Worked Examples and Solutions for the Book: Programming Collective Intelligence by Toby Segaran.

John L. Weatherwax*

December 1, 2007

Introduction

In this book Segaran has demystified the abstract ideas often associated with collective intelligence and machine learning. To make the presentation even more clear and applicable he presents the concepts in the excellent and easy to learn programming language Python.

I found this book is so enjoyable that I found myself wanting to experiment with the code and work the problems. In doing this I ended up making this document and thought it might be of interest to others. Please let me know of any errors that might exist. Enjoy!

*wax@alum.mit.edu

Chapter 2: Making Recommendations

Exercise Solutions

Exercise 1 (Tanimoto similarity score)

The Tanimoto coefficient is a modification/extension of the so called Jaccard similarity which is used to compute the *similarity* of two sets. The Jaccard similarity between sets A and B is defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}\,. \tag{1}$$

Two sets that are exactly the same have a Jaccard similarity of one, while sets that are completely different (having no overlap) have a Jaccard similarity of zero. The Tanimoto score is a modification of this idea to account for continuous valued vector variables (rather than set variables) and is defined for *vectors* A and B as

$$T(A,B) = \frac{A \cdot B}{||A||^2 + ||B||^2 - A \cdot B}\,. \tag{2}$$

Note that if $A = B$ the above gives the value of 1, while if A and B are orthogonal the above expression is zero. The Tanimoto coefficient reduces to the Jaccard coefficient in the case of binary vectors where the element zero corresponds to absence of an element in the set and element one corresponds to the presence of an element. This metric is implemented in the `recommendations.py` module under the name `sim_tanimoto`.

Chapter 3: Discovering Groups

Notes on the Text

Notes on Hierarchical Clustering

If one reads the python code that is provided with the book one notes that the book uses a somewhat strange definition of the cluster representative for a merged cluster. Effectivly the representive of the *merged* cluster is the arithmetic average of the two individual cluster represntatives of the individual clusters involved in the merge. While this is certainly a viable technique, it might not be optimal for some applications. The reason it might not be optimal can be observed when we consider that representative location will result when we combine a large clusters containing many points with a singleton cluster (a cluster that contains only a single point). Since the newly formed cluster will have a cluster representative given by the arithmetic average of the two participating clusters when merging into into a singleton set the merged cluster will have its representative moved "half way" towards the single point (no matter how many points are in the other cluster). The hierarchical clustering method described in the book is known as the Generalized Agglomerative Scheme (GAS) [2] and has several varients depending on how one choose to mesure the distance between clusters. Other methods in this same family require number of feature vectors in each cluster as well as its cluster reprentative. One could imagine clustering routines that might require the individual cluster elements as well. One very common and quite good cluster varient of the GAS family uses cluster means as the cluster reprentative and then determines the distance between clusters i and j with

$$d'_{ij} = \frac{n_i n_j}{n_i + n_j} d_{ij} \,.$$

See [2]. I modified the `cluster.py`

Exercise Solutions

Exercise 5 (returning the total distance to the cluster centers)

See the modified python function `kcluster` in `clusters.py` which now returns the resested total distance.

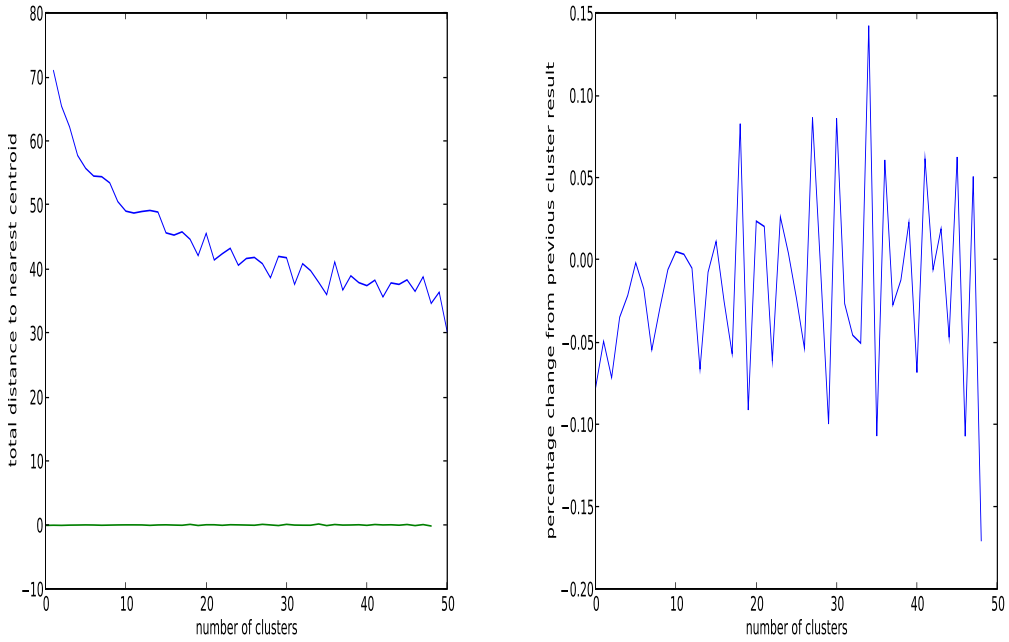

Figure 1: **Left:** The steady decrease in the total distance to each points membership cluster as we add more and more cluster centers. This is a plot of the cumulative distance from each point to the cluster center that it is in. **Right:** The *fractional* change in total distance as we consider more and more clusters.

Exercise 6 (running several k-means clustering)

See the python function `runSeveralKClusters` in `clusters.py` for an implemention of this. If we run this on the blog data set, in Figure 1 (left) we see that the total distance decreases as expected and in Figure 1 (right) we see that a plot of the fractional change

$$\frac{\text{dist}(k+1) - \text{dist}(k)}{\text{dist}(k)},$$

begins to look more and more like noise as we add clusters. The point where this plot seems to oscillate (taken backwards) would be a good place to stop clustering. From this picture it looks as if 5 clusters should be considered.

Chapter 4: (Searching and Ranking)

Book Notes

For this chapter I didn't do any of the problems but simply ran the code presented in the book. It is quite amazing to be able to develop a search engine in only a few lines of python code. The script to run much of the examples from the book is presented in the chapter directory as `run_all.py`.

Exercise Solutions

Exercise 1 (word separation)

To do this problem one would need to modify the `separatewords` function to make it separate "tokens" differently.

Chapter 5: Optimization

Book Notes

Notes on Student Dorm Optimization

For the student dorm optimization problem we can explicitly determine the number of possible student dorm assignments (or solutions we might have to search over) as follows. Assuming each dorm is an "entity", that can select two students for membership, we see that the first dorm can select two of the ten students in

$$\begin{pmatrix} 10 \\ 2 \end{pmatrix},$$

ways. Once these two students have been selected and placed that dorm the next dorm can select two students from the remaining eight students in

$$\begin{pmatrix} 8 \\ 2 \end{pmatrix},$$

ways. This process continues until all of the students are selected. The *total* number of ways we can do this selection process is then given by the product of all of these expressions. Thus the total number is given by

$$\begin{pmatrix} 10 \\ 2 \end{pmatrix}\begin{pmatrix} 8 \\ 2 \end{pmatrix}\begin{pmatrix} 6 \\ 2 \end{pmatrix}\begin{pmatrix} 4 \\ 2 \end{pmatrix}\begin{pmatrix} 2 \\ 2 \end{pmatrix} = 113400\,,$$

in agreement with the value of around 100000 suggested in the book. At this size explicit enumeration of all possible values for the cost function is possible.

Exercise Solutions

Exercise 2 (multiple annealing with random starting points)

See the python function `multiple_start_annealing` in the `optimization` module for an implementation of this routine.

Exercise 3 (quick exit if genetic optimization stalls)

See the python function `geneticoptimize` in the `optimization` module for an implementation of some exit criterion in case the computed cost function has not changed after ten iterations.

Exercise 5 (pairing students)

One might have each student rank all the other students according to their preference for each of the other person being his/her roommate. Thus each person would assign the rank of one to the person they would most likely like to room with and the rank of N (where N is the maximal number of students) to the person they would least like to room with. A solution to this problem would be a pairing of students. Given the ith student we can represent such a pairing as the function mapping $S(i)$. This mapping implies that if we have $S(i) = j$ then person j is the roommate of person i and correspondingly we must also have $S(j) = i$. To determine the cost of a particular solution $S(\cdot)$ if the ith student ranks his/her j classmate with a cost c_{ij}, then we could assume that if person i had to room with person j who was his fifth pick say he would experience a cost given by $c_{iS(j)} - 1 = 5 - 1 = 4$. Thus the total cost for the solution S might look like the sum of such individual cost functions over all students or

$$C(S) = \sum_{i=1}^{N} (c_{iS(i)} - 1) \,.$$

The subtraction of 1 is to make the cost of a persons first choice of a roommate zero.

Exercise 6 (line angle penalization)

One could use the fact that the dot product of two vectors x and y and denoted by the expression $x^T y$ equals $||x||||y|| \cos(\theta)$ where θ is the *angle* between the two vectors x and y to introduce an additional cost in the optimization objective function. Specifically, for every two lines attached to the same vertex one would add the penalty

$$|\cos(\theta)| = \frac{|x^T y|}{||x||||y||} \,,$$

to the total cost.

Chapter 6: Document Filtering

Book Notes

For this chapter I didn't do any of the problems but simply ran the code presented in the book. I will say that running the `fisherclassifier` on the RSS python feed data is really cool. To watch the classifier learn in real time is quite a nice effect. One can actually see the predictions of the blog entries improve as one tags the inputs. The script to run much of the examples from the book is presented in the chapter directory as `run_all.py`.

Chapter 7: Modeling with Decision Trees

Exercise Solutions

Exercise 1 (result probabilities)

We can map the count dictionaries to probabilities with a simple function

```
def count_to_prob( d ):
  total = sum( d.values() )
  res = {}
  for kk in d:
    res.setdefault(kk,d[kk]/total)
  return res
```

In the python script `chap_7_ex_1.py` several example of this function are demonstrated.

Exercise 2 (missing data ranges)

Now for this problem we will modify the logic in `mdclassify` when a feature is missing. When reviewing this code to make the needed modifications for this problem I think that I discovered an error. The code in the version of the book that handled missing feature cases looked like

```
    if( v==None ): # we have no measurement ... go down BOTH branchs
      tr,fr=mdclassify(observation,tree.tb),mdclassify(observation,tree.fb)
      tcount=sum(tr.values())
      fcount=sum(fr.values())
      tw=float(tcount)/(tcount+fcount)
      fw=float(fcount)/(tcount+fcount)
      result={}
      for k,v in tr.items(): result[k]=v*tw
      for k,v in fr.items(): result[k]=v*fw
      return result
```

The problem with this code is that the *assignment* in the `fr.items` loop in the second line from the end. In that loop if `tr` and `fr` share any keys in common this second statement will override the first rather than computing the needed average. For example if

```
tr = {'Premium': 3, 'Basic': 100}
fr = {'Basic': 1}
```

which is a case where its much more likely that this person will sign up for basic service but the buggy code above gives when we print result

```
{'Premium': 2.9711538461538463, 'Basic': 0.0096153846153846159}
```

indicating that premium is the more likely choice. The correct code would change these two for loops over the items of `tr` and `fr` lines to

```
for k,v in tr.items(): result[k] = v*tw # get everything from tr
for k,v in fr.items():
  if k not in result: result[k]  = v*fw # append new keys from fr
  else:               result[k] += v*fw # add old keys from fr
```

In that case when we print `result` we get

```
{'Premium': 2.9711538461538463, 'Basic': 99.048076923076934}
```

which is a more reasonable result.

To implement the requested performance for this problem we modify the `treepredict` function by adding an `if` statement that tests if the observation in the given column is a tuple or not. If it is a tuple we may have to send this observation to both sides of the tree. We will *not* have to do this if the values in the tuple are all on one side of the split point. If the split point is in the interior of the tuple then we take a percentage of the results from the left and a percentage of the results from the right. For example if the split point had a value of 5.0 and we supplied the tuple $(0.0, 15.0)$ then we would expect $1/3$ of the time an observation would be less than the split value and $2/3$ of the time our observation would be greater than the split value. Logic in this direction resulted in the following code to classify an instance that has tuple

```
elif( isinstance(v,tuple) ):
  assert v[1]>v[0], "incorrectly ordered tuple %10s encountered" % (v)
  maxv = max( v )
  minv = min( v )
  if( minv>=tree.value ):
    return mdclassify(observation,tree.tb)
  elif( maxv<tree.value ):
```

```
    return mdclassify(observation,tree.fb)
  else:
    assert v[0]<=tree.value<=v[1], "expected split point to be between tuple end p
    tr,fr=mdclassify(observation,tree.tb),mdclassify(observation,tree.fb)
    fw=float(tree.value-v[0])/(v[1]-v[0])
    tw=1.-fw
    result={}
    for k,v in tr.items(): result[k] = v*tw
    for k,v in fr.items():
      if k not in result: result[k]  = v*fw
      else:               result[k] += v*fw
    return result
```

In the python script `chap_7_ex_2.py` we test this new code by constructing a tree and then classifying several instances with tuples.

Exercise 3 (early stopping when tree building)

For this problem the code for the function `treebuild` has to be modified to accept a threshold such that if the reduction in entropy is too small for the best split at a point we will *not* perform the subsequent split. This is to prevent overfitting where by the produced decision tree has too many branches and is tuned to the intricacies of the specific training data set. To do this one needs to introduce a variable that specifies this threshold into the call of this routine. We denote this variable `mars` denoting the "minimum acceptable reduction in score" and change the function declaration to

```
def buildtree(rows,scoref=entropy,mars=0.):
```

Next we modify the splitting criterion depending how large the largest entropy gain found was large enough

```
  if best_gain > mars:
    trueBranch=buildtree(best_sets[0],scoref=scoref,mars=mars)
    falseBranch=buildtree(best_sets[1],scoref=scoref,mars=mars)
    return decisionnode(col=best_criteria[0],value=best_criteria[1],
                        tb=trueBranch,fb=falseBranch)
  else:
    return decisionnode(results=uniquecounts(rows))
```

Note that if we want any input parameter specifications to propagate to the trees built later we need to explicitly specify them in the recursive function calls. In the version of the code I was working off had a bug in that it didn't recursively pass the `scoref` to subtrees.

When we add the code above we can compare some differences between the trees built for different values of `mars`. In the python script `chap_7_ex_3.py` we find with no additional threshold `mars=0` that

```
tree = treepredict.buildtree( treepredict.my_data )
treepredict.printtree(tree)

0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> 0:slashdot?
  T-> {'None': 3}
  F-> 2:yes?
    T-> {'Basic': 4}
    F-> 3:21?
      T-> {'Basic': 1}
      F-> {'None': 3}
```

while with an additional threshold say `mars=0.3` that

```
tree = treepredict.buildtree( treepredict.my_data,mars=0.3 )
treepredict.printtree(tree)

0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> {'None': 6, 'Basic': 5}
```

showing that the splits down the "false" branch of the initial google split don't reduce the entropy that much are are perhaps not very informative out

of sample.

Exercise 4 (building a classification tree with missing data)

For this exercise we would like to modify the `buildtree` function to send any vectors with a None for a feature value in a given column to both sides of the potential split. Currently if a None is found as a feature the function `buildtree` will call `divideset` and will split the data into two sets: all vectors with None in the given component and those without. Thus the routine as written provides a reasonable way to handle None's.

If we consider the current logic in `buildtree`

```
# Now try dividing the rows up for each value in this column
for value in column_values.keys():
  (set1,set2)=divideset(rows,col,value)

  # Information gain
  p=float(len(set1))/len(rows)
  gain=current_score-p*scoref(set1)-(1-p)*scoref(set2)
  if gain>best_gain and len(set1)>0 and len(set2)>0:
    best_gain=gain
    best_criteria=(col,value)
    best_sets=(set1,set2)
```

We see that if one wanted to send all feature vectors with a None value in the given column down both sides of the tree one would need to compute the information gain twice; the first time with all None vectors in `set1` and the second time with the None vectors in `set2`. Each of the resulting two splits would be compared to the best split entropy reduction seen thus far.

Chapter 8: Building Price Models

Exercise Solutions

Exercise 1 (optimizing the number of neighbors)

We can create a "function generator" that will take a value for k, representing the number of nearest neighbors, perform k nearest neighbor cross validation on a given data set using that number of neighbors and return the mean square error obtained under each model. The python code to create this function generator is given by

```
def createKNNcostfunction(algf,data):
  def knn_costf(kk):
    return lambda x, y: algf(x,y,k=kk)
  def costf(kk):
    return numpredict.crossvalidate(knn_costf(kk),data,trials=20)
  return costf
```

Using code like the above we can perform cross-validation using three nearest neighbors with

```
knn_fn = createKNNcostfunction( numpredict.knnestimate, data )
knn_fn(3)
```

One can then very easy perform an exhaustive search of the number of neighbors using the commands

```
k_vals = range(1,15)
mse    = map( knn_fn, k_vals )
```

When we execute these commands and then plot the results we obtain the plot given in Figure 2 (left) This exercise is worked in the python script `chap_8_ex_1.py`.

Exercise 2 (leave-one-out cross-validation)

The python code to implement leave-one-out cross-validation is simple. One way to do this would be with the following function

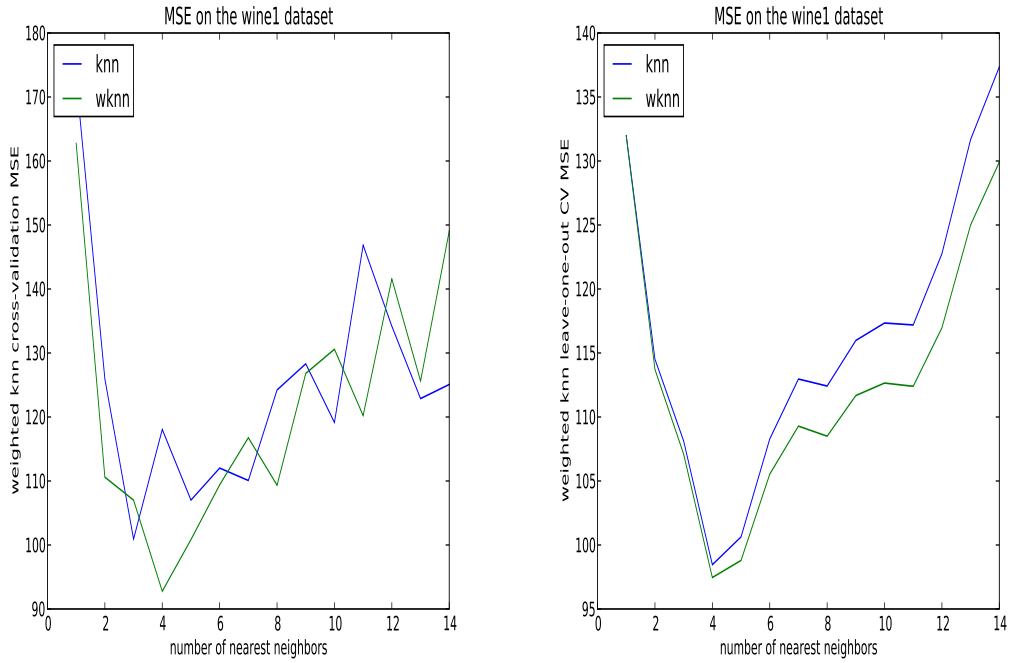

Figure 2: **Left:** Plots of the mean square error vs. the number of neighbors, when classifying the first wine data set with k-nearest neighbors (in blue) and weighted k-nearest neighbors (in green). From this plot it looks like a value of k between 3 or 4 is the optimal number of neighbors. **Right:** Plots of the estimated mean square error vs. the number of neighbors, when classifying the first wine data set with k-nearest neighbors (in blue) and weighted k-nearest neighbors (in green). In this case we use the method of leave-one-out cross validation to estimate the out of sample mean square error. Note that this method is much smoother than that derived from the cross-validation results presented in Exercise 1.

```
def leave_one_out_CV(algf,data):
  """
  Perfoms leave-one-out cross validation on the given data set
  """
  error=0.0
  for i in range(len(data)):
    testset   = [ data[i] ] # get the i-th sample in a list
    trainset  = data[0:i] + data[(i+1):] # exclude the i-th sample
    error    += numpredict.testalgorithm(algf,trainset,testset)
  return error/len(data)
```

Using this routine we can perform the same experiment as in Exercise 1 to observe how changing the number of nearest neighbors in k nearest neighbors affect the mean square error performance metric. In the python script `chap_8_ex_2.py`, we do this and then plot the results and obtain the plot given in Figure 2 (right). Note that these results are much smoother than those found in Exercise 1 and are probably more representative of what would be found out-of-sample. This result indicates that $k = 4$ looks to be the optimal choice for both methods.

Exercise 3 (eliminating variables)

One way to do this might to use some of the ideas from the section on the appropriate scaling to apply to various features. In that discussion we allowed an optimizer to find the best multiplicative scaling to apply to a set of features. If we simply want to specify *which* variables are predictive we could specify our possible feature scalings to take only the values 0 or 1. Then when we use the `rescale` function we get that any feature multiplied by a 0 will not contributed to the pointwise distance calculations. We can modify some of the code presented in this chapter to solve this problem. In the code `chap_8_ex_3.py` we use the `geneticoptimize` routine developed in the chapter on optimization with a possible domain for scaling each features specified by the tuple $(0, 1)$ which allows only two choices for scaling. The code to test this procedure on the the `wineset2` function is simple

```
# generate data and create a cost function to optimize over:
#
data  = numpredict.wineset2()
costf = numpredict.createcostfunction( numpredict.knnestimate, data )
```

```
# next create a domain that will have only two values
# 0 => that feature is not needed
# 1 => that feature will be used
#
n_vars = len(data[0]['input'])
domain = [(0,1)] * n_vars

optimization.geneticoptimize( domain, costf, popsize=50, step=1, maxiter=20 )
```

When we run this we get

```
[1, 1, 0, 1]
```

Indicating that the third dimension (which is the aisle variable) is not important to making predictions on price.

We can also try to make this problem more challenging by constructing data in the same way as we have above but appending five additional measurements that tell us nothing about the price of wine. In the above python script we define a function `wineset4` that does just that. We then run our "feature" selector code in much the same way as earlier. When we do this we now have feature vectors with 9 inputs 6 of which don't provide any information. An example vector in this case looks like

```
{'input': (50.641494249981434,
           12.656975253781578,
           14.0,
           375.0,
           12.371621826161466,
           10.340159242792581,
           3.6478535849936455,
           10.280064086090633,
           2.8716313510491078),
 'result': 0.0}
```

Given this input data when we run the command

```
optimization.geneticoptimize( domain, costf, popsize=50, step=1, maxiter=20 )
```

we end up with

```
[1, 1, 0, 1, 0, 0, 0, 0, 0]
```

which is quite a nice result since it explicitly shows the variables that are not relevant.

Chapter 9: Advanced Classification: Kernel Methods and SVMs

Exercise Solutions

Exercise 1 (Bayesian classifier)

One example of using Bayesian classification that is outside of chapter 6 (where this technique was discussed) is given in chapter 10 where we classify articles based on the count of the words in each article. In this chapter we could train a Bayesian classifier using real valued features such as age. The data set `ageonly` has already been separated into two classes: the pairs of ages that are good matches and the ones that are not.

Exercise 2 (Optimizing a dividing line)

The book talks about using a SVM classifier but does not discuss how to train one. In fact the dividing line obtained when training a SVM classifier is computed as the solution to an optimization problem. This optimization problem results in a solution that is different than the individual class averages. This is best explained in more specialized source, for example see [1].

Exercise 3 (Choosing the best kernel parameters)

In the python function `chap_9_ex_3.py` we implement some functions to perform optimization over the radial basis function (RBF) kernel parameter gamma. Some of the results we obtain when we run this routine are shown in in Figure 3. In that figure we present a plot of the probability of error for a radial basis function classifier as a function of the kernel parameter gamma, where gamma is taken to be an integer between 1 and 50. In that plot we see that for the region of gamma greater than about 15 there does not seem to be large changes in the value of classification error probability and the error probability does not change much. For fun, in the python function `chap_9_ex_3.py` we also use some of the optimization routines developed in chapter 5 to find the "optimal" setting for gamma. These routines find that gamma ≈ 35 seems best. From the above plot this seems to be a reasonable number. Note that I did not take the time to get a Yahoo! application key so the `milesdistance` routine is certainly incorrect. With a different version of that function one would certainly get different results.

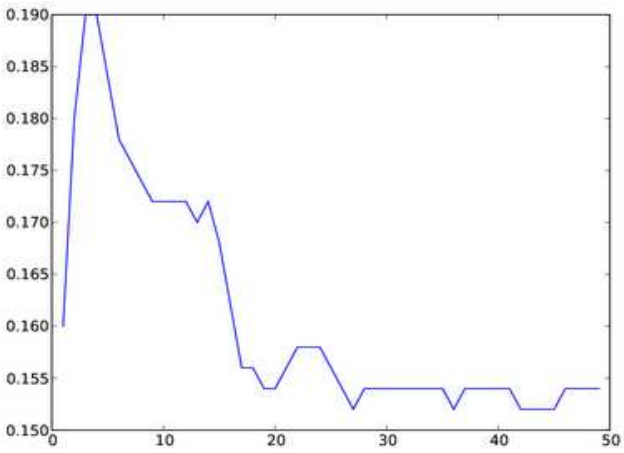

Figure 3: A plot of the estimated probability of error as a function of gamma when classifying the matchmaker data set using a RBF classifier.

Chapter 11: Evolving Intelligence

Book Notes

Notes on a simple mathematical test

It is very interesting to run the genetic algorithm developed in this chapter on the numeric data set generated by the user specified function `hiddenfunction`. While similar results are discussed in the book it is very interesting to observe the trees that the `evolve` routine generates when one runs the command

```
gp.evolve(2,500,rf,mutationrate=0.2,breedingrate=0.1,pexp=0.7,pnew=0.1)
```

At the end of that routine a program (displayed as a tree) is presented. On one of the calls gave the following program

```
add
 multiply
  p0
  add
   p0
   3
 add
  p1
  add
   5
   p1
```

When we convert this program into its equivalent algebraic expression recalling that `p0` is x and `p1` is y we see that it is equivalent to

$$(x(x+3)) + ((y+5)+y) = x^2 + 3x + 2y + 5\,,$$

the exact function we used to generate the given results! Since on each call to the `evolve` function we begin our algorithm search using a randomly selected population another run can and will give a different solution to the same problem. A second run gave

```
add
 add
  subtract
   multiply
    add
     p0
     5
    p0
   p0
  add
   p1
   6
 add
  subtract
   p1
   p0
  subtract
   8
   9
```

When we write this as its equivalent algebraic expression we get

$$\begin{aligned}[(x(x+5)+(y+6)]+[(y-x)+(8-9)] &= x^2+5x-x+y+6+y-x-1 \\ &= x^2+3x+2y+5\,.\end{aligned}$$

As an aside, it would seem useful to have the given program display itself as an algebraic expression

Exercise Solutions

Exercise 1 (other function types)

Other functions could be division, logarithms, trigonometric etc. There is really no limit on what type of functions one could consider. We can implement a Euclidean distance function (and create a wrapper for it) with the python code

```
def euclidianDist(l):
  """
  The Euclidian distance between the points (l[0],l[1]) and (l[2],l[3])
  """
  return sqrt( (l[0]-l[2])**2 + (l[1]-l[3])**2 )
edw=fwrapper(euclidianDist,4,'euclidiandistance')
```

Exercise 2 (replacement mutation)

See the function `function_replacement_mutate` for the implementation of the discussed mutation function and the script `chap_11_ex_2.py` for an example at using this function. To test this new function we modified the `evolve` function to use this new function rather than the books `mutate` function. For some reason after some initial convergence towards a solution this new function seemed to *slow down* convergence.

Exercise 4 (stopping evolution when there is no change in the score)

For this exercise we simply save the optimal score found in a given generation. We compare this value to the optimal score found in the next generation. If there has been no change in this optimal value for some number of generations we simply exit our search loop early. This procedure was needed to work other exercises and is implemented in the `evolve` function of the `gp` module.

Exercise 5 (alternative mathematical functions to guess)

One alternative to the `hiddenfunction` function that is interesting to consider is to modify the `hiddenfunction` by adding noise to it. This is done in the `noise_hiddenfunction` function which is implemented in the `gp` module. We can run our genetic program with data generated via this function and observe its behavior. In general we cannot expect our algorithm to drive the error to zero and the `evolve` routine as provided will only end when it has considered all `maxgen` generations (which can take a long time). When we run `evolve` on this function we see that for this type problem our genetic algorithm will initially be able to reduce the objective function but at some point the best scores from the subsequent generations will all be the same (or nearly the same), thus we need to implement the suggestion from Exercise 4. Thus it seems that the genetic algorithm has "stalled". It eventually returns the best program it could come up with.

A second alternative that is interesting to consider is a function that has operations that are *not* expressed in the provided nodes vocabulary. For example, in the examples from this chapter the *divide* function is not a possible candidate and I would expect that our genetic algorithms would have a hard time at approximating a function that has a division operation. We tested this hypothesis on the function

$$\frac{1}{x+1},$$

and found that the function was rather poorly approximated. If we add a division function and its wrapper with the following code

```
def simple_divide(l):
  if(l[1]==0):
    return float(100.) # +infinity
  else:
    return float(l[0])/l[1]
divw=fwrapper(simple_divide,2,'divide')
```

and rerun the corresponding genetic program we quickly converge to the correct functional representation. One solution found while running `evolve` was

```
divide
 2
 subtract
  add
   p0
   p0
  subtract
   3
   5
```

When we express this result algebraically we get

$$\frac{2}{x+x-(3-5)} = \frac{2}{2x+2} = \frac{1}{x+1},$$

the input expression.

References

[1] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 2001.

[2] S. Theodoridis and K. Koutroumbas. *Pattern Recognition, Third Edition*. Academic Press, Inc., Orlando, FL, USA, 2006.