7. Machine Learning

February 16, 2014

In [112]:
```
import RDatasets
import DataFrames
import Clustering
import SVM
import DecisionTree
```

In [112]:
```
# read data
iris = RDatasets.data("datasets","iris");
features = matrix(iris[:,2:5]);
labels = matrix(iris)[:,6];
sz,ft = size(features);
train = randbool(sz)
test = !train;
```

1 K-means Clustering

In [112]:
```
nc=3
km=Clustering.kmeans(features',nc);
centers=km.centers;
```

```
  Iters               objv        objv-change | affected
-------------------------------------------------------------
      1       9.215602e+01      -5.825398e+01 |        2
      2       8.368890e+01      -8.467122e+00 |        3
      3       7.986398e+01      -3.824915e+00 |        2
      4       7.919714e+01      -6.668418e-01 |        2
      5       7.885144e+01      -3.457012e-01 |        0
      6       7.885144e+01       0.000000e+00 |        0
K-means converged with 6 iterations (objv = 78.85144142614597)
```

In [113]:
```
km.centers
```

Out [113]:
```
4x3 Array{Float64,2}:
 5.90161  5.006  6.85
 2.74839  3.428  3.07368
 4.39355  1.462  5.74211
 1.43387  0.246  2.07105
```

2 SVM

In [114]:
```
Y = [species == "setosa" ? 1.0 : -1.0 for species in iris[:, "Species"]];
X=features'
model=SVM.svm(X[:,train],Y[train])
prediction=SVM.predict(model, X[:,test])
DecisionTree.confusion_matrix(prediction,Y[test])
```

Out [114]:
```
Classes:  [-1.0,1.0]
Matrix:   2x2 Array{Int64,2}:
 50   0
  0  22
Accuracy: 1.0
Kappa:    1.0
```

3 Random Forest

In [114]:
```
modelrf=DecisionTree.build_forest(labels[train],features[train,:],4,30);
predrf=DecisionTree.apply_forest(modelrf,features[test,:]);
println(DecisionTree.confusion_matrix(labels[test],predrf));
```

```
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 22   0   0
  1  24   2
  0   3  20
Accuracy: 0.9166666666666666
Kappa:    0.8745280278826603
```

In [114]:
```
accuracyrf = DecisionTree.nfoldCV_forest(labels, features, 4, 30, 3);
```

```
Fold 1
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 17   0   0
  0  15   0
  0   1  17
Accuracy: 0.98
Kappa:    0.96996996996997

Fold 2
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 16   0   0
  0  16   1
  0   3  14
Accuracy: 0.92
Kappa:    0.879951980792317

Fold 3
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 17   0   0
  0  16   2
  0   1  14
Accuracy: 0.94
Kappa:    0.9099099099099098

Mean Accuracy: 0.9466666666666667
```

4 Adaptive-Boosted Tree (Adaboost)

In [114]:
```
modelad,coeffs=DecisionTree.build_adaboost_stumps(labels[train],features[train,:],50);
predad=DecisionTree.apply_forest(modelad,features[test,:]);
println(DecisionTree.confusion_matrix(labels[test],predad));
```

```
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 22  0   0
  1  8  18
  0  0  23
Accuracy: 0.7361111111111112
Kappa:    0.6112531969309464
```

In [114]:
```
accuracyad = DecisionTree.nfoldCV_stumps(labels, features, 50, 3);
```

```
Fold 1
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 0  22   0
 0  18   0
 0   0  10
Accuracy: 0.56
Kappa:    0.34523809523809534

Fold 2
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 13  0   0
  0  5  13
  0  0  19
Accuracy: 0.74
Kappa:    0.6019595835884874

Fold 3
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 15   0   0
  0  10   4
  0   1  20
Accuracy: 0.9
Kappa:    0.8453927025355595

Mean Accuracy: 0.7333333333333334
```

5 Pruned Tree

In [114]:
```
modelpt=DecisionTree.build_tree(labels[train],features[train,:])
predpt=DecisionTree.apply_tree(modelpt,features[test,:]);
println(DecisionTree.confusion_matrix(labels[test],predpt));
```

```
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 22   0   0
  1  25   1
  0   3  20
Accuracy: 0.9305555555555556
Kappa:    0.8953184065135215
```

```
accuracypt = DecisionTree.nfoldCV_tree(labels, features, 0.9, 3);
```

In [114]:
```
Fold 1
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 20   0   0
  0  14   1
  0   0  15
Accuracy: 0.98
Kappa:    0.9696969696969696

Fold 2
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 15   0   0
  0  15   0
  0   1  19
Accuracy: 0.98
Kappa:    0.9697885196374622

Fold 3
Classes:  {"setosa","versicolor","virginica"}
Matrix:   3x3 Array{Int64,2}:
 15   0   0
  3  15   2
  0   1  14
Accuracy: 0.88
Kappa:    0.8203592814371259

Mean Accuracy: 0.9466666666666667
```

6 Neural Network

In [8]:
```
# https://github.com/nwenzel/Julia_Neural_Network
# Code by Nathan Wenzel

function sigmoid(z)
  # sigmoid is a basic sigmoid function returning values from 0-1
  1. / ( 1. + 1e.^(-z) )
end

function sigmoidGradient(z)
  sigmoid(z) .* ( 1 - sigmoid(z) )
end

function initialize_theta(input_unit_count, output_class_count, hidden_unit_length_lis
  #
  #initialize_theta creates architecture of neural network
  #
  #Parameters:
  #  hidden_unit_length_list - Array of hidden layer units
  #  input_unit_count - integer, number of input units (features)
  #  output_class_count - integer, number of output classes
  #
  #Returns:
  #
  #Array of theta arrays randomly initialized to from -.5 to .5
  #
```

```
  if length( hidden_unit_length_list ) == 0
    hidden_unit_length_list = [2]
  end

  unit_count_list = [input_unit_count]
  unit_count_list = [unit_count_list, hidden_unit_length_list]
  unit_count_list = [unit_count_list, output_class_count]
  layers = length(unit_count_list)

  Theta_L = [ rand( unit_count_list[i], unit_count_list[i-1]+1 ) - .5 for i = 2:layers

end

function print_theta(Theta_L)
  # print_theta() is a helper function that prints Theta_L and architecture info
  # It does not actually "do" anything except print to stdout

  T = length(Theta_L)

  println()
  println( "NN ARCHITECTURE" )
  println( "$(T+1) Layers ($(T-1) Hidden)" )
  println( "$T Thetas" )
  println( "$(size(Theta_L[1],2)-1) Input Features" )
  println( "$(size(Theta_L[end], 1)) Output Classes" )
  println()

  println( "Units per layer (excl. bias unit)" )
  for t = 1:T
    if t == 1
      println( " - Input: $(size(Theta_L[t],2)-1) Units" )
    end
    if t < T
      println( " - Hidden $t: $(size(Theta_L[t],1)) Units" )
    else
      println( " - Output: $(size(Theta_L[t],1)) Units" )
    end
  end
  println()

  println( "Theta Shapes" )
  for l = 1:T
    println( "Theta $l: $(size(Theta_L[l]))" )
  end
  println()

  println( "Theta Values" )
  for t= 1:T
    println( "Theta $t:"  )
    println ( Theta_L[t])
  end
  println()

end

function nn_cost(Y, Y_pred)
  #
  # nn_cost implements cost function for array inputs Y and Y_pred
  #
  # y is array of n_observations by n_classes
  # Y_pred is array with same dimensions as Y of predicted y values
  #
  if size(Y) != size(Y_pred)
    if size(Y,1) != size(Y_pred,1)
      error("Wrong number of predictions", "$(size(Y,1)) Actual Values. $(size(Y_pred,
```

```
    else
      error("Wrong number of prediction classes", "$(size(Y,2)) Actual Classes. $(size
    end
  end

  n_observations = size(Y,1)

  # Cost Function
  J = (-1.0 / n_observations ) * sum((Y .* log(Y_pred)) + ((1-Y) .* log(1-Y_pred)))

  J

end



function nn_predict(X, Theta_L)
  #
  # nn_predict calculates activations for all layers given X and thetas in Theta_L
  # return all inputs and activations for all layers for use in backprop
  #
  # Parameters
  #  X is matrix of input features dimensions n_observations by n_features
  #  Theta_L is a 3D array where first element corresponds to the layer number, second
  #
  # Returns
  #  a_N - 1D Array of activation 2D arrays for each layer: Input (1), Hidden (2 to T)
  #  a_Z - 1D Array of input 2D arrays to compute activations for all non-bias units
  #  a_N[end] - 2D Array of predicted Y values with dimensions n_observations by n_cla
  #

  a_N = {}
  z_N = {}

  m = size(X,1)
  T = length(Theta_L)

  # Input Layer inputs
  push!(a_N, X)  # List of activations including bias unit for non-output layers
  push!(z_N, zeros(1,1))  # add filler Z layer to align keys/indexes for a, z, and The

  # Loop through each Theta_List theta
  # t is index of Theta for calculating layer t+1 from layer t
  for t=1:T
    # Reshape 1D Array into 2D Array
    if ndims(a_N[t]) == 1
      a_N[t] = reshape(a_N[t], 1, size(a_N[t],1))
    end

    # Add bias unit
    a_N[t] = [ ones(size(a_N[t],1), 1)  a_N[t] ]

    # Calculate and Append new z and a arrays to z_N and a_N lists
    push!(z_N, a_N[t] * Theta_L[t]') #'
    push!(a_N, sigmoid(z_N[t+1]))
  end

  z_N, a_N, a_N[end]

end


function back_prop(X_train, Y_train, Theta_L, lmda)
  #
  # Parameters
```

```
  #  X_train - Array of feature inputs with dimensions n_observations by n_features
  #  Y_train - Array of class outputs with dimensions n_observations by n_classes
  #  Theta_L is a 1D array of Theta values where 1D element is the layer number, the 2
  #  lmda - Float64 - lambda term for regularization
  #
  # Returns
  #  Y_pred as array of predicted Y values from nn_predict()
  #  Theta_Gradient_L as 1D array of 2D Theta Gradient arrays
  #

  n_observations = size(X_train,1)

  T = length(Theta_L)

  # Create Modified copy of the Theta_L for Regularization with Coefficient for bias u
  # Create variable to accumulate error caused by each Theta_L term in layer a_N[n+1]
  Theta_Gradient_L = {}
  regTheta = {}
  for i=1:T
    push!(regTheta, [zeros(size(Theta_L[i],1),1) Theta_L[i][:,2:]])
    push!(Theta_Gradient_L, zeros(size(Theta_L[i])))
  end


  # Forward Pass
  z_N, a_N, Y_pred = nn_predict(X_train, Theta_L)

  # Backprop Error Accumulator
  delta_N = {}
  for t=1:T
    push!(delta_N, {})
  end

  # Error for Output layer is predicted value - Y training value
  delta = Y_pred - Y_train
  if ndims(delta) == 1
    delta = reshape(delta, 1, length(delta) )
  end

  # Loop backwards through Thetas to apply Error to prior Layer (except input layer)
  # Finish at T-2 because start at 0, output layer is done outside, the loop and input

  # Output Error
  delta_N[T] = delta

  # Hidden Layers Error
  for t=0:T-2
    delta = (delta * Theta_L[T-t][:,2:]) .* sigmoidGradient(z_N[T-t])
    delta_N[T-t-1] = delta
  end

  # Calculate error gradients (no error in input layer)
  # t is the Theta from layer t to layer t+1
  for t=1:T
    Theta_Gradient_L[t] = Theta_Gradient_L[t] + delta_N[t]' * a_N[t] #'
  end

  # Average Error + regularization penalty
  for t=1:T
    Theta_Gradient_L[t] = Theta_Gradient_L[t] * (1.0/n_observations) + (lmda * regThet
  end

  Y_pred, Theta_Gradient_L

end
```

```
function fit(X_train, Y_train, Theta_L, lmda, epochs)
  #
  #fit() calls the training back_prop function for the given number of cycles
  #tracks error and error improvement rates
  #
  #Parameters:
  #  X_train - Array of training data with dimension n_observations by n_features
  #  Y_train - Array of training classes with dimension n_observations by n_classes
  #  Theta_L - 1D array of theta 2d arrays where each theta has dimensions n_units[lay
  #  epochs -  integer of number of times to update Theta_L
  #
  #Returns
  #  Theta_L - 1D array of Theta arrays
  #  J_List - Array (length = epochs) of result of cost function for each iteration

  J_list = zeros( epochs )

  for i=1:epochs
    # Back prop to get Y_pred and Theta gradient
    Y_pred, Theta_grad = back_prop(X_train, Y_train, Theta_L, lmda)
    # Record cost
    J_list[i] = nn_cost(Y_train, Y_pred)
    # Update Theta using Learning Rate * Theta Gradient
    for t=1:length(Theta_L)
      # Start with a large learning rate; need to update this to be more intelligent t
      # Need to update to change learning rate based on progress of cost function
      if i < 100
        learning_rate = 5.0
      else
        learning_rate = 1.0
      end
      Theta_L[t] = Theta_L[t] - ( learning_rate * Theta_grad[t] )
    end
    #println("Cost $i: $(J_list[i])")
  end

  Theta_L, J_list

end


function XOR_test(hidden_unit_length_list, epochs)
  #
  #XOR_test is a simple test of the nn printing the predicted value to std out
  #Trains on a sample XOR data set
  #Predicts a single value
  #Accepts an option parameter to set architecture of hidden layers
  #


  println( "Training Data: X & Y")

  # Training Data
  X_train = [1 1; 1 0; 0 1; 0 0] # Training Input Data
  Y_train = [0 1; 1 0; 1 0; 0 1]  # Training Classes
  println( X_train )
  println( Y_train )

  # Hidden layer architecture
  hidden_layer_architecture = hidden_unit_length_list

  # Regularization Term
  lmda = 1e-5
```

```
    # Initialize Theta based on selected architecture
    Theta_L = initialize_theta(size(X_train,2), size(Y_train,2), hidden_layer_architectu


    # Fit
    Theta_L, J_list = fit(X_train, Y_train, Theta_L, lmda, epochs)

    # Print Architecture
    print_theta(Theta_L)

    # Print Cost
    println("Cost Function Applied to Training Data: $(J_list[end])")

    # Predict
    X_new = [1 0;0 1;1 1;0 0]
    println( "Given X:")
    println("$X_new")
    z_N, a_N, Y_pred = nn_predict(X_new, Theta_L)
    println( "Predicted Y:")
    println("$(round(Y_pred,3))")

    Y_pred

end
```

Out [8]:

```
XOR_test (generic function with 1 method)
```

In [14]:

```
Y_pred = XOR_test([2], 5000)
```

```
Training Data: X & Y
1       1
1       0
0       1
0       0

0       1
1       0
1       0
0       1



NN ARCHITECTURE
3 Layers (1 Hidden)
2 Thetas
2 Input Features
2 Output Classes

Units per layer (excl. bias unit)
 - Input: 2 Units
 - Hidden 1: 2 Units
 - Output: 2 Units

Theta Shapes
Theta 1: (2,3)
Theta 2: (2,3)

Theta Values
Theta 1:
-4.244057658582526      8.011119225793868       -8.218268440922987
3.9453941972708795      8.156949365387408       -7.86130314531543
```

```
Theta 2:
5.735531100620542       12.358587250630872      -11.947865946940489
-5.7395079853749005     -12.366440373984625     11.955769589062232


Cost Function Applied to Training Data: 0.006301094016920786
Given X:
1       0
0       1
1       1
0       0

Predicted Y:
.997    .003
.996    .004
.003    .997
.003    .997
```

Out [14]:

```
4x2 Array{Float64,2}:
 0.997177    0.0028127
 0.995939    0.00404581
 0.00273109  0.997279
 0.00298615  0.997025
```