Guillaume Fray, **17008776** Wednesday, 7 October 2020

Machine Learning Coursework 1:

Experimental comparison of k-NN and linear classification
on the Iris data-set

1. Introduction

1.1 Classification problem

A classification problem in Machine Learning is a supervised learning problem where each given example of input data must be recognised by the algorithm as belonging to a certain group, called a class.
A dataset is provided, which contains several groups of labelled data to train the model. Once the model has been trained, new examples of input data are passed into the model, and each of them is given a predicted discrete value or a label that corresponds to the class it belongs to.

1.2 Iris dataset

The Iris dataset provided for this experiment consists of one *150 by 4 matrix of doubles*, named *meas*, and one *150 by 1 cell array of Strings*, named *species*. Each of the 150 rows for both the matrix and the cell array corresponds to one given Iris example.

- For the matrix, each of the 4 columns is a particular dimension of the Iris chosen features, defined as follows:

  column 1: Sepal length in **cm**
  column 2: Sepal width in **cm**
  column 3: Petal length in **cm**
  column 4: Petal width in **cm**

  Sepals are the leaves of a flower, usually green.

- Each string in the only column, corresponds to the species' type of each particular example of Iris. There are 3 different species in this dataset:
  - setosa
  - versicolor
  - virginica

  These 3 string values are the labels of this particular classification problem.

1.3 kNN and linear classifications

- The k-nearest neighbours or kNN classification is a non-parametric method and algorithm to solve a classification problem. The output is thus a class membership. This method is not linear.

An object is classified by a plurality vote of its neighbours, with the object being assigned to the most common class among its *k* nearest neighbours (*k* is a positive integer). Often, different weights are assigned to the contributions of the neighbours, so that the nearer neighbours contribute more to the average than the more distant ones.

- The linear classification uses an object's features to identify which class it belongs to. A linear classifier achieves this by making a classification decision based on the value of a linear combination of the features, represented as vectors. Often, different weights are assigned to the contributions of each of these features.

1.4 Application of the kNN and linear models to the Iris dataset

- The kNN model will be given new Iris input examples. Based on its training values, including “distances” to previous Iris examples, weights for each neighbour and k value, the model will decide which species each Iris example belongs to.
- The linear classification model, on the other hand, will take in values from all 4 columns, where each column corresponds to one Iris feature, and try to establish the optimum linear relationship between them. Different weights will be given for each of those 4 features, according to how much they define the Iris. For example, the sepals’ dimensions could be less specific than the Petals’ dimensions.

2. Experiment

2.1 Description

The “bagging” method will be used to evaluate and compare the 2 classification models described above, because the data set given is relatively small. With this method, we can obtain:

$$\frac{n!}{(n-k)!k!}$$ different sets, as the elements are not repeated

(we only count one particular Iris once in a set), and the order of the data inputs in the set does not matter (every Iris matters as much as the next one).

n = number of Iris in the training-validation set
k = number of Iris in the training data set
n-k = number of Iris in the validation data set

For example, for a training-validation set of 90 Iris split into 60 Iris for the training set and 30 for the validation set, we obtain **6.7313297e+23** possible training combinations, whereas with the “leave one out” method, we could only obtain **n** different sets. **90** in this example.

Both classification methods will be tested in a range of different set splits to observe how well they perform in different scenarios. Thereby, one scenario will correspond to one combination of splits.
There will be 2 splits:

- **Outer split**
- **Inner split**

The outer split divides the original Iris data set into one **training-validation set** and one **test set**. The test set data will be put aside and will only be used once at the end of the program run.
The inner split contains the data that will be randomly shuffled by matrix permutation (to keep all features of each Iris together). This split divides the training-validation set into one **training set** and one **validation set.**
There will be 1000 shuffles in order to create 1000 different pairs of sets. Each created pair of sets will be tested once: this is one fold. For each fold, the chosen classification model will be trained on the training data and be tested on the validation data.
Every program run will include:

- 1,000 folds = 1,000 training-validation random shuffles and splits
- 1,000 average fold-errors
  where 1 average fold error = number of wrong predictions within one fold, divided by number of data inputs in the validation set
- 1 **Average Validation Error** = sum of the 1,000 average folds-errors divided by 1,000
- 1 **Average Test Error** = number of wrong predictions made using the test set, divided by the number of data inputs in the test set.

Graphs and tables will be shown for the first scenario to illustrate the testing process. Then the results for the other scenarios will be summarised in a table.

Each scenario will consist of 3 test phases:

- **Phase 1:** 10 program runs testing the linear classification model (10,000 tests + 1 final test on unused data)
- **Phase 2:** 1 program run to determine the best k value (25,000 tests)
- **Phase 3:** 10 program runs testing the kNN model (10,000 tests + 1 final test on unused data)

2.2 First Scenario: error rate evaluation method example

Scenario 3.3 **Outer split: 90/60** **Inner split: 60/30**					
Test Run ID	**ML Classification Model**	**k**	**No of random “folds” (Bagging)**	**Average Validation Error**	**Average Test Error (on new data)**
1.1	Linear	N/A	1,000	0.0330	0.0333

⋮	⋮	⋮	⋮	⋮	⋮
1.10	Linear	N/A	1,000	0.0333	0.0500
Total average for test runs **1.1 to 1.10**: • Total Average for Validations Errors = **0.03312** • Total Average for Tests Errors = **0.0300** • Program's Total Elapsed Time: **181.457877 sec**					
2.1	kNN	1	1,000	0.0553	0.0333
2.2	kNN	2	1,000	0.0626	0.0667
2.3	kNN	3	1,000	0.0514	0.0167
2.4	kNN	4	1,000	0.0555	0.0333
⋮	⋮	⋮	⋮	⋮	⋮
2.25	kNN	25	1,000	0.1095	0.1000
Further testing for best k value, **k = 3**, to compare with linear model.					
2.3.1	kNN	3	1,000	0.0513	0.0167
⋮	⋮	⋮	⋮	⋮	⋮
2.3.10	kNN	3	1,000	0.0519	0.0333
Total average for test runs **2.3.1 to 2.3.10**: • Average Validations Error = **0.05165** • Average Tests Error = **0.02335** • Program's Total Elapsed Time: **62.301395 sec**					

Table 1: *Scenario 3.3 tests results*

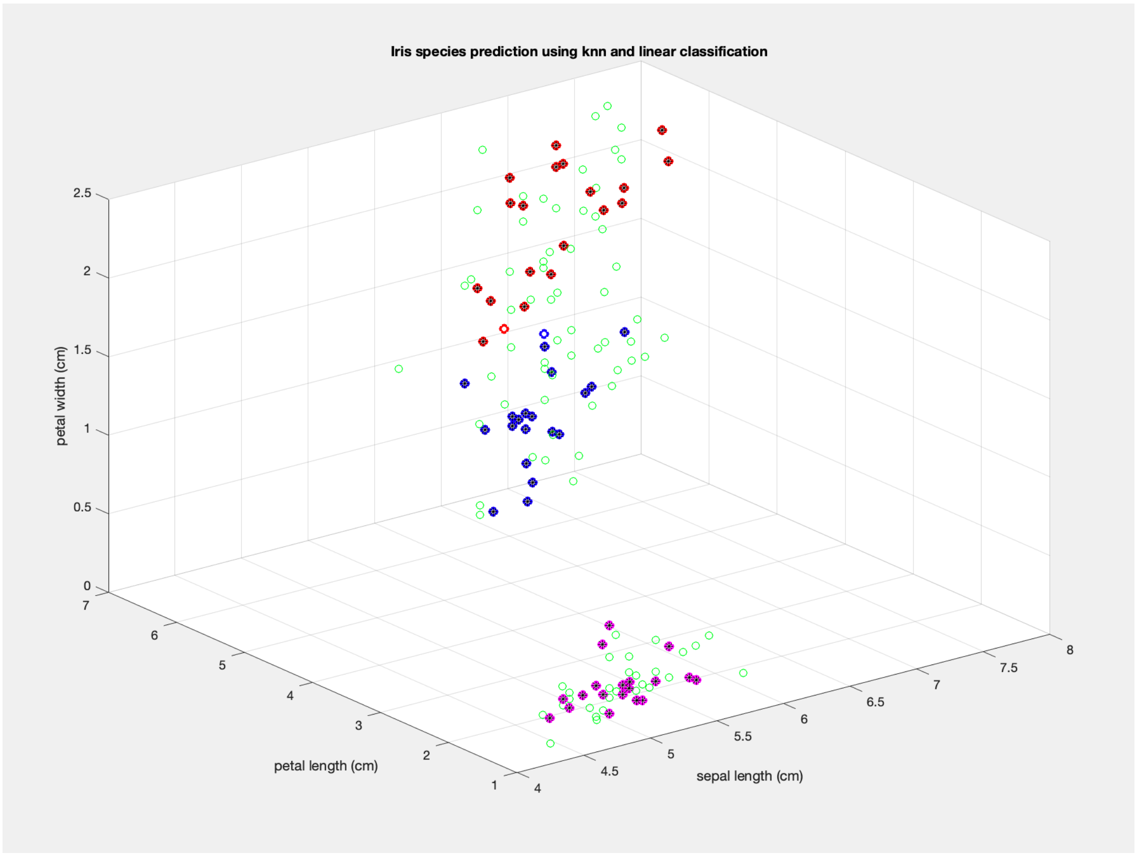

Figure 1. *Iris Predictions Scatter Graph*

The axes of this coordinate frame correspond to the Iris' petals length and width, and the sepal length. The green points are all the Iris examples from the original data set. The other filled colored points are the test examples correctly labelled. (magenta for Setosa Iris; blue for Versicolor Iris; red for Virginica Iris).
The 2 hollow red and blue circles are the Iris examples that the model could not label correctly in one particular test run for this scenario (3.3). This final test run shows therefore an average test error A = 2 errors / 60 Iris examples (test set), which is A = **0.0333** or 3.33%, like in test runs 1.1, 2.4 and 2.3.10 for instance.

2.3 Summary of all tested scenarios

Scenario ID	**Outer Split** (Train-Valid / Test)	**Inner Split** (Training / Validation)	**ML Classification Model**	**Optimum k value**	**Elapsed Time (s)**	**Total Average Validation Error**	**Total Average Test Error**	**Best Pre-trained Model**
1.1	60/90	15/45	Linear	N/A	174	0.0475	0.0711	Linear
			kNN	1	44	0.0570	0.0722	
1.2	60/90	30/30	Linear	N/A	174	0.0219	0.0467	Linear
			kNN	3	45	0.0337	0.0644	
1.3	60/90	45/15	Linear	N/A	160	0.0187	0.0433	Linear
			kNN	3	44	0.0228	0.0611	
2.1	75/75	25/75	Linear	N/A	169	0.0416	0.0493	Linear
			kNN	3	48	0.0587	0.0600	
2.2	75/75	45/30	Linear	N/A	184	0.0291	0.0560	Linear
			kNN	5	48	0.0443	0.0573	
2.3	75/75	65/10	Linear	N/A	170	0.0242	0.0400	Linear
			kNN	5	45	0.0323	0.0720	
3.1	90/60	60/30	Linear	N/A	181	0.03312	0.0300	kNN
			kNN	3	62	0.05165	0.02335	
3.2	90/60	45/45	Linear	N/A	186	0.03839	0.06672	kNN
			kNN	3	61	0.05679	0.03668	
3.3	90/60	70/20	Linear	N/A	189	0.03046	0.03166	kNN
			kNN	3	49	0.04857	0.02169	
3.4	90/60	80/10	Linear	N/A	179	0.02975	0.02501	kNN
			kNN	3	58	0.0469	0.0167	
4.1	120/30	30/90	Linear	N/A	167	0.0563	0.0167	Linear
			kNN	1	49	0.0725	0.0233	
4.2	120/30	60/60	Linear	N/A	176	0.0432	0.0000	Linear
			kNN	3	58	0.0572	0.0100	
4.3	120/30	90/30	Linear	N/A	170	0.0000	0.0408	kNN
			kNN	3	52	0.0504	0.0000	
4.4	120/30	110/10	Linear	N/A	182	0.0431	0.0000	kNN
			kNN	5	48	0.0443	0.0000	

Table 2: *Tests results summary for all scenarios*

3. Analysis and Conclusion

According to the results shown in Table 2 above, the kNN model appears to be more effective with small odd numbers, with a predominance for the k = 3 value (64.3% against 21.4% and 14.3% for k = 5 and k = 1 respectively). This apparent “odd feature” could be explained by the fact that even numbers could lead to draws, which are prone to errors (e.g. For k = 2, 1 nearest neighbour could be a Versicolor Iris and the other nearest neighbour a Virginica). A high value of k is also prone to errors because it might be greater than the number of input examples belonging to a class, taking therefore the other nearest neighbours from another class. Furthermore, if one class has a larger amount of examples than the other, then a high k value will most likely lead the model to predict that a given input belongs to the larger class.

It is essential to highlight the fact that after each validation test, except the last one, the model is discarded. This means that the model (n) is not being pretrained with the previous model (n-1), from one fold to another.
As we can see in Table 2 above, the total average validation error of the linear classification model is always smaller than that of the kNN model.
Therefore, when the model has not been pretrained, the linear classification model outperforms the kNN model.
However, at the end of a program run, the last model of the training-validation tests is reused for the final test. In other words, in the final test both the kNN and the classification models have been pretrained as they use the “knowledge” or the “experience” acquired in the last fold (the 1000th fold). The second last column in Table 2 shows that when they do so, both models offer similar results and performances.
But this particular pattern, of how much training both models require, also emerges in the data of the Outer Split and Inner Split columns. The highlighted figures show that when the training set contains a relatively small amount of inputs in either the outer or the inner splits, the linear model will most likely outperforms the kNN model.

If we now take other factors into account, such as computational complexity, kNN clearly outperforms the linear model, as the latter requires up to 3 times more time to perform the same amount of predictions. This is shown in the “Elapsed Time” column that represents the total elapsed time of a program run (1 run: 1,000 + 1 tests).

We can therefore conclude that for cases where we cannot pretrain a model or where the amount of training data inputs is insufficient, the linear classification model will be the most suitable option. In cases where the data set is large enough and pretraining is possible, based on these tests, the non-linear kNN model appears to outperform the linear one.

Note: all test results can be found in the Github repository provided.

CODE

Main.m

```
% Main File
% tic toc here is to measure the elapsed time for each program run in order
% to assess the performance
tic
iris_data = load('fisheriris.mat');
X = iris_data.meas;
Y = iris_data.species;
%Mean folds error of ALL folds for one particular model
global e;
e = 0;



% --- Draw all Iris points ---
% the sepal width has been taken out, but ONLY for the visualisation
% sepal_length = X(:,1);
% petal_length = X(:,3);
% petal_width = X(:,4);
% scatter3(sepal_length, petal_length, petal_width, 'MarkerEdgeColor', 'green');
% hold on;
%
% xlabel('sepal length (cm)');
% ylabel('petal length (cm)');
% zlabel('petal width (cm)');
% title('Iris species prediction using knn and linear classification');



% ------------------------------- U.I. ------------------------------------

% STEP 1: Choose Outer split: n/(150-n)
%        Divide data set into Training-Validation data set and Test data set
%        n input entries for the training set, 150-n for the Test set
%        150-n and n must both be divisible by 3 because data divided in 3
%        species sets of 50 Iris each.
%        Possible values: n = 15,30,45,60,75,90,105,120,135
n = input('Choose n for the outer split n/(150-n):  \n');
fprintf('\n ');

% n input entries for the Training-Validation set
nTrain = n/3;
A = X(1:nTrain,:);
B = X(51:(50+nTrain),:);
C = X(101:(100+nTrain),:);
D = Y(1:nTrain,:);
E = Y(51:(50+nTrain),:);
F = Y(101:(100+nTrain),:);
%Note that this data set will be randomly shuffled
%Training-Validation Set:
xTrainingMeas = [A;B;C];
yTrainingSpecies = [D;E;F];

%150-n unused input entries for the final test set
nTest = (150-n)/3;
I = X((50-nTest+1):50,:);
J = X((100-nTest+1):100,:);
K = X((150-nTest+1):150,:);
L = Y((50-nTest+1):50,:);
```

```
M = Y((100-nTest+1):100,:);
N = Y((150-nTest+1):150,:);
%This data set will NOT be shuffled. It will only be used once at the end,
%to test the model
%Test Set:
xTestMeas = [I;J;K];
yTestRealVal = [L;M;N];
%The string values of this iris species test set (setosa, versicolor and
%virginica) are converted to the values 1,2 and 3 respectively
yTestRealVal = convert_to_ID(yTestRealVal);

% STEP 2: Inner split: p/(n-p)
%       Make new training sets & validation sets in a randomly manner
%       for a given p/(n-p) split, where n was the value chosen for the
%       outer split and p is the value to be chosen for the inner split.
p = input('Choose p for the inner split  p/(n-p):  \n');
fprintf('\n ');

% STEP 3: choose number of training data folds
folds = input('How many times do you want to train your model? \n');
fprintf('\n ');

% STEP 4: pick your model
entry = input('Model: linear or knn?   ---> type 1 or 2 \n');
fprintf('\n ');
if entry == 2
  % STEP 4B: choose k
    k = input('Enter k:  \n');
    fprintf('\n ');
end
fprintf('\n ');

%-----------------------------------------------------------------------


% This for loop below is used for automation of the tests ONLY
% Please take out for better visualisation of the scatter plot
total_test_mean = 0;
total_validation_mean = 0;
for a = 1:10

  % --- Build different random sets for each fold ---
  % Transform the meas/species data split into a training/validation data split
  for i = 1:folds
    [trainSet,validSet] = make_training_sets(xTrainingMeas,yTrainingSpecies,n,p);
    % Training sets and Validation sets visualisation
  %     disp('training set'), disp('      X1        X2        X3        X4         Y');
  %     disp([trainSet(:,1),trainSet(:,2),trainSet(:,3),trainSet(:,4), trainSet(:,5)]);
  %
  %     disp('validation set'), disp('      X1        X2        X3        X4         Y');
  %     disp([validSet(:,1),validSet(:,2),validSet(:,3),validSet(:,4),validSet(:,5)]);
    % Chosen model
    if entry == 1
      model = fitcecoc(trainSet(:,1:4),trainSet(:,5));
    else
      model = fitcknn(trainSet(:,1:4),trainSet(:,5),'NumNeighbors',k);
    end
    % Prediction
    yPred = predict(model,validSet(:,1:4));
    % add all mean classification errors from each run
  %     disp('The mean error for fold No'),disp(i),disp(' is: '), disp(fold_error(yPred, validSet(:,5)));
    e = e + fold_error(yPred, validSet(:,5));
  end
```

```
    %Calculate the mean folds error of all runs
    e = e/folds;

    % --- Test data prediction and final results ---
    yPred = predict(model,xTestMeas);
    eTest = fold_error(yPred, yTestRealVal); % real model error on new data

    fprintf('\n Run %.4f % \n', a);
    fprintf('\n The average validation error (AVE) for the training-validation data, for this particular
model is: %.4f % \n', e);

    fprintf('\n The average test error (ATE) for the TEST data, for this particular model is: %.4f % \n',
eTest);
    fprintf('\n  ');

    total_test_mean = total_test_mean + eTest; %for automation of the tests only
    total_validation_mean = total_validation_mean + e; %for automation of the tests only
end

    %for automation of the tests only
    fprintf('\n  ');
    fprintf('\n The total_test_mean for all runs is: %.4f % \n', total_test_mean/10);
    fprintf('\n  ');

    fprintf('\n The total_validation_mean for all runs is: %.4f % \n', total_validation_mean/10);
    fprintf('\n  ');



% --- Final visualisation of the test prediction ---
% the sepal width has been taken out but ONLY for the visualisation
Xtest_sepal_length = xTestMeas(:,1);
Ytest_petal_length = xTestMeas(:,3);
Ztest_petal_width = xTestMeas(:,4);

scatter3(Xtest_sepal_length(yPred == 1), Ytest_petal_length(yPred == 1), ...
        Ztest_petal_width(yPred == 1), ...
        'MarkerEdgeColor', 'magenta','LineWidth',2);%predicted setosa iris points
hold on;
scatter3(Xtest_sepal_length(yPred == 2), Ytest_petal_length(yPred == 2), ...
        Ztest_petal_width(yPred == 2), ...
        'MarkerEdgeColor', 'blue','LineWidth',2);%predicted versicolor iris points
hold on;
scatter3(Xtest_sepal_length(yPred == 3), Ytest_petal_length(yPred == 3), ...
        Ztest_petal_width(yPred == 3), ...
        'MarkerEdgeColor', 'red','LineWidth',2);%predicted virginica iris points
hold on;
scatter3(Xtest_sepal_length(yPred == yTestRealVal), Ytest_petal_length(yPred == yTestRealVal), ...
        Ztest_petal_width(yPred == yTestRealVal), ...
        'MarkerEdgeColor', 'black','Marker','*','LineWidth',0.5);%correct predictions
hold on;
toc
```

best_k.m

```
function min_error = best_k()
%To determine the best possible k for the kNN model

tic
iris_data = load('fisheriris.mat');
X = iris_data.meas;
Y = iris_data.species;
%Mean folds error of ALL folds for one particular model
global e;
e = 0;

% STEP 1: Choose n for outer split like in Main
fprintf('\n ');

% n input entries for the Training-Validation set
nTrain = n/3;
A = X(1:nTrain,:);
B = X(51:(50+nTrain),:);
C = X(101:(100+nTrain),:);
D = Y(1:nTrain,:);
E = Y(51:(50+nTrain),:);
F = Y(101:(100+nTrain),:);
%Note that this data set will be randomly shuffled
%Training-Validation Set:
xTrainingMeas = [A;B;C];
yTrainingSpecies = [D;E;F];


% STEP 2: Choose p for inner split like in Main
p = input('Choose p for the inner split p/(n-p):  \n');
fprintf('\n ');


%1000 folds for each k, to find out which k value works best for kNN algorithm
arr_errors_k = zeros(25,1);
for k = 1:25
    for i = 1:1000
        [trainSet,validSet] = make_training_sets(xTrainingMeas,yTrainingSpecies,n,p);
        model = fitcknn(trainSet(:,1:4),trainSet(:,5),'NumNeighbors',k);
        yPred = predict(model,validSet(:,1:4));
        e = e + fold_error(yPred, validSet(:,5));
    end
    e = e/1000;
    arr_errors_k(k) = e;
    fprintf('\n k: %.4f % \n', k);
    fprintf('\n error: %.4f % \n', e);
    fprintf('\n  ');
end
[min_error,bestK] = min(arr_errors_k);

fprintf('\n The best possible k for this particular model is: %.4f % \n', bestK);
fprintf('\n  ');

fprintf('\n The minimum average validation error is: %.4f % \n', min_error);
fprintf('\n  ');

toc
end
```

fold_error.m

```
function e = fold_error(prediction, Y)
% Classification error function commonly used for linear classification
numY = height(Y);
global I;
I = 0;
for i = 1:numY
%     disp('prediction '),disp(i), disp('    '), disp(prediction(i)), ...
%         disp(' , real value: '),disp(Y(i));
    if prediction(i) ~= Y(i)
        I = I+1;
    end
e = I/numY;

end
end
```

——

——

make_training_sets.m

```
function [trainingSet,validationSet] = make_training_sets(meas,species,n,p)
%Return a training set and a validation set based on the input meas and species.

% The function merges the data from meas and species into one n by 5
% matrix. Then it shuffles all rows. Finally it divides the matrix into 2
% matrices:
%           - Training Set = p by 5 matrix
%           - Validation Set = n-p by 5 matrix


%Convert species to numerical IDs (1,2,3 respectively) before appending
%them to the meas matrix
type = convert_to_ID(species);

%Append the species data to the meas data so that we can divide the
%training-validation set into one training set and one valiadation set
X = [meas, type];

%randomly shuffle the whole iris data set for a better training and validation
X = X(randperm(size(X, 1)), :);

%p input entries for the Training Set
trainingSet = X(1:p,:);

%n-p input entries for the Validation Set
validationSet = X((p+1):n,:);


end
```

convert_to_ID.m

```
function [speciesID] = convert_to_ID(species)
%Convert species (setosa, versicolor & virginica) to IDs (1,2,3 respectively)
global type;
type = zeros(height(species),1);
for i = 1:height(species)
    if strcmp(species(i),'setosa')
        type(i) = 1;
    elseif strcmp(species(i),'versicolor')
        type(i) = 2;
    else
        type(i) = 3;
    end
end
speciesID = type;
end
```

Final Personal Note: I didn’t use the very handy randperm(length(X)) functionality that saves all shuffled positions (for every shuffle) and can be reused for different matrices because all testing was already performed when I found out about it.