of weights from training). For the next training/testing session, training resumes with the MLP network initialized to the weights contained in *trn_0.wts*. The output files from this session are *trn_1.err*, *trn_1.wts*, and *tst_1.err*. The weights file *trn_1.wts* is then used as input to the next session and so on until the final session is complete. The files *trn_6.err* and *trn_6.wts* contain the final results of training and *tst_6.err* contains the error rate achieved by using the final set of weights to classify the testing set contained in *h7_d.pat*.

As can be seen from the lists above, there are numerous parameters to be specified in the specfile for running the program *mlp*. A good strategy for training the MLP on a new classification problem is to first work with a single training/testing session, surveying different combinations of parameter settings until a reasonable amount of training is achieved within the first 50 iterations, for example. This typically involves using a relatively high value for regularization (such as 2.0 with handprint character recognition); varying the number of hidden nodes in the network; and trying different levels of temperature, typically incrementing or decrementing by powers of 10. For handprint character classification, the number of hidden neurodes should be set to equal or greater than the number of input KL features, and a temperature of 1.0e-4 works well.

Once reasonable training is achieved, these parameters should remain fixed, and successive sessions of training/testing are performed according to a schedule of decreasing regularization. For handprint character classification it works well to specify about 50 iterations for each training session, and to use a regularization factor schedule starting at 2.0 and decreasing to 1.0, 0.5, 0.2, 0.1, 0.01, and 0.001 for each successive training session. This process of multiple training/testing sessions initiates MLP training within a reasonable solution space, and then enables the machine learning to refine its solution so that convergence is achieved while maintaining a high level of generalization by controlling the dynamics of constructing well behaved decision surfaces. The intermediate testing sessions allow one to evaluate the progress made on an independent testing set, so that a judgment can be made as to whether incremental gains in training have reached diminishing returns. The theory behind the control of dynamical changes within the MLP learning process is discussed in Reference [8].

Training the MLP in this fashion generates superior decision surfaces thus providing more robust activations for use as confidence values when rejecting confusing classification. This improvement in accuracy does however come with a cost. The program *mlp* is computationally intense. For example, the training of the weights in *weights/mlp/digits* required approximately 5.5 days of continuous CPU time on a Sun SPARCstation 2 with a Weitek CPU upgrade. This process is of course done once off-line, and then the resulting weight files are reused over and over by the actual recognition system.

A.4 Explanation of the output produced during MLP training

When the program *mlp* does a training run, it writes output to the standard error and writes the same output to the **short_outfile** specified in the specfile. The purpose of this section is to explain the meaning of this output. (*Mlp* produces similar output for a testing run except that the “training progress” part is missing.)

A.4.1 Pattern-Weights

As a preliminary, it will be helpful to discuss the “pattern-weights” which *mlp* uses, since they are used in the calculations of many of the values shown in the output. The pattern-weights are “prior” weights, one for each pattern;² they remain constant during a training (or testing) run, although it is possible to do a training “meta-run” that is a sequence of training runs and to change the pattern-weights between the runs. The setting of the pattern-weights is controlled by the **priors** value set in the specfile and may be affected by provided data files, as follows (in all cases, the division by N is merely a normalization that slightly reduces the amount of calculation needed later):

allsame: if **priors** is **allsame** then each pattern-weight is set to $(1/N)$, where N is the number of patterns.

class: if this is the **priors** value, then a file of class-weights must be supplied; each pattern-weight is set to the class-weight of the class of the corresponding pattern, divided by N .

pattern: a file of (original) pattern-weights must be supplied; each of them is divided by N to produce the corresponding pattern-weight.

both: files of class-weights and (original) pattern-weights must both be supplied; each pattern-weight is then set to the class-weight of the class of the corresponding pattern, times the corresponding (original) pattern-weight, divided by N .

The pattern-weights are used in the calculation of the error value that *mlp* attempts to minimize during training: when the training patterns are sent through the network, each pattern produces an error contribution, which gets multiplied by the pattern-weight for that pattern before being added to an error accumulator (Section A.4.2.2.2). The pattern-weights are also involved in the calculations of several other quantities besides the error value; all these uses are described below. Reference [8] discusses the use of class-based prior weights (Section 5.4, pages 10-11), which correspond to the **class** setting of **priors**.

A.4.2 Explanation of Output

A.4.2.1 Header

The first part of the output is a “header” showing the specfile parameter values. Here is the header of the short outfile *weights/mlp/digit/trn_0.err* produced by the first training run of a sequence of runs used to train the digits classifier:

2. A pattern is a feature-vector/class or feature-vector/target-vector pair.

```

Classifier MLP
Training run
Patterns file: h6_d.pat; using all 61094 patterns
Final pattern-wts: set all equal,
  no files read
Error function: sum of squares
Reg. factor: 2.000e+00
Activation fns. on hidden, output nodes: sinusoid, sinusoid
Nos. of input, hidden, output nodes: 128, 128, 10
Boltzmann pruning, thresh. exp(-w^2/T), T 1.000e-04
Will use SCG
Initial network weights: random, seed 12347
Final network weights will be written as file trn.wts.0
Stopping criteria (max. no. of iterations 50):
(RMS err) <= 0.000e+00 OR
(RMS g) <= 0.000e+00 * (RMS w) OR
(RMS err) > 9.900e-01 * (RMS err 10 iters ago) OR
(OK - NG count) < (count 10 iters ago) + 1. (OK level: 0.000)
Long outfile not made

```

SCG: doing <= 50 iterations; 17802 variables.

A.4.2.2 Training Progress

The next part of the output lists a running update on the training progress. The first few lines of training progress reported are:

```

pruned  282   28  310  C  1.46372e+05  H  2.33407e+04  R  84.05  M  0.00  T  0.0851
  Iter  Err (  Ep  Ew)   OK  UNK   NG   OK  UNK  NG
    0  0.691 (0.557 0.289)  5999   0 55095 =  9.8  0.0 90.2 %
  0.0  0  5 19  0  0 66 11  0  0  1
pruned  363   25  388  C  1.51345e+05  H  2.63755e+04  R  82.57  M  -0.01  T  0.0853
pruned  419   27  446  C  1.46145e+05  H  2.63513e+04  R  81.97  M  -0.01  T  0.0853
pruned  449   28  477  C  1.64731e+05  H  2.68884e+04  R  83.68  M  -0.01  T  0.0849
pruned  472   32  504  C  1.72004e+05  H  2.71783e+04  R  84.20  M  -0.01  T  0.0846
pruned  490   32  522  C  1.39698e+05  H  2.70099e+04  R  80.67  M  -0.01  T  0.0845
pruned  514   37  551  C  1.88008e+05  H  2.73029e+04  R  85.48  M  -0.01  T  0.0844
pruned  534   38  572  C  1.49777e+05  H  2.70401e+04  R  81.95  M  -0.01  T  0.0838
pruned  540   40  580  C  1.93717e+05  H  2.72770e+04  R  85.92  M  -0.01  T  0.0814
pruned  539   38  577  C  1.66433e+05  H  2.56886e+04  R  84.57  M  -0.01  T  0.0774
pruned  548   37  585  C  2.07274e+05  H  2.71835e+04  R  86.89  M  -0.01  T  0.0741
  10  0.488 (0.307 0.268) 10906   0 50188 = 17.9  0.0 82.1 %
  5.2 15 93  5  6  5  6  8  9  6 16

```

The line

```

  Iter  Err (  Ep  Ew)   OK  UNK   NG   OK  UNK  NG

```

comprises column headers that pertain to those subsequent lines that begin with an integer (“first progress lines”); each first progress line is followed by a “second progress line”, and there are “pruning lines” if Boltzmann pruning is used. These three types of lines are discussed below, second progress lines first because some of the calculations used to produce them are later used to make the first progress lines.

A.4.2.2.1 Second progress lines

These are the lines that begin with fractional numbers; the first of them in the above example is

```
0.0 0 5 19 0 0 66 11 0 0 1
```

Ignoring for a moment the first value in such a line, the remaining values are the “percentages” right by class, which *mlp* calculates as follows. It maintains three pattern-weight-accumulators for each class:

$a_i^{(r)}$ = right pattern-weight-accumulator for correct class i

$a_i^{(w)}$ = wrong pattern-weight-accumulator for correct class i

$a_i^{(u)}$ = unknown (rejected) pattern-weight-accumulator for correct class i

When *mlp* sends a training pattern through the network the result is an output activation for each class; the hypothetical class is, of course, whichever class receives the highest activation. If the highest activation equals or exceeds the rejection threshold **oklvl** set in the specfile, then *mlp* accepts its result for this pattern, and adds its pattern-weight (Section A.4.1) either to $a_i^{(r)}$ or to $a_i^{(w)}$ - where i is the correct class of the pattern - according to whether the network classified the pattern rightly or wrongly. Otherwise, i.e. if the highest activation is less than **oklvl**, *mlp* adds the pattern-weight to $a_i^{(u)}$. These accumulators reach their final values as a result of the sending of all the training patterns through the network. *Mlp* then defines the right “percentage” of correct class i to be

$$\frac{100 \times a_i^{(r)}}{a_i^{(r)} + a_i^{(w)} + a_i^{(u)}}$$

It shows these values, rounded to integers, in the second progress lines, as the values after the first one. For example, the second progress line above shows that the right “percentages” of correct classes 0 and 1 are 0 and 5.³

If **priors** is **allsame** then the pattern-weights are all equal and so $a_i^{(r)}$, etc. are the numbers classified rightly, etc. times this single pattern-weight; the pattern-weight cancels out between the numerator and denominator of the above formula, so that the resulting value really is the percentage of the patterns of class i that the network classified rightly. If **priors** has a value other than **allsame** - i.e. **class**, **pattern**, or **both** - then the right “percentages” of the classes are not the simple percentages but rather are weighted quantities, which may make more sense than the simple percentages if some patterns should have more impact than others, as indicated by their larger weights.⁴

As for the first value of a second progress line, this is merely the minimum of the right “percentages” of the classes, but shown rounded to the nearest tenth rather than to the nearest integer. This minimum value shows how the network is doing on its “worst” class.⁵

3. In this case the classes whose “index numbers” are 0 through 9 happen to be the digits 0 through 9, but that is entirely coincidental. The classes could be letters, fingerprint classes, phonemes, or who knows what. In this discussion, “class i ” merely means the class whose index number, numbering starting at 0, is i . Note also that although the software uses class index numbers that start at 0, the class index numbers it writes to **long_outfile** start at 1.

4. In particular, if the training patterns set is such that the proportions of the patterns belonging to the various classes are not approximately equal to the natural frequencies of the classes, then it may be a good idea to use class-weights (**priors** set to **class**, and class-weights provided in a file) to compensate for the erroneous distribution. See [8].

5. When *mlp* uses hybrid SCG/LBFGS training rather than only SCG - it does this only if pruning is not specified - it switches from SCG to LBFGS when the minimum reaches or exceeds a specified threshold, **scg_earlystop_pct**.

A.4.2.2.2 First progress lines

These are the lines that begin with an integer. The column headings -- which pertain to these lines -- and the first of these lines in the example, are:

```

Iter  Err (  Ep  Ew)      OK   UNK   NG      OK   UNK   NG
      0 0.691 (0.557 0.289) 5999   0 55095 =  9.8  0.0 90.2 %

```

The values in a first progress line have the following meanings:

Iter: Training iteration number, numbering starting at 0. A first progress line (and second progress line) are produced every **nfreq**'th iteration (set in the specfile).

Err, Ep, Ew: The calculations leading to these values are as follows.

$$\begin{aligned}
 N &= \text{number of patterns} \\
 n &= \text{number of classes} \\
 a_{ij} &= \text{activation produced by pattern } i \text{ at output node } j \text{ (i.e. class } j) \\
 t_{ij} &= \text{target value for } a_{ij} \\
 w_i^{(\text{pat})} &= \text{pattern-weight of pattern } i \text{ (Section A.4.1)} \\
 E_i^{(\text{pat, mse})} &= \sum_{j=0}^{n-1} (a_{ij} - t_{ij})^2 \\
 &= \text{error contribution for pattern } i \text{ if } \mathbf{errfunc} \text{ is } \mathbf{mse} \\
 E_1^{(\text{mse})} &= \frac{1}{2n} \sum_{i=0}^{N-1} w_i^{(\text{pat})} E_i^{(\text{pat, mse})} \\
 E_i^{(\text{pat, type1})} &= 1 - \frac{1}{1 + \sum_{j \neq k} \exp(-\alpha (a_{ik} - a_{ij}))}, \text{ where } k \text{ is correct class of pattern } i \\
 &= \text{error contribution for pattern } i \text{ if } \mathbf{errfunc} \text{ is } \mathbf{type_1} \text{ (}\alpha \text{ is } \mathbf{alpha}) \\
 E_1^{(\text{type1})} &= \frac{1}{n} \sum_{i=0}^{N-1} w_i^{(\text{pat})} E_i^{(\text{pat, type1})} \\
 E_i^{(\text{pat, possum})} &= \sum_{j=0}^{n-1} (10|a_{ij} - t_{ij}| + 1) |a_{ij} - t_{ij}| \\
 &= \text{error contribution for pattern } i \text{ if } \mathbf{errfunc} \text{ is } \mathbf{pos_sum} \\
 E_1^{(\text{possum})} &= \frac{1}{n} \sum_{i=0}^{N-1} w_i^{(\text{pat})} E_i^{(\text{pat, possum})} \\
 E_1 &= E_1^{(\text{mse})}, E_1^{(\text{type1})}, \text{ or } E_1^{(\text{possum})}, \text{ according to } \mathbf{errfunc} \\
 \mathbf{Ep} &= E_1 \text{ if } \mathbf{errfunc} \text{ is } \mathbf{pos_sum}, \sqrt{2E_1} \text{ otherwise} \\
 s^{(\text{wsq})} &= \text{half of mean squared network weight} \\
 \mathbf{Ew} &= \sqrt{2s^{(\text{wsq})}} \\
 E &= E_1 + \mathbf{regfac} \times s^{(\text{wsq})} \\
 \mathbf{Err} &= \sqrt{2E}
 \end{aligned}$$

Mlp prints the **Err**, **Ep** and **Ew** values as defined above. Note that the value *mlp* attempts to minimize is *E*, but presumably the same effect would be had by attempting to minimize **Err**, since it is an increasing function of *E*.

OK, UNK, NG, OK, UNK, NG: “Numbers” of patterns OK (classified correctly), UNKnown (rejected), and wrong or No Good (classified incorrectly), then the corresponding “percentages”. *Mlp* calculates these values as follows. It adds up the by-class accumulators $a_i^{(r)}$, $a_i^{(w)}$, and $a_i^{(u)}$ defined earlier to make overall accumulators, where n is the number of classes:

$$a^{(r)} = \sum_{i=0}^{n-1} a_i^{(r)}$$

$$a^{(w)} = \sum_{i=0}^{n-1} a_i^{(w)}$$

$$a^{(u)} = \sum_{i=0}^{n-1} a_i^{(u)}$$

It computes “numbers” right, wrong, and unknown -- the first **OK**, **NG**, and **UNK** values of a first progress line -- as follows, where N is the number of patterns and square brackets denote rounding to an integer:

$$a^{(rwu)} = a^{(r)} + a^{(w)} + a^{(u)}$$

$$n^{(r)} = [Na^{(r)} / a^{(rwu)}] = \text{“number” right}$$

$$n^{(w)} = [Na^{(w)} / a^{(rwu)}] = \text{“number” wrong}$$

$$n^{(u)} = N - n^{(r)} - n^{(w)} = \text{“number” unknown}$$

From these “numbers”, *mlp* computes corresponding “percentages” -- the second **OK**, **NG**, and **UNK** values -- as follows:

$$p^{(r)} = [100 \times n^{(r)} / N]$$

$$p^{(w)} = [100 \times n^{(w)} / N]$$

$$p^{(u)} = [100 \times n^{(u)} / N]$$

If **priors** is **allsame** then since the pattern-weights are all equal, cancellation of the single pattern-weight occurs between the numerators and denominators of the formulas above for $n^{(r)}$ and $n^{(w)}$, so that they really are the numbers of patterns classified rightly and wrongly, and then it is obvious that $n^{(u)}$ really is the number unknown and that $p^{(r)}$, etc. really are the percentages classified rightly, etc.

A.4.2.2.3 Pruning lines (optional)

These lines, which begin with “pruned”, appear if Boltzmann pruning is specified (**boltzmann** set to **abs_ -prune** or **square_prune** in specfile, and a **temperature** set). The first pruning line of the example is

```
pruned 282 28 310 C 1.46372e+05 H 2.33407e+04 R 84.05 M 0.00 T 0.0851
```

Regardless of **nfreq**, *mlp* writes a pruning line every time it performs pruning. The first three values of a pruning line are the numbers of network weights that *mlp* pruned (temporarily set to zero) in the first weights layer, in the second

layer, and in both layers together. The remaining values announced by the letters **C**, **H**, **R**, and **M**, are calculated as follows (the value announced by **T** actually is not calculated correctly, and should be ignored):

$$\begin{aligned}
 n^{(\text{wts})} &= \text{number of network weights (both layers)} \\
 n^{(\text{pruned})} &= \text{number of weights pruned} \\
 n^{(\text{unpruned})} &= n^{(\text{wts})} - n^{(\text{pruned})} \\
 w^{(\text{max})}, w^{(\text{min})} &= \text{maximum \& minimum absolute values of unpruned weights} \\
 \mathbf{C} &= n^{(\text{unpruned})} ((\log w^{(\text{max})} - \log w^{(\text{min})}) / (\log 2) + 1) = \text{capacity} \\
 s^{(\log \text{abs})} &= \text{sum of logarithms of absolute values of unpruned weights} \\
 s^{(w12)} &= s^{(\log \text{abs})} / (\log 2) + n^{(\text{unpruned})} (1 - (\log w^{(\text{min})}) / (\log 2)) \\
 \mathbf{H} &= \mathbf{C} - s^{(w12)} = \text{entropy} \\
 \mathbf{R} &= 100 \times s^{(w12)} / \mathbf{C} = \text{redundancy} \\
 \mathbf{M} &= \text{mean of unpruned weights}
 \end{aligned}$$

A.4.2.3 Confusion Matrices and Miscellaneous Information (Optional)

If **do_confuse** is set to **true** in the specfile, the next part of the output consists of two “confusion matrices” and some miscellaneous information:

```

oklv1 0.00
# Highest two outputs (mean) 0.856 0.126; mean diff 0.730
key  name
0  0
1  1
2  2
3  3
4  4
5  5
6  6
7  7
8  8
9  9
# key:      0  1  2  3  4  5  6  7  8  9
# row: correct, column: actual
#   0: 5754 43  6 11 65  6 33  3 10  8
#   1:  0 6547 36 20 14 31 31 12 17  2
#   2: 28 26 5826 66 23  7 26 29 47  8
#   3:  9 10 76 5827  8 50  1 44 39 21
#   4: 14 14 22  2 5828  0 23  4 35 68
#   5: 34 15  5 125 21 5502 55  4 52 25
#   6: 18 23 14  0  7 18 5970  0  1  0
#   7:  3 12 11  6 30  1  0 6186 18 67
#   8: 17 129 56 136 65 81 14 17 5393 58
#   9: 10 11  1 16 58  7  0 79 37 5856
# unknown
# *  0  0  0  0  0  0  0  0  0  0

percent of true IDs correctly identified (rows)
97 98 96 96 97 94 99 98 90 96
percent of predicted IDs correctly identified (cols)
98 96 96 94 95 96 97 97 95 96

```

```

# mean highest activation level
# row: correct, column: actual
# key:      0  1  2  3  4  5  6  7  8  9
# 0:      91  51  39  44  45  35  43  30  36  33
# 1:       0  90  40  43  48  42  37  41  47  29
# 2:      44  41  86  46  37  42  47  46  43  25
# 3:      53  45  45  88  32  52  27  50  48  43
# 4:      35  45  52  41  82   0  44  38  44  56
# 5:      46  37  47  53  38  85  49  32  44  51
# 6:      35  50  35   0  42  48  91   0  14   0
# 7:      39  39  47  32  45  22   0  89  36  49
# 8:      40  56  36  44  41  47  38  35  81  40
# 9:      40  36  16  50  50  46   0  66  50  87
# unknown
# *      0  0  0  0  0  0  0  0  0  0

```

Histogram of errors, from 2^{-10} to 1

82893	33517	46509	62676	80193	94688	90535	64963	34608	14910	5448
13.6	5.5	7.6	10.3	13.1	15.5	14.8	10.6	5.7	2.4	0.9%

The first line of this optional section of the output shows the value of the rejection threshold **oklvl** set in the specfile (this was already shown in the header). The next line shows the mean values, over the training patterns as sent through the network at the end of training, of the highest and second-highest output node values, and the mean difference of these values. Next is a table showing the short classname (“key”) and long classname (“name”) of each class. In this example the keys and names are the same, but in general the names can be quite long whereas the keys must be no longer than two characters in length: the short keys are used to label the confusion matrices.

Next are the confusion matrices of “numbers” and of “mean highest activation level”. *Mlp* has the following accumulators:

$$\begin{aligned}
 a_{ij}^{(\text{patwts})} &= \text{pattern-weight accumulator for correct class } i \text{ and hypothetical class } j \\
 a_{ij}^{(\text{highac})} &= \text{high-activation accumulator for correct class } i \text{ and hypothetical class } j \\
 a_i^{(\text{highac, u})} &= \text{high-activation unknown accumulator for correct class } i
 \end{aligned}$$

If a pattern sent through the network produces a highest activation that meets or exceeds **oklvl** -- so that *mlp* accepts its result for this pattern -- then *mlp* adds its pattern-weight to $a_{ij}^{(\text{patwts})}$ and adds the highest activation to $a_{ij}^{(\text{highac})}$ where i and j are the correct class and hypothetical class of the pattern. Otherwise, i.e. if *mlp* finds the pattern to be unknown (rejects the result), it adds its pattern-weight to $a_i^{(\text{u})}$ (Section A.4.2.2.1) and adds the highest activation to $a_i^{(\text{highac, u})}$, where i is the correct class of the pattern. After it has processed all the patterns, *mlp* calculates the confusion matrix of “numbers” and its “unknown” line; some additional information concerning the rows and columns of that matrix; and the confusion matrix of “mean highest activation level” and its “unknown” line, as follows.

First define some notation:

$$\begin{aligned}
 N_i^{(\text{pat})} &= \text{number of patterns of correct class } i \\
 n_{ij}^{(\text{confuse})} &= \text{value in row } i \text{ and column } j \text{ of first confusion matrix (of “numbers”)} \\
 n_i^{(\text{confuse, u})} &= i^{\text{th}} \text{ value of “unknown” line at bottom of first confusion matrix} \\
 p_i^{(\text{r, row})} &= i^{\text{th}} \text{ value of “percent of true IDs correctly identified (rows)” line} \\
 p_j^{(\text{r, col})} &= j^{\text{th}} \text{ value of “percent of predicted IDs correctly identified (cols)” line} \\
 h_{ij}^{(\text{confuse})} &= \text{value in row } i \text{ and column } j \text{ of second confusion matrix (of “mean highest activation level”)} \\
 h_i^{(\text{confuse, u})} &= i^{\text{th}} \text{ value of “unknown” line at bottom of second confusion matrix}
 \end{aligned}$$

Mlp calculates the values as follows, where $a_i^{(r)}$, $a_i^{(w)}$, and $a_i^{(u)}$ are as defined in Section A.4.2.2.1 and square brackets again denote rounding to an integer:⁶

$$n_{ij}^{(\text{confuse})} = \left[\frac{N_i^{(\text{pats})} a_{ij}^{(\text{patwts})}}{a_i^{(u)} + \sum_{j=0}^{n-1} a_{ij}^{(\text{patwts})}} \right]$$

$$n_i^{(\text{confuse, u})} = \left[\frac{N_i^{(\text{pats})} a_i^{(u)}}{a_i^{(r)} + a_i^{(w)} + a_i^{(u)}} \right]$$

$$p_i^{(\text{r, row})} = \left[\frac{100 \times n_{ii}^{(\text{confuse})}}{N_i^{(\text{pats})} - n_i^{(\text{confuse, u})}} \right]$$

$$p_j^{(\text{r, col})} = \left[\frac{100 \times n_{jj}^{(\text{confuse})}}{\sum_{i=0}^{n-1} n_{ij}^{(\text{confuse})}} \right]$$

$$h_{ij}^{(\text{confuse})} = \left[\frac{100 \times a_{ij}^{(\text{highac})}}{n_{ij}^{(\text{confuse})}} \right]$$

$$h_i^{(\text{confuse, u})} = \left[\frac{100 \times a_i^{(\text{highac, u})}}{n_i^{(\text{confuse, u})}} \right]$$

If **priors** is **allsame**, then since the pattern-weights are all equal, cancellation of the single pattern-weight between numerator and denominator causes $n_{ij}^{(\text{confuse})}$ above to be really the number of patterns of correct class i and hypothetical class j ; similarly, $n_i^{(\text{confuse, u})}$ really is the number of patterns of correct class i that were unknown; $p_i^{(\text{r, row})}$ and $p_j^{(\text{r, col})}$ really are the percentages that the on-diagonal -- correctly classified -- numbers in the matrix comprise of their rows and columns respectively; $h_{ij}^{(\text{confuse})}$ really is the mean highest activation level (multiplied by 100 and rounded to an integer) of the patterns of correct class i and hypothetical class j ; and $h_i^{(\text{confuse, u})}$ really is the mean highest activation level of the patterns of correct class i that were unknown. If **priors** has one of its other values, the printed values are weighted versions of these quantities.

The final part of this optional section of the output is a histogram of errors. This pertains to the absolute errors between output activations and target activations, across all output nodes (10 nodes in this example) and all training patterns (61,094 patterns in this example), when the patterns are sent through the trained network. Of the resulting set of absolute error values (610,940 values in this example), this histogram shows the number (first line) and percentage (second line) of these values that fall into each of the 11 intervals $(-\infty, 2^{-10}]$, $(2^{-10}, 2^{-9}]$, ..., $(2^{-1}, 1]$.

A.4.2.4 Final Progress Lines

The next part of the output consists of a repeat of the column-headers line, final first-progress-line, and final second-progress-line of the training progress part of the output, but with an **F** prepended to the final first-progress-line:

```

Iter  Err (  Ep  Ew)   OK  UNK   NG   OK  UNK  NG
F    50 0.156 (0.101 0.084) 58689   0  2405 = 96.1  0.0  3.9 %
90.4  97 98 96 96 97 94 99 98 90 96

```

6. The denominators of the expressions shown here for $n_{ij}^{(\text{confuse})}$ and $n_i^{(\text{confuse, u})}$ are equal, but these expressions show what the software actually calculates, rather than what it would have calculated if it had been more efficient.

A.4.2.5 Correct-vs.-Rejected Table (Optional)

If `do_cvr` is set to `true` in the specfile, the next part of the output is a correct-vs.-rejected table; the first and last few lines of this table, from the example output, are:

	thresh	right	unknown	wrong	correct	rejected
1tr	0.000000	58690	0	2404	96.07	0.00
2tr	0.050000	58690	0	2404	96.07	0.00
3tr	0.100000	58690	2	2402	96.07	0.00
4tr	0.150000	58689	12	2393	96.08	0.02
5tr	0.200000	58672	73	2349	96.15	0.12
. . .						
48tr	0.975000	15760	45333	1	99.99	74.20
49tr	0.980000	13687	47406	1	99.99	77.60
50tr	0.985000	11519	49574	1	99.99	81.14
51tr	0.990000	8969	52124	1	99.99	85.32
52tr	0.995000	5971	55122	1	99.98	90.22

Mlp produces this table values as follows. It has a fixed array of rejection-threshold values, which have been set in an unequally-spaced pattern that works well, and it uses three pattern-weight-accumulators for each threshold:

$$\begin{aligned}
 t_k &= k^{\text{th}} \text{ threshold} \\
 a_k^{(\text{cvr}, r)} &= \text{right pattern-weight-accumulator for } k^{\text{th}} \text{ threshold} \\
 a_k^{(\text{cvr}, w)} &= \text{wrong pattern-weight-accumulator for } k^{\text{th}} \text{ threshold} \\
 a_k^{(\text{cvr}, u)} &= \text{unknown pattern-weight-accumulator for } k^{\text{th}} \text{ threshold}
 \end{aligned}$$

As *mlp* sends each pattern through the finished network,⁷ it loops over the thresholds t_k : for each k , it compares the highest network activation produced for the pattern with t_k to decide whether the pattern would be accepted or rejected if t_k were used. If accepted, it adds the pattern-weight of that pattern either to $a_k^{(\text{cvr}, r)}$ or to $a_k^{(\text{cvr}, w)}$ according to whether it classified the pattern rightly or wrongly; if rejected, it adds the pattern-weight to $a_k^{(\text{cvr}, u)}$. After all the patterns have been through the network, *mlp* finishes the table as follows. For each threshold t_k it calculates the following values:

$$\begin{aligned}
 a^{(\text{cvr}, r w u)} &= a_k^{(\text{cvr}, r)} + a_k^{(\text{cvr}, w)} + a_k^{(\text{cvr}, u)} \\
 n^{(\text{cvr}, r)} &= [N a_k^{(\text{cvr}, r)} / a^{(\text{cvr}, r w u)}] = \text{“number right”} \\
 n^{(\text{cvr}, w)} &= [N a_k^{(\text{cvr}, w)} / a^{(\text{cvr}, r w u)}] = \text{“number wrong”} \\
 n^{(\text{cvr}, u)} &= N - n^{(\text{cvr}, r)} - n^{(\text{cvr}, w)} = \text{“number unknown” (rejected)} \\
 p^{(\text{cvr}, \text{corr})} &= 100 \times n^{(\text{cvr}, r)} / (n^{(\text{cvr}, r)} + n^{(\text{cvr}, w)}) = \text{“percentage correct”} \\
 p^{(\text{cvr}, \text{rej})} &= 100 \times n^{(\text{cvr}, u)} / N = \text{“percentage rejected”}
 \end{aligned}$$

Mlp then writes a line of the table. The values of the line are the threshold index k plus 1 with “tr”⁸ appended, t_k (“thresh”), $n^{(\text{cvr}, r)}$ (“right”), $n^{(\text{cvr}, u)}$ (“unknown”), $n^{(\text{cvr}, w)}$ (“wrong”), $p^{(\text{cvr}, \text{corr})}$ (“correct”), and $p^{(\text{cvr}, \text{rej})}$ (“rejected”). If `priors` is `allsame` then, since all pattern-weights are the same, cancellation of the single pattern-weight occurs between numerator and denominator in the above expressions for $n^{(\text{cvr}, r)}$ and $n^{(\text{cvr}, w)}$, so they really are the number of patterns classified rightly and wrongly if threshold t_k is used; and then it is obvious that $n^{(\text{cvr}, u)}$ really is the number of patterns unknown for this threshold, $p^{(\text{cvr}, \text{corr})}$ really is the percentage of the patterns accepted at this

7. If `do_cvr` is `true` then *mlp* calculates a correct-vs.-rejected table, but only for the final state of the network in the training run, of course: if it produced such a table for each training iteration, its output would be extremely verbose.

8. for “training”; the correct-vs.rejected table for a test run uses “ts”

threshold that were classified correctly, and $p^{(cvt, rej)}$ really is the percentage of the N patterns that were rejected at this threshold. If **priors** has one of its other values, then the tabulated values are weighted versions of these quantities.

A.4.2.6 Final Information

The final part of the output shows miscellaneous information:

```
Iter 50; ierr 1 : iteration limit
Used 51 iterations; 155 function calls; Err 0.156; |g|/|w| 2.444e-04
Rms change in weights 0.241

User+system time used: 71087.7 (s) 19:44:47.7 (h:m:s)
Wrote weights as file trn.wts.0
```

The first line here shows what iteration the training run ended on, and the value and meaning of the return code *ierr*, which indicates why *mlp* stopped its training run: in the example, the specified maximum number of iterations (**niter_max**), 50, had been used. (This training run was actually the first run of a sequence that were used; its initial network weights were random, but each subsequent run used the final weights of the preceding run as its initial weights. The only parameter varied from one run to the next was the regularization factor **regfac**, which was decreased at each step: successive regularization. Each run was limited to 50 iterations, and it was assumed that this small iteration limit would be reached before any of the other stopping conditions were satisfied. When sinusoid activation functions are used, as in this case, best training requires that successive regularization be used. If sigmoid functions are used, it is just as well to do only one training run, and in that case one should probably set the iteration limit to a large number so that training will be stopped by one of the other conditions, such as an error goal (**egoal**.)

The next line shows: how many iterations *mlp* used (counting the 0th iteration; yes, this is stupid after it already said what iteration it stopped on); how many calls of the error function it made; the final error value; and the final size of the error gradient vector (square root of sum of squares), normalized by dividing it by the final size of the weights. The next line shows the root-mean-square of the change in weights, between their initial values and their final values. The next line shows the combined user and system time used by the training run.⁹ The final line merely reports the name of the file to which *mlp* wrote the final weights.

9. Setting the initial network weights, reading the patterns file, and other (minor) setup work, are not timed.