Appendix A

Specimen Spitbol Program: A Monte-Carlo Feature-Finder

As the idea of Monte-Carlo textual feature-finding is one of the more interesting contributions of this project, there follows a listing of the program TEFF, written in the Spitbol dialect of Snobol4. This is the most advanced of the Monte-Carlo Feature-Finders written for the present thesis.

```
**  TEFF -- Program to find distinctive textual features :
**  Copyright (c) Richard Forsyth, 1994, 1995.
*
**  First version : 19-Oct-94
**  Last revision : 27-Jun-95
*

**  Global settings & patterns :
    &anchor = 1
**  &trim & &anchor better as 0 ??
    alphanum = "1234567890" &ucase &lcase "_"
    alphapat = break(alphanum) span(alphanum) . word
    qq = '"' "'" ; null = ''
    nongram = 'Zz$$'
    tiny = 1.2e-23
    maxtries = 3600
    strings = 2048
**  rather slow.

**  Function declarations :
    define('vadd(v,n,tv)j')
    define('tote(s,cats,gramvec)j,n')
    define('chis(t,cats,gv)e,j')

define('loadsubs(maxtries,strings,cats,gramsize)c,card,fpos,j,k,line,n,s,t
')
    define('randstr(line,g,n)r')
    define('tallies(stab,cats,n)c,card,j,k,line,tc')

-include "sno4.inc"
-include "iran.inc"
-include "util.inc"
-include "teff.inc"
-include "ps.inc"

**  Prelude :

    terminal = 'TEFF, version 1.1 -- ' date()
    terminal = 'Text-Extending Feature Finder'
    terminal = 'Please give training directory filename (.d1) :'
```

```
    filename = input
    input(.dirline,1,filename)                  :f(dudmain)
    mainfile = filename
    terminal = 'Please give o/p file for text markers (.tff) :'
    teffile = input
    output(.dump,12,teffile)                    :f(dudteff)
    terminal = 'Please give output listing file (.out) :'
    outfile = input
    output(.output,10,outfile)                  :f(dudfout)
    output = 'TEFF output;  date: ' date()
preloop  terminal = 'Please give pre-stretched maximum string size (2..9) :'
    gramsize = input
    (integer(gramsize) gt(gramsize,1) le(gramsize,9))  :f(preloop)
    gramsize = +gramsize
    terminal = 'Ignore fragments of earlier substrings (n=No) ?'
    dropmode = replace(input,&ucase,&lcase)
    terminal = dropmode                         :(main)

dudmain  terminal = 'Cannot read file ' filename ' !' :(end)
dudfout  terminal = 'Cannot open ' outfile ' !'  :(end)
dudteff  terminal = 'Cannot open ' teffile ' !'  :(end)

**  Main Program :
main  &dump = 2 ; t1 = time()
    output = 'gramsize = ' gramsize
    output = 'dropmode = ' dropmode
    readdir(60) ; terminal = nf ' training files.'
    cats = nf ; gt = 0.0
    endfile(1)
    entries = 0
**  Compute base frequencies :
    stab = table(chop(strings / 2))
    totchars = array(cats) ; symbols = array(cats)
    frac = array(cats)
    entries = loadsubs(maxtries,strings,cats,gramsize)
    terminal = entries

**  Do the work :
usefreqs  s = 0
    stab = sort(stab)
    x = prototype(stab)
    x break(',') . ns
    ns = +ns
    terminal = ns ' different symbols: ' x
    gramtab = table(201) ; freq = table(251)
    used = null
tallying  j = 0 ; s = 0
    gt = tallies(stab,cats,ns)
    terminal = gt ' bytes.'
totloop  j = lt(j,cats) j + 1                   :f(mainloop)
    frac[j] = totchars[j] / (gt + 0.5)
    output = terminal = 'proportion in class ' j ' = ' frac[j]  :(totloop)
mainloop  s = lt(s,ns) s + 1                    :f(endprog)
    t = tote(s,cats,stab[s,2])
    le(t,12)                                    :s(delement)
**  skips if too rare to matter.
    x = chis(t,cats,stab[s,2])
    lt(x,cats)                                  :s(delement)
```

```
    kept = kept + 1
    gramtab<stab[s,1]> = x
    freq<stab[s,1]> = stab[s,2]                :(mainloop)
**  records frequencies of non-trivial entries.

delement  stab[s,2] = null                     :(mainloop)
**  supposed to save memory.

endprog  &dump = 0
    dump = mainfile ' ' date()
**  sort table & print details.
    output = ; output = 'Grams kept = ' kept
    output = ; stab =
    gramvec = rsort(gramtab,2)
    j = postsort(gramvec,kept)
    j = gt(j,2) postsort(gramvec,kept)
    terminal = 'Swaps = ' +j
    j = 0 ; &anchor = 0
    nout = 0
**  only keep if not substring of string higher in list, unless dropmode=n :
outloop  j = lt(j,kept) j + 1                  :f(done)
    leq(dropmode,'n')                          :s(outloop0)
    used  gramvec[j,1]                         :s(outloop8)
outloop0  used = used '~' gramvec[j,1]
    dump = gramvec[j,1] ; nout = nout + 1
    js = 1
outloop1  lout = rpad(j * js,4)  lpad('`' gramvec[j,1],17)
    lout = lout '  ' gramvec[j,2]
    v = freq<gramvec[j,1]> ; c = 0 ; lout = lout '  '
outloop2  c = lt(c,cats) c + 1                 :f(outloop7)
    lout = lout '  ' chop(v[c])                :(outloop2)
outloop7  output = lout
    lt(nout,144)                               :s(outloop)f(done)
outloop8  js = -1                              :(outloop1)

done  output =
    terminal = 'TEFF finished : ' date()
    terminal = 'Runtime = ' (time() - t1) / 1000.0 ' secs.'  :(end)

**  Function definitions :


**  TALLIES :
**  Counts substrings in stab[,] :
tallies  c = 0 ; tc = 0
tallies1  c = lt(c,cats) c + 1                 :f(talliex)
    input(.card,4,filelist[c] '[-l4096]')
    terminal = output = c '  ' filelist[c]
tallies2  line = card                          :f(tallies4)
    k = size(line) ; tc = tc + k
    totchars[c] = totchars[c] + k
    j = 0
tallies3  j = lt(j,n) j + 1                    :f(tallies2)
    k = subcount(line,stab[j,1])
    stab[j,2][c] = stab[j,2][c] + k            :(tallies3)
**  loops thru all substrings.
tallies4  endfile(4)
    output = totchars[c] ' bytes.'             :(tallies1)
```

```
talliex  tallies = tc                              :(return)


**  LOADSUBS :
**  Puts random substrings into stab<> :
loadsubs  c = 0 ; j = 0 ; k = 0 ; t = 0
loadsub1  c = c + 1 ; c = gt(c,cats) 1
    input(.card,2,filelist[c] '[-l4096]')
loadsub2  line = card                              :f(loadsub4)
    gt(k,strings)                                  :s(loadsubx)
    j = j + 1
    n = size(line)
    s = randstr(line,gramsize,n)
    t = t + 1 ; gt(t,maxtries)                     :s(loadsubx)
**  save file position on unit 2 :
    fpos = set(2,0,1)
**  stretch s to maximum length :
    s = textend(s,17,2)
**  will return 'zonk' if useless (no matter).
    fpos = set(2,fpos,0)
**  file position restored.
    differ(stab<s>)                                :s(loadsub3)
    stab<s> = array(cats,0) ; k = k + 1
loadsub3  s = randstr(line,gramsize,n)
    t = t + 1 ; gt(t,maxtries)                     :s(loadsubx)
    fpos = set(2,0,1)
    s = textend(s,17,2)
    fpos = set(2,fpos,0)
    differ(stab<s>)                                :s(loadsub2)
    terminal = k '  `' s '`'
    stab<s> = array(cats,0)
    k = k + 1                                      :(loadsub2)
**
loadsub4  endfile(2)                               :(loadsub1)
loadsubx  endfile(2)
    terminal = j ' lines; ' k ' entries; ' t ' tries.'
    loadsubs = k                                   :(return)
**  cycles thru files till table full enough.


**  RANDSTR :
**  Extracts random substring from line :
randstr  randstr = ge(g,n) '!!'                    :s(return)
    r = iran(n)
    randstr = substr(line,r,iran(g))               :s(return)
    randstr = '$'                                  :(return)


**  TOTE :
**  Sums contents of freq. array.
tote  j = lt(j,cats) j + 1                         :f(return)
    gramvec[j] = gramvec[j] + 0.5
    tote = tote + gramvec[j]                       :(tote)
**  side-effect on gramvec[] slugs total towards equality.


**  CHIS :
**  Chi-squared with cats-1 d.f.
```

```
chis  j = lt(j,cats) j + 1                      :f(return)
    e = frac[j] * t
    lt(e,2.618034)                              :s(chis)
    chis = chis + (gv[j] - e) ^ 2.0 / e         :(chis)
**  omits cells with e < 2.618


END
```

```
**  TEFF -- String Extension routines :
**  Copyright (c) Richard Forsyth, 1995
*
**  First version : 20-Mar-95
**  Last revision : 27-Jun-95
*

    zonk = 'zonk'
**  non-word.

**  Function declarations :
    define('textend(m,maxsize,unit)a,z')
    define('preceder(core,unit)a,ch,k,line,maintext,next,p,rest')
    define('follower(core,unit)a,ch,k,line,maintext,next,p,rest')
                                                :(teff_end)

**  TEXTEND :
**  Extends text marker m at both ends if possible :
textend  ge(size(m),maxsize)                    :s(textends)
    a = preceder(m,unit)
    ident(a,zonk)                               :s(textendz)
    m = a m
    z = follower(m,unit)
    m = m z
    differ(a z)                                 :s(textend)f(textends)
**  exit with possibly extended string :
textends  textend = m                           :(return)
textendz  textend = zonk                        :(return)
**  zonk as result indicates insufficient freq.
**  better keep preceder call before follower.


**  PRECEDER :
**  Finds constant precursor of substring core, if there is 1 :
preceder  a = &anchor ; &anchor = 0
    next = null ; k = 0
**  caller must save position on file unit.
    input(.maintext,unit) ; rewind(unit)
    p = len(1) . ch core rem . rest
precede1  line = maintext                       :f(precedes)
precede2  line p                                :f(precede1)
    k = lt(k,40) k + 1                          :f(precedes)
    line = rest
    ident(next)                                 :f(precede4)
**  gets here first time only :
    next = ch                                   :(precede2)
```

```
**
precede4  ident(next,ch)                          :s(precede2)
**  not always same predecessor :
    &anchor = a ; preceder = null                 :(return)
**  40 instances without exception is plenty :
precedes  &anchor = a
    next = le(k,2) zonk
    preceder = next                               :(return)
**  if less than 3 occurrences, consistency is trivial.


**  FOLLOWER :
**  Sees whether core is always followed by same character :
follower  a = &anchor ; &anchor = 0
    next = null
    p = core len(1) . ch rem . rest
    input(.maintext,unit)
**  caller must save position on file unit.
    rewind(unit) ; k = 0
**  read next line :
follow1  line = maintext                          :f(follows)
follow2  line p                                   :f(follow1)
    k = lt(k,39) k + 1                            :f(follows)
    line = ch rest
    differ(next)                                  :s(follow4)
**  gets here first time only :
    next = ch                                     :(follow2)
**
follow4  ident(next,ch)                           :s(follow2)
**  successors differ, no need to go on:
    &anchor = a ; follower = null                 :(return)
follows  &anchor = a
    follower = next                               :(return)
**  Returns successor char if consistent, else null.

teff_end
```

```
**  PS.INC : routine(s) for Chisubs & allies :
**  Copyright (C) Richard Forsyth, 1995.
**
**  First version : 26-Mar-95
**  Last revision : 26-Jun-95
**

**  Function declarations :
    define('postsort(a,n)j,jump,k,t1,t2')
    define('fabs(v)x')
                                                  :(ps_ends)

**  Function definitions :



**  FABS :
**  Absolute value function, ignoring sign :
```

```
fabs  ge(v,0)                                    :s(fab1)f(fab2)
fab1  fabs = v                                   :(return)
fab2  fabs = -v                                  :(return)


**  POSTSORT :
**  Puts equal a[,2] items in decreasing order of size(a[,1]) :
postsort  j = 1;  jump = 'ps_1'
**  forward pass :
ps_1  j = lt(j,n) j + 1                          :f(ps_3)
    gt(fabs(a[j - 1,2] - a[j,2]),1e-8)           :s(ps_1)
**  more or less equal :
    ge(size(a[j - 1,1]),size(a[j,1]))            :s(ps_1)
**  swap if same value & ascending in size :
ps_2  t1 = a[j - 1,1] ; t2 = a[j - 1,2]
    a[j - 1,1] = a[j,1] ; a[j - 1,2] = a[j,2]
    a[j,1] = t1 ; a[j,2] = t2
    k = k + 1                                    :($jump)
**  backward pass :
ps_3  jump = 'ps_4'  ; k = 0
ps_4  j = gt(j,2) j - 1                          :f(ps_9)
    gt(fabs(a[j - 1,2] - a[j,2]),1e-8)           :s(ps_4)
    ge(size(a[j - 1,1]),size(a[j,1]))            :s(ps_4)f(ps_2)

ps_9  postsort = k                               :(return)

ps_ends
```

Appendix B

Specimen Spitbol Program: Basic Bayesian Text Classifier

As the Basic Bayesian Text Classifier emerged from the benchmarking exercises of chapters 4-8 as overall `winner', despite its simplicity, there follows for reference a listing of the program BBTC, written in the Spitbol dialect of Snobol4.

```
**  BBTC -- Basic Bayesian Text Classifier :
**  Copyright (c) Richard Forsyth, 1994.
*
**  First version : 15-Sep-94
**  Last revision : 16-Jul-95
*

**  Global settings & patterns :
    &trim = 1 ; &anchor = 1
**  &trim & &anchor better as 0 ??
    alphanum = "1234567890" &ucase &lcase "_"
    alphapat = break(alphanum) span(alphanum) . word
    qq = '"' "'" ; null = ''
    nongram = 'Zz$$'
    blocsize = 20 ; tiny = 1.2e-23
    probmin = 1.0 / 144 ;** provides floor on prob. estimates.

**  Function declarations :
    define('baseline(c,filename)card,ch,e,j,t,x')
    define('results(c,k,cats,filename)j,lout,p2,outline,x')
    define('ent1(charray,t)e,j,r')
    define('prob(x,c)f,p')
    define('vadd(v,n,tv)j')
    define('nodeconv(s)ch,f,j,n,m,t')
    define('readdir(vecsize)dire,card,fn,extn')
    define('usebase(c,k,cats,card)j,p,x')
    define('logprobs(e,cats,n,pvec)j,sp')
    define('scaling(pvec,n)j,sp')
    define('precall(cm,cats)i,j,p,x')


-include "sno4.inc"

**  Prelude :

    terminal = 'BBTC, version 1.4 -- ' date()
    terminal = 'Please give training directory filename (.v1) :'
    filename = input
    input(.dirline,1,filename)                :f(dudmain)
    mainfile = filename
    terminal = 'Please give testing directory file-name (.v2) :'
    testfile = input
    input(.testline,2,testfile)                :f(dudtest)
    terminal = 'Please give output filename (.out) :'
```

```
    outfile = input
    output(.output,10,outfile)                  :f(dudfout)
    output = 'BBTC output;  date: ' date()
    terminal = 'Append summary to log-file (Y=yes) ?'
    logdump = replace(input,&ucase,&lcase)
    logdump span(' ') =
preloop  terminal = 'Please give N-gram size (1..4) :'
    gramsize = input
    (integer(gramsize) gt(gramsize,0) le(gramsize,4))  :f(preloop)
    gramsize = +gramsize                        :(main)

dudmain  terminal = 'Cannot read file ' filename ' !' :(end)
dudtest  terminal = 'Cannot open ' testfile ' !'  :(end)
dudfout  terminal = 'Cannot open ' outfile ' !'  :(end)


**  Main Program :
main  &dump = 2 ; t1 = time()
    output = 'gramsize = ' gramsize
    dots = dupl('.',gramsize)
    readdir(60) ; terminal = nf ' training files.'
    training = copy(filelist)
    cats = nf
    endfile(1) ; endfile(2)
    input(.dirline,2,testfile)
    readdir(60) ; terminal = nf ' test files.'
**  Compute base frequencies :
    c = 0 ; basefreq = table(201)
    totchars = array(cats) ; symbols = array(cats)
    tent = array(cats) ; minfreq = array(cats)
    confuse = array('1:' cats ',1:' cats,0)
baseloop  c = lt(c,cats) c + 1                  :f(usefreqs)
    input(.mainline,1,training[c] '[-l4096]')
    symbols[c] = baseline(c,training[c])
    endfile(1)                                  :(baseloop)

**  Do the work :
usefreqs  c = 0 ; correct = 0
    samples = 0 ; penalty = 0.0
    brierval = 0.0
    bent = array(cats) ; pvec = array(cats)
mainloop  c = lt(c,nf) c + 1                    :f(endprog)
    endfile(2) ; k = 0
    input(.mainline,2,filelist[c] '[-l4096]')
    output = ; output = terminal = c '  ' filelist[c]
    ok = 0
lineloop  card = mainline                       :f(endfile)
    k = k + 1
    usebase(c,k,cats,card)
    b = results(c,k,cats,filelist[c])
    brierval = brierval + b
    penalty = penalty + ln(pvec[c])
    vadd(tent,cats,bent)
**  entropy added after attenuation.
**  (eq(remdr(k,blocsize),0) showbloc(c,k,cats,bent))
                                                :(lineloop)
endfile  output = ok ' / ' k ' = ' ok / (k + 0.0)
    samples = samples + k ; correct = correct + ok
```

```
**  (ne(remdr(k,blocsize),0) showbloc(c,k,cats,bent))
                                                    :(mainloop)
**  showbloc zeroes bent[] before returning.

endprog  &dump = 0
    output =
    output = 'Percentage correct = ' 100 * correct / (samples + tiny)
    meanplog = (penalty / ln(2.0)) / (samples + tiny)
    output = 'Mean log-penalty = ' meanplog
    output = 'Mean Brier score = ' brierval / samples
    output =
**  also dump confusion matrix :
    i = 0
conloop  i = lt(i,cats) i + 1                  :f(lend)
    lout = rpad(i,7) ; j = 0
conloop1  j = lt(j,cats) j + 1                 :f(conloop7)
    lout = lout ' ' lpad(confuse[i,j],7)       :(conloop1)
conloop7  output = lout                        :(conloop)
lend  output = ; prec = precall(confuse,cats)
    output = 'Precall = ' prec * 100.0
    terminal = 'BBTC finished : ' date()
    terminal = 'Runtime = ' (time() - t1) / 1000.0 ' secs.'

    ident(substr(logdump,1,1),'y')             :f(end)
**  lastly, dump summary stats:
    output(.logline,19,'bbmm.log[-a]')         :s(logdump)
    terminal = 'Cannot open logfile !'         :(end)
logdump  v1 = correct / (samples + tiny)
    logline = 'BBTC [sqrt] on ' mainfile '  ' date()
    logline = cats ' ' gramsize ' 0 0 2  ' v1 '  ' meanplog '  ' prec '  0.0
0.0'
    terminal = logline
                                               :(end)

**  Function definitions :


**  BASELINE :
**  Computes basic character frequencies :
**  mainline, symbols, basefreq<> & charray[,] are global.
baseline  symbols[c] = 0 ; t = 0
    terminal = 'Compiling ' gramsize '-gram frequencies.'
baselin1  card = mainline                      :f(baselin7)
    j = 1
baselin2  ch = substr(card,j,gramsize)         :f(baselin1)
    j = j + 1 ; t = t + 1
    differ(basefreq<ch>)                       :s(baselin4)
**  new item :
    symbols[c] = symbols[c] + 1
    basefreq<ch> = array(cats,0)
baselin4  basefreq<ch>[c] = basefreq<ch>[c] + 1      :(baselin2)
baselin7  output = ; output = 'Source file : ' filename
    output = 'No. of symbols = ' t
    output = 'No. of different symbols = ' symbols[c]
    totchars[c] = t
    baseline = symbols[c]                    :(return)
```

```
**  ENT1 :
**  Works out first-order entropy :
ent1  e = 0.0 ; t = t + 0.0
ent2  j = j + 1
    r = charray[j,2] / t                        :f(entx)
    e = e - ln(r) * charray[j,2]                :(ent2)
entx  e = (e / t) / ln(2.0)
    ent1 = e                                    :(return)


**  PROB :
**  Probability of x in category c :
prob  f = (ident(basefreq<x>) 0.618034, basefreq<x>[c])
    f = lt(f,1) 0.618034
    p = f / (totchars[c] + 0.618034)
    prob = p                                    :(return)


**  VADD :
**  Adds v[] to tv[] :
vadd  j = lt(j,n) j + 1                         :f(return)
    tv[j] = tv[j] + v[j]                        :(vadd)


**  LOGPROBS :
**  Converts log2-entropies to probabilities :
logprobs  sp = 0.0
logprob1  j = lt(j,cats) j + 1                  :f(logprob2)
    e[j] = (e[j] * ln(2.0)) / n
    pvec[j] = exp(-e[j])
    sp = sp + pvec[j]                           :(logprob1)
**  e[j] = sqrt(e[j] * ln(2.0)) very conservative, taken out after logprob1.
logprob2  j = 0
**  normalize :
logprob4  j = lt(j,cats) j + 1                  :f(return)
    pvec[j] = pvec[j] / sp                      :(logprob4)
**  USEPROBS uses max.entropy, thus logprobs uses -e[j].
**  Also scales by n to avoid over-extremity.


**  SCALING :
**  Re-scales probs to avoid near-zero estimates.
scaling  sp = 1.0 + n * probmin
scaling1  j = lt(j,n) j + 1                     :f(return)
    pvec[j] = (pvec[j] + probmin) / sp          :(scaling1)
**  probmin is global : minimum reasonable probability.


**  USEBASE :
**  Uses base frequencies in simple-minded Bayesian way :
usebase  j = 0
    ment = 9.9E99 ; mentcat = 1
usebase1  j = lt(j,cats) j + 1                  :f(return)
    tent[j] = 0.0 ; guesses = 0
usebase2  x = substr(card,guesses + 1,gramsize)  :f(usebase9)
    guesses = guesses + 1
    p = prob(x,j)
    tent[j] = tent[j] - ln(p)                   :(usebase2)
```

```
usebase9  tent[j] = tent[j] / ln(2.0)
    ge(tent[j],ment)                          :s(usebase1)
    ment = tent[j] ; mentcat = j              :(usebase1)
**  ment is minimum entropy.


**  RESULTS :
**  Shows results :
results  outline = lout =
    logprobs(tent,cats,sqrt(1 + guesses),pvec)
    scaling(pvec,cats) ; p2 = 0.0
results1  j = lt(j,cats) j + 1                :f(results9)
    p2 = p2 + pvec[j] ^ 2
    x = chop(pvec[j] * 10000 + 0.49) / 10000.0
    lout = lout lpad(x,9) ' '
    outline = outline lpad(tent[j],11) ' '    :(results1)
results9  output = k '  ' lout '  ' mentcat '  ' c '  ' p2
    confuse[mentcat,c] = confuse[mentcat,c] + 1
    ok = eq(mentcat,c) ok + 1
    results = 2.0 * pvec[c] - p2              :(return)
**  returns Brier score.


**  PRECALL :
**  Computes average precision*recall :
precall  rsum = array(cats,0.0)
    csum = array(cats,0.0)
precall1  i = lt(i,cats) i + 1                :f(precall4)
    j = 0
precall2  j = lt(j,cats) j + 1                :f(precall1)
    x = cm[i,j] + tiny
    rsum[i] = rsum[i] + x ; csum[j] = csum[j] + x  :(precall2)
precall4  p = 0.0 ; j = 0
precall7  j = lt(j,cats) j + 1                :f(precall9)
    x = cm[j,j] + tiny
    p = p + sqrt((x * x) / (rsum[j] * csum[j]))  :(precall7)
**  goes along main diagonal.
precall9  precall = p / cats                  :(return)
**  arith.mean of geo.mean of precision & recall, by categories.


**  READDIR (reads directory of file names):
**  nf, direline & filelist[] are global.
readdir  dire = 'C:'
**  default drive is hard-disc C:
    filelist = array(vecsize)
    nf = 0
readdir1  card = dirline                      :f(return)
    card ' Directory of ' rem . dire          :s(readdir1)
    card ' Directory' break(alphanum) rem . dire  :s(readdir1)
    card span(alphanum) . fn span(' ') span(alphanum) . extn :f(readdir1)
**  Append next file name :
    nf = nf + 1
    filelist[nf] = dire '\' fn '.' extn       :f(return)
                                              :(readdir1)

END
```

Appendix C

Specimen C Code: GA Routines and Heap-Tree Evaluation Routines

Over 50 C programs were written for this thesis. As a sample, two of the more innovative pieces of C software arising from this project have been listed below: the genetic-algorithm function-library used by IOGA, MAWS and GLADRAGS; and the heap-tree rule-evaluation routines used by the GLADRAGS system.

Tests on a problem analyzed by Jennison & Sheehan (1993) suggest that the evolutionary search procedure implemented in GA.inc is more efficient than most genetic algorithms.

```
/*
  GA.inc :
  Semi-Generic Genetic Algorithm Library:
  Copyright (c) 1994, 1995,  Richard S. Forsyth.
  First version : 04-Oct-94
  Last revision : 08-Jun-95
*/

/* fitness function must be provided elsewhere */

/* expects caller to have
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <direct.h>
#include <math.h>
#include <float.h>
#include <time.h>
#include <malloc.h>
#include <io.h>

#include <glib.h>
#include <statlib.h>

#include "glob.h"
#include "util.c"

before including this file */


/* constants */
#define  TINY  FLT_EPSILON
#define  HUGE  FLT_MAX
#define  POPSIZE  49
```

```
#define  MAXTRIES  4096
#define  MAXFLAT  3600
#define  BOREDOM  200
#define  NPROBES  4
#define  POSITIVE  1
#define  NEGATIVE  0


/* globals */

int  maxgene=256;
  /* 1 more than highest allowable gene value */

#define  Randbyte  ((unsigned char) iran(maxgene))


/*  data structures */

Byte  *genepool[POPSIZE];
double  goodness[POPSIZE];
  /* eval() & o/p functions impose structure */

int  glen;  /* length of genestring */
int  elements;
  /* no. of patterns per genestring */
int  whengen=POPSIZE;
  /* record of when best-ever gene-string was created */


/* prototypes */

unsigned iran (int topmost);
int initpop (int glen, int farg);
double ga (int glen, int elements, int maxloops, int farg);
void copygene (int from, int n, int unto);
int findpoor (double av, unsigned n);
int findgood (double av, unsigned n);
void mate (int p1, int p2, unsigned n, int x);
int mutator (int popsize, unsigned glen);
void enforce (int k, int glen);
double fitness (int g, int farg);
  /* must be defined elsewhere */
void swapsegs (int g, unsigned p, unsigned q);

void gout (FILE *fp, int g, int glen, int vars);
  /* defined in rule-eval file (e.g. TOGRULE.inc), declared here as well */


/* SUBPROGRAMS */



unsigned iran (int n) {
/******************/
  /* pseudo-random integer between 0 & n-1 */

  return (rand() % n);
  }
```

```
int initpop (int glen, int farg) {
/*****************************/
  /* makes & scores initial genepool */
  int  b, bestpos, j, k;
  double  best, f;

  if (genepool[0] == NULL)  /* shouldn't really be needed */
    genepool[0] = malloc(glen+1);
  /* item zero is used as archive for best-ever */

  best = -HUGE;
  for (j=1; j<POPSIZE; j++) {
    if (genepool[j] == NULL)
      genepool[j] = malloc(glen+1);  /* extra byte for safety */
    if (genepool[j] == NULL)  printf("Memory overflow in initpop() !!\n");
    for (k=0; k<glen; k++)
      genepool[j][k] = Randbyte;
      /* character string */
    enforce(j,glen);  /* keep it legal */
    f = fitness(j,farg);
    if (f > best) {  /* best is maximum */
      best = f;  bestpos = j;
#ifdef Savetab
      Savetab;
#endif
      /* allows caller to keep related info if a Savetab macro is defined */
      }
    goodness[j] = f;
    printf(".");  /* visual evidence of work going on */
    }
  printf("\n");
  return  bestpos;
  }


double ga (int glen, int elements, int m, int farg) {
/***********************************************/
  /* Forsyth's home-made genetic algorithm */
  double  av, best, f, rn;
  int  bestpos,g,j,k;
  int  p1, p2;
  unsigned  nextfill, plateau;

  av = 0.0;
  rn = POPSIZE - 1.0;  /* real number of chromosomes */
  /* create initial population */
  bestpos = initpop(glen,farg);
  best = goodness[bestpos];
  /* keep a copy of best in location zero */
  printf("bestpos = %d  glen = %d  in ga \n", bestpos,glen);
  copygene(bestpos,glen,0);  goodness[0] = best;

  /* also compute mean score */
  for (j=1; j<POPSIZE; j++)  av += goodness[j];
  av = av / rn;
  printf("Mean fitness score = %8.4f\n", av);
```

```
  printf("Best fitness score = %8.4f\n", best);
  plateau = 0;  nextfill = 100;  whengen = bestpos;

  /* main loop */
  for (g=POPSIZE-1; g<m && plateau<MAXFLAT; g++) {
      /* m is maximum tries to make */
    p1 = findgood(av,POPSIZE);
    p2 = iran(POPSIZE);  /* 2 parents selected */
    k = findpoor(av,POPSIZE);  /* place of offspring */
    mate(p1,p2,glen,k);  /* crossover */
    enforce(k,glen);
    f = fitness(k,farg);
    if (f > best) {  /* keep it */
      copygene(k,glen,0);  goodness[0] = f;
#ifdef Savetab
      Savetab;
#endif
      plateau = 0;  best = f;
      whengen = g;
      printf("Best @ %d = %10.4f\n", g,goodness[0]);
      gout(stdout,k,glen,vars);
      /* show user best ever with gout */
      }
    else  /* no improvement */
      plateau++;
    av = av + (f/rn) - goodness[k]/rn;
    goodness[k] = f;
    if (plateau > BOREDOM && iran(3) == 1) {
      /* sometimes does mutation outside crossover, when stuck */
      k = mutator(POPSIZE,glen);
      enforce(k,glen);
      /* also must revise mean fitness */
      f = fitness(k,farg);
      av = av + f/rn - (goodness[k]/rn);
      goodness[k] = f;  g++;  plateau++;
      if (f > best) {  /* just might be best ever */
        copygene(k,glen,0);  goodness[0] = f;
#ifdef Savetab
        Savetab;
#endif
        best = f;  plateau = 0;
        whengen = g;
        printf("best @ %d = %10.4f\n", g,best);
        gout(stdout,k,glen,vars);  /* show user */
        }
      }
    if (g >= nextfill) {
      printf("After %d trials; mean = %11.5f, best = %10.5f\n", g,av,best);
      nextfill += 100;
      }
    }
  printf("No. of trials = %d\n", g);
  return best;
  }


void copygene (int from, int n, int unto) {
/*************************************/
```

```
  /* chromosome copy routine */
  int  k;  /* genepool[][] is global */

  for (k=0; k<n; k++)
    genepool[unto][k] = genepool[from][k];
  }


int findpoor (double av, unsigned n) {
/*********************************/
  /* finds a poor genestring */
  int  j, r, w;
  double  poor;

  poor = HUGE;
  for (j=0; j<NPROBES; j++) {
    r = iran(n-1) + 1;
    if (goodness[r] < poor) {  /* bigger is better */
      poor = goodness[r];
      w = r;
      }
    }
  return w;
  }


int findgood (double av, unsigned n) {
/*********************************/
  /* locates a good genestring */
  int  b, j, r;
  double  good;

  good = -HUGE;
  for (j=0; j<NPROBES; j++) {
    r = iran(n-1) + 1;
    if (goodness[r] > good) {  /* most is best */
      good = goodness[r];
      b = r;
      }
    }
  return  b;
  }



void mate (int a, int b, unsigned n, int x) {
/*************************************/
  /* uniform crossover routine */
  int  j;  /* genepool[][] is global */

  for (j=0; j<n; j++) {
    if (iran(2))
      genepool[x][j] = genepool[a][j];
    else
      genepool[x][j] = genepool[b][j];
    /* mutation possible here also (at present) */
    if (iran(100) > 96)  genepool[x][j] = Randbyte;
    }
```

```
  /* better if biased towards switch at syntactic boundaries ? */
  }


int mutator (int popsize, unsigned glen) {
/*************************************/
  /* mutation routine for genestrings */
  int  k, loop, t;
  unsigned  p, q, r;
    /* genepool[][] is global, also subtext[], totbytes, elements & patsize
*/

  k = iran(popsize-1) + 1;
    /* won't alter item zero */
  for (loop=0; loop<2; loop++) {
    switch (iran(7)) {
      case 0:  case 1:
        p = iran(glen);
        genepool[k][p] = Randbyte;
        break;
      case 2:
        p = iran(glen);
        q = iran(glen);
        swapsegs(k,p,q);
        break;
      case 3:
        p = iran(glen-1);
        swapsegs(k,p,p+1);
        break;
        /* swapsegs used to respect segment boundaries */
      case 4:  /* swap successive atoms */
        p = iran(glen-1);
        t = genepool[k][p];
        genepool[k][p] = genepool[k][p+1];
        genepool[k][p+1] = t;
        break;
      default:  /* small increment/decrement */
        p = iran(glen);
        t = (int) genepool[k][p] + iran(5) - 2;
        if (t < 0)  t = Randbyte;
        genepool[k][p] = (Byte) t;
        break;
      }
    }
  /* caller to tidy whole genestring after this */
  return k;  /* lets caller know what genestring was mutated */
  }


void enforce (int k, int glen) {
/***************************/
  /* enforces rules on genestring */
  int  j;

  for (j=0; j<glen; j++) {
    genepool[k][j] = (genepool[k][j]) % maxgene;
    }
  }
```

```
void swapsegs (int k, unsigned p, unsigned q) {
/*****************************************/
  /* swaps chunks of a genestring k in genepool[] */
  Byte  ch;

  if (k<=0 || k>POPSIZE)  printf("Swapsegs k !!\n");
  if (p<0 || p>glen)  printf("Swapsegs p %d\n", p);
  if (q<0 || q>glen)  printf("Swapsegs q %u\n", q);
    /* checking */
  ch = genepool[k][p];
  genepool[k][p] = genepool[k][q];
  genepool[k][q] = ch;
  return;
  }
```

```
/*
  TOGRULE.inc :
  Rule Evaluation Routines for Text Oriented Genetic Algorithm:
  designed to find classification rules for strings;
  Copyright (c) 1994, 1995  Richard S. Forsyth.
  First version : 04-Oct-94
  Last revision : 25-May-95
*/

/* expects enclosing program to have :

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <direct.h>
#include <math.h>
#include <float.h>
#include <time.h>
#include <malloc.h>
#include <io.h>

#include <glib.h>
#include <statlib.h>

#include "glob.h"
#include "util.c"

  before including this file */

/* constants */
#define  TINY  FLT_EPSILON
#define  HUGE  FLT_MAX
#define  NOPS  4
#define  MATHOPS  5
#define  MONADICS  7

/* macros */
#define  Shop(C) (mathops[C % MATHOPS])
```

```
  /* Show dyadic math operator in character form */

#define  Atom(P)  ((P+2)*2 >= glen)
#define  Lsub(P)  ((P+1)*2)
#define  Rsub(P)  ((P+1)*2 + 2)
  /* above macros treat gene-string as heap-tree */

/* data types Byte & Subtype expected */

/* globals */

int  evalmode=0;
int  fun1=0, fun2=0;
  /* fun1 & fun2 control usage of monadic operators (functions) */

char *opstring = "&|><";
char *mathops = "+-*^\\";

char *funcname[] = {
  " ", "Fabs ", "Root ", " ", "Slog ", "Tanh ", " ",
  };
/* operator symbols */


/* prototypes */
void gout (FILE *fp, int g, int glen, int vars);
void ruleout (FILE *fp, int g, int p, int glen, int vars);
int ruleval(int g, unsigned s, int vars, int glen);
int testrel (int op, double x, double y);
double math (int op, double x, double y);
double leibniz (int op, double v);
double slog (double v);
void ruledump (FILE *fp, int g, int glen, int cats);


/* SUBPROGRAMS */


void gout (FILE *fp, int g, int glen, int vars) {
/*****************************************/
  /* prints genestring in (semi-)intelligible form */
  unsigned  n;

  fprintf(fp,"Glen = %d.\n", glen);
  ruleout(fp,g,0,glen,vars);
  fprintf(fp,"\n");
  }


void ruleout (FILE *fp, int g, int p, int glen, int vars) {
/*************************************************/
  /* prints a single rule from genepool */
  int lh, rh;

  lh = genepool[g][p];
  rh = genepool[g][p+1];
  if (Atom(p)) { /* atom */
    if (fun1)
```

```
      fprintf(fp, "%s", funcname[lh % MONADICS]);  /* unary op */
    if (rh < vars)
      fprintf(fp,"`%s`", subs[rh].subtext);
    else  fprintf(fp, "%d", rh-vars);
    }
  else {  /* subexpression */
    if (p>0 && fun2>0)
      fprintf(fp, "%s", funcname[lh % MONADICS]);
      /* root does not have monadic op */
    if (p>0)  fprintf(fp,"(");
    ruleout(fp,g,Lsub(p),glen,vars);
    if (p==0) {  /* main operator always GT */
      if (glen > 17 || fun1 && fun2)
        fprintf(fp, "\n ");
      fprintf(fp, " > ");
      }
    else
      fprintf(fp, " %c ", Shop(rh));
    ruleout(fp,g,Rsub(p),glen,vars);
    if (p>0)  fprintf(fp,")");
    }
  }


int testrel (int op, double x, double y) {
/************************************/
  /* does a logical/relational operation */

  switch (op % NOPS) {
    case 0:
      return (x > 0.0) && (y > 0.0);  /* and */
    case 1:
      return (x > 0.0) || (y > 0.0);  /* or */
    case 2:
      return (x > y);  /* gt */
    case 3:
      return (x < y);  /* lt */
    default:
      return  (x > 0.0) && (y > 0.0);  /* default is && */
    }
  }


double math (int op, double x, double y) {
/************************************/
  /* applies dyadic arithmetic operator */

  switch (op % MATHOPS) {
    case 0:
      return (x + y);
    case 1:
      return (x - y);
    case 2:
      return (x * y);
    case 3:  /* maximum operator */
      if (x > y)  return  x;
      else  return (y);
    case 4:  /* minimum */
```

```
      return (x < y ? x : y);
    default:
      return  (x + y);
    }
  return  0.0;  /* should never fall here */
  }


int ruleval (int g, unsigned s, int vars, int glen) {
/*********************************************/
  /* applies pattern rule g to line s, current line */
  int  h, j, lh, rh;
  double  x, y;
  static double  v[64];  /* wastes space to save time */

  h = glen / 2 - 2;
  for (j=h*2; j>=h; j -= 2) {  /* atoms */
    lh = genepool[g][j];  rh = genepool[g][j+1];
    if (rh < vars)  /* variable */
      x = (double) subs[rh].freq[(long)s];
    else /* constant */
      x = (double) (rh - vars);
    if (fun1)  x = leibniz(lh,x);
    v[j] = x;
    }
  for (j=h-2; j>0; j -= 2) {  /* arithmetic subexpressions */
    lh = genepool[g][j];
    rh = genepool[g][j+1];
    x = math(rh,v[Lsub(j)],v[Rsub(j)]);
    if (fun2)  x = leibniz(lh,x);
    v[j] = x;
    }
  return  (v[2] > v[4]);  /* main op always greater than */
  }


double leibniz (int op, double v) {
/*****************************/
  /* applies a monadic operator */

  switch (op % MONADICS) {
    case 0:  case 3:  case 6:
      return  v;
    case 1:
      return  fabs(v);
    case 2:  /* safe root */
      if (v >= 0.0)  return sqrt(v);
      else  return -sqrt(-v);
    case 4:  /* safe logarithm */
      return  slog(v);
    case 5:
      return  tanh(v);
    default:
      return  v;  /* shouldn't happen */
    }
  return  0.0;  /* should never happen */
  }
```

```
double slog (double v) {
/********************/
  /* safe natural logarithm */
  double  x;

  x = log(fabs(v) + 1.0);  return  ((v >= 0.0) ? x : -x);
  }


void ruledump (FILE *fp, int g, int glen, int cats) {
/************************************************/
  /* dumps genestring in form easy to read by next program */
  int  h,i,j,s;

  h = glen / 2 - 2;
  fprintf(fp, "%d \n%d \n",
    genepool[g][0],genepool[g][1]);
    /* first 2 items don't change */

  for (j=2; j<h; j++) {
    i = genepool[g][j];
    fprintf(fp, "%3d  ", i);
    if (j % 2 == 0) {  /* monadic op */
      if (fun2)  fprintf(fp, " %s", funcname[i % MONADICS]);
      }
    else {  /* dyadic op */
      fprintf(fp, " %c", Shop(i));
      }
    fprintf(fp, "\n");
    }
  for (j=h; j<glen-1; j++) {  /* atoms */
    i = genepool[g][j];  fprintf(fp, "%3d  ", i);
    if (j % 2 == 0) {
      if (fun1)  fprintf(fp, " %s\n", funcname[i % MONADICS]);
      else  fprintf(fp, "\n");
      }
    else {
      if (i < vars)  fprintf(fp, "`%s`\n", subs[i].subtext);
      else  fprintf(fp, "%d\n", i-vars);
      }
    }
  fprintf(fp, "%3d \n", genepool[g][glen-1]);
  }
```

Appendix D

Details concerning Benchmark Data Sets

The 13 text-classification problems that constitute TBS1 (Text Benchmark Suite, Mark 1) form an essential backbone to the work of this project. They also constitute, in embryonic form, a valuable resource for future studies in text analysis. Collecting and editing TBS1 was a reasonably arduous chore, but enhancing and maintaining it could become a full-time job. Already some of the problems of corpus management have presented themselves. It is hoped that support to overcome such problems may in due course be forthcoming, so that successors to TBS1 may offer a genuine resource to the global research community. Meanwhile, this Appendix details the contents of TBS1 as it stood on 11 August 1994. (Some extensions to the `electronic anthology' from which TBS1 was drawn have been made since that date, but, to preserve comparability among the tests of Chapters 4-8 in this thesis, none of this additional material has yet been incorporated into the benchmark suite itself.)

Information about the selection of works from various authors and subdivision into training and test files is contained in the body of Chapter 4 (section 4.2). Here this is amplified by giving further details concerning the sources and sizes of the texts used in the benchmark suite of 13 textual discrimination problems (TBS1).

NOTE: A policy adhered to throughout was never to split a single work (article, essay, poem or program) between training and test sets -- except in the case of *The Waves* by Virginia Woolf (problem WAVE), where the object of the exercise was to classify portions of a single novel according to the `speaker'.

D.1 Sources & Subdivisions of Benchmark Data

Authorship / English Poetry

LIT1(4 classes): Poetry by Bob Dylan, Dylan Thomas, John Pudney and Richard Forsyth.

Songs by Bob Dylan (born Robert A. Zimmerman) were obtained from *Lyrics 1962-1985* (Dylan, 1994). In addition, two tracks from the album *Knocked Out Loaded* (Dylan, 1988) and the whole A-side of *Oh Mercy* (Dylan, 1989) were transcribed by hand and included, to give fuller coverage. An electronic version of *Lyrics 1962-1985* is apparently available from the Oxford Text Archive, but I was not aware of this until after this selection had been compiled. Further information about the Oxford Text Archive can be obtained by sending an electronic mail message to

ARCHIVE@vax.oxford.ac.uk

Poems of Dylan Thomas were obtained from *Collected Poems 1934-1952* (Thomas, 1952) with four more early works added from *Dylan Thomas: the Notebook Poems 1930-1934* (Maud, 1989).

Poems by John Pudney were: a handful from *Selected Poems* (Pudney, 1946), plus most of *Spill Out* (Pudney, 1967) and most of *Spandrels* (Pudney, 1969).

Poems by Richard Forsyth have not been previously published in book form.

LIT2(3 classes): Poems by Helen Forsyth, James Forsyth and Richard Forsyth, i.e. my mother, my father and myself.

Poems by Helen Forsyth were from *Enamelled in Fire* (Forsyth, 1994). Poems by James Forsyth came from *On Such a Day as This* (Forsyth J.L., 1989) and *From Time to Time* (Forsyth J.L., 1990). R.S. Forsyth's selection was as in LIT1 but with a different division into test and training files.

LIT3(3 classes): Poems by Ezra Pound, T.S. Eliot and William B. Yeats.

Poems by Ezra Pound came from *Selected Poems 1908-1969* (Pound, 1977).

Poems by T.S. Eliot were from *Collected Poems 1909-1962* (Eliot, 1963). *The Waste Land*, which was written by T.S. Eliot but heavily amended on the advice of Ezra Pound (see Valerie Eliot (1971)) was **excluded** from this test.

Poems by W.B. Yeats were from *Collected Poems* (Yeats, 1961).

As is usual in machine learning the data was divided into training and testing sets. This division was made by allocating **files** randomly to test or training sets until at least 30% of the available corpus of each author had been assigned to the test set, any remainder going in the training set. As this data is held (with a few exceptions) in files each of which contains writings composed by one author in a single year, this mode of division meant that works composed at about the same time were usually kept together; and, even more important, that single poems were never split between test and training files. The effect of this file-based blocking is presumably to make these tests somewhat more stringent than random allocation of individual poems to test and training sets would have been.

Authorship / English Prose

FEDS(3 classes): a selection of **undisputed** papers by the three *Federalist* authors Hamilton, Jay and Madison.

An electronic text of the entire Federalist papers was obtained by anonymous ftp from Project Gutenberg at
MRCNEXT@cso.uiuc.edu

For checking purposes the Mentor edition was used (Hamilton et al., 1961).

Here the division into test and training sets was as follows.

Author	Training paper nos.	Test paper nos.
Hamilton	1, 21-29, 60, 70	7, 11, 15-17, 30, 32, 33, 35, 36, 69, 80
Jay	2-5	64
Madison	40-48	10, 14, 37-39

Authorship / Snobol4 Programming Language

SNOB(4 classes): Samples of program source code written by four different programmers in the Snobol4 programming language. These were Richard Forsyth, James Gimpel, Ralph Griswold and Mike Shafto.

My own programs were mostly written in 1994, for this project, with one program from 1979 and another from 1988. The other three sets came from diskettes purchased from Catspaw Inc. of Salida, Colorado CO 81201, USA, who sell them as useful program libraries. In fact the Gimpel programs come originally from the book *Algorithms in Snobol4* (Gimpel, 1986) and those by Griswold from *String and List Processing in Snobol4* (Griswold, 1975).

This problem is the only one to fulfill the requirement that not all texts should be in the English language; indeed it is not even in natural language, although since the programs were written by English-speakers, a fair proportion of natural English is included, as comment lines.

Division into test and training sets was made by concatenating each author's files, in the order that they appeared on the distribution disk, and then dividing roughly at the half-way point, always at a program or function ending.

Chronology / English Poetry

AGE1(2 classes): Songs by Bob Dylan, written before and after his motorcycle crash of July 1966. Same source as LIT1.

AGE2(2 classes): Poems by Emily Dickinson, early work being written up to 1863 and later work being written after 1863. Emily Dickinson had a great surge of poetic composition in 1862 and a lesser peak in 1864, after which her output tailed off gradually.

The work included was all of *A Choice of Emily Dickinson's Verse* selected by Ted Hughes (Hughes, 1993) as well as a random selection of 16 other poems from the *Complete Poems* (edited by T.H. Johnson, 1970).

AGE3(2 classes): Poems by James Forsyth, written up to 1945 (early work) or from 1985 onwards (later work). Same sources as LIT2.

Even without the family connection it would be interesting to have such a clear-cut example of a division into two productive periods, separated by an unusually long, 40-year, span during which the author wrote very little poetry.

AGE4(2 classes): Early and late poems of W.B. Yeats. Early work taken as written up to 1914, the start of the First World War, and later work being written in or after 1916, the date of the Irish Easter Rising, which had a profound effect on Yeats's beliefs about what poetry should aim to achieve. Same source as LIT1.

For these four problems the classification objective was to discriminate between early and late works by the same poet. The division into test and training sets was once again file-based: in these cases the files of each poet were ordered chronologically and assigned alternately (i.e. from odd then even positions in the sequence) to two sets. The larger of the two resulting files was designated as training and the smaller as test file.

Content / Technical Prose

MAGS (2 classes): This used articles from the two journals *Literary and Linguistic Computing* and *Machine Learning*. The task was to classify texts according to which journal they came from. (See subsection 3.3.3.)

Here the training set for *Literary and Linguistic Computing* consisted of the years 1990 and 1991, with the test set being the articles from 1992, 1993 and 1994. For *Machine Learning* the training set comprised the two largest files, namely those of 1990 and 1992; the test set was from years 1987, 1988, 1989, 1991, 1993 and 1994. (The files for years 1987-89 each contained only a single article.)

Persona / English Prose

WAVE (3 classes): This problem was taken from the novel *The Waves* by Virginia Woolf (1931, 1992), the text of which was obtained on diskette from Alison Truelove of Royal Holloway College in a form that enabled convenient extraction of passages by individual speakers, as well as the narrator. (The original source of this text is the Oxford Text Archive, but without speaker-identification tags[1].) This book consists of monologues or soliloquies by six different characters, linked by descriptive passages. The classification task was to distinguish speech by two of the characters in the novel, called `Jinny' and `Rhoda', from narrative text by the author herself (`Virginia'). This might be considered a problem of distinguishing between the genres of narrative and dialogue, but, while it is in part a genre problem, differentiating betweem two fictional character's speech is something else. In any case the book is so peculiar that this problem is probably better characterized under some other label, as done here.

[1] All such tags, including and especially speaker-identifying tags, were of course removed from the texts before inclusion in TBS1. In addition, the forms "said Jinny" and "said Rhoda", which were invariably used in the first sentence of a passage by each of the speakers concerned, were altered to "said x".

Test/training sets for WAVE were made (for each of the three categories) by dividing at the first paragraph break after 400 lines by the speaker or narrator concerned and putting the text up to that point into the training file and the rest into the test file.

Syntax / Artificial Data

ART1(2 classes): Output from two different `poetry generation' programs, one re-written in C by myself from an earlier version in Basic (Forsyth, 1978) and another written in Snobol4 by James Gimpel (Gimpel, 1986). See Chapter 4 (section 4.2) for sample output from these generators.

ART2(2 classes): Output from two different versions of my random `poetry' generator: version 1 with the grammar as in the original (Forsyth, 1978) and version 2 with identical grammar except for the addition of relative clauses in the form `that' + Verb-Phrase allowed at the end of Noun Phrases with a 15% probability. (The use of `that', which was already in the lexicon as a determiner, to introduce a relative clause rather than `which' avoided animate/inanimate discord and also meant that the vocabularies of the two systems were **exactly the same**, unlike in ART1 -- although relative frequencies among words differed.)

Here test and training sets were obtained by running the programs to produce the required amount of text (about 6400 words) each time re-seeding the pseudo-random number generator.

D.2 Sizes of Benchmark Problems

Problem	Categories	Kilobytes (training, test)	Lines=SSCs, (training, test)
LIT1	Bob Dylan Dylan Thomas John Pudney R.S. Forsyth	83, 27 50, 17 41, 14 33, 11	131, 43 78, 27 65, 23 52, 17
LIT2	Helen Forsyth James Forsyth R.S. Forsyth	28, 9 29, 15 30, 14	44, 14 46, 23 47, 22
LIT3	Ezra Pound T.S. Eliot W.B Yeats	28, 16 32, 27 50, 16	44, 25 50, 42 78, 25
FEDS	Alexander Hamilton John Jay James Madison	154, 162 36, 13 151, 82	244, 253 58, 21 240, 129
SNOB	R.S. Forsyth James Gimpel Ralph Griswold Mike Shafto	20, 19 47, 53 33, 29 57, 59	31, 30 74, 83 51, 46 90, 92
AGE1	Bob Dylan to 1966 after 1966	37, 31 22, 20	58, 49 34, 32
AGE2	Emily Dickinson to 1863 after 1863	13, 13 7, 6	21, 20 11, 10
AGE3	James Forsyth to 1945 after 1984	15, 13 8, 6	24, 20 12, 10
AGE4	W.B. Yeats to 1914 after 1915	18, 7 32, 9	28, 11 50, 14
MAGS	Literary & Linguistic Computing Machine Learning	73, 45 81, 30	116, 70 128, 47
WAVE	Ginny Rhoda Virginia	23, 9 29, 13 20, 5	36, 14 45, 20 31, 8
ART1	Gimpel Generator Forsyth Generator	48, 48 33, 29	75, 75 53, 46
ART2	Forsyth Generator Ditto + relative clauses	33, 29 39, 34	53, 46 63, 55