Learning and Adaptivity Assignment II

Alexander Hagg, Adam Gaier *BRS University of Applied Sciences*
email: alexander.hagg@smail.inf.h-brs.de github: @alexander-hagg

April 25, 2014

Decision tree learning is used to learn concepts or other discrete-valued functions. It allows building a short tree where nodes represent a decision, based on the value of a certain parameter, which has to be defined for all data tuples.

6 Decision Tree Learning

6.1 Simple Prediction

1. Predicting animal type from all other attributes (except for the species' name). Two decision trees were built using all data from the zoo database. The next best attribute was chosen using two available methods: gini index and entropy. The gini index on a data set T is defined as gini(T) $:= 1 - \sum_i p_i^{21}$, where p_i is the relative frequency of class i. The entropy measure measures the impurity of sample S of training examples. As can be taken from the results in 1 and 2, they differ in the sense that using the entropy split criterion tends to be a bit more conservative and gini tends to pick and split larger classes faster.

2. Predicting animal type from all other attributes (including the species' name). As can be taken from the results in 3 and 4, including the name of the animal, which is unique for every animal, obviously does not help in classification.

3. Predicting, whether an animal is airborne, from all other attributes (including type, excluding name). Results can be taken from figure 5. The attributes "feathers", "domestic" and "fins" are the first three best splitters, according to the gini criterion, which leads to large categories being split earlier on in the process.

4. Predicting species' name from all other attributes. Figure 6 shows that some groups of animals cannot be distinguished, because they have the same attributes. This is shown by the leaves' gini value being greater than zero (the relative frequency of the members of the "class" is not 1).

6.2 Scalability

1. As we do not use the applet, we did not run into any problems performance wise, as we can choose not to visualize.

2. We were not able to visualize the decision trees for training sets large than 200 samples. This issue is known by the TA. Instead, we decided to display the number of nodes in the tree to get a grasp on the size of it. As can be taken from figure 8, the number of nodes is near linear to the percentage of total data used for training. The time taken is linear as well. The top figure in figure 8 shows the mean error rate on the test set, with the blue region representing the standard deviation of the error. We can see that the error mean goes down until approximately 70% of the data is used for training. After that, a plateau occurs and eventually the error rate (and especially its standard deviation) goes up again, which is probably caused by overfitting. Figure 7 shows the error rate mean and standard deviation for the individual letters.

[1] Taken from lecture notes from the "Knowledge Extraction and Machine Learning" course at the University of Porto, http://paginas.fe.up.pt/~ec/

6.3 Noise

Data preprocessing algorithm:

1. Sort and split data in 'letter' groups
    a) For all groups: get the 'mean' letter (using k-means, or by just taking the median)
    a) For all letters (and their supposed category label):
        i. The noise is defined as the euclidean distance to the mean of its category
        ii. If the distance is smaller than than a certain innerclass noise threshold, e.g. twice the standard deviation of the group, we call this innerclass noise. We assume the label is correct and the distance to the mean is caused by the variance within the group.
        iii. If the distance is greater than a certain outerclass noise threshold, e.g. twice the standard deviation, we call this intraclass noise.

- Noise(group i) = The number of noisy instances / number of nonnoisy instances * 100%
- Noise is now controlled by picking a certain number of noisy instances and a number of nonnoisy instances. The intraclass noise can be separately controlled by instead of taking "the number of noisy instances", taking "number of noisy inner class instances + number of noisy intraclass instances"
- A refined noise quantization could weight the noisy instances by their actual distance to the mean.

Conclusion

The assignment definitely got improved by giving us the opportunity to use Python instead of a preprogrammed Java applet. But instead of taking two different databases, the assignment could be reduced to just the letter database, and instead increase the "in-depth" on e.g. overfitting and when this happens. Writing an algorithm to determine the noise was done by us, but there was absolutely no time left to actually implement and analyze its results.

Sources

Theory: http://paginas.fe.up.pt/~ec/
Code snippets: from our TA.

Appendix A: Code

code/zoo_ml.py

```
s_1_gini = SimplePredictionQuestion()
X = s_1_gini.data[:][:,0:15]
Y = s_1_gini.data[:][:,16]
s_1_gini.build_classifier(X, Y)
s_1_gini.draw_graph(s_1_gini.clf, '7-1-gini')

s_1_entropy = SimplePredictionQuestion()
X = s_1_entropy.data[:][:,0:15]
Y = s_1_entropy.data[:][:,16]
s_1_entropy.build_classifier(X, Y, 'entropy')
s_1_entropy.draw_graph(s_1_entropy.clf, '7-1-entropy')
```

code/letters_ml.py

```
while True:
    i+=1
    samples = i*100
    s_2_1 = ScalabilityQuestion()
    start_time = time.time()
    training_X = s_2_1.data[0:samples][:,0:15]
    training_Y = s_2_1.encoded_letters[0:samples][:]

    s_2_1.build_classifier(training_X, training_Y)
    classifier_time = (time.time() - start_time)

    threshold = s_2_1.clf.tree_.threshold

    test_X = s_2_1.data[samples:][:,0:15]
    test_Y = s_2_1.encoded_letters[samples:][:]

    #don't include visualization in classifier performance
    #s_2_1.draw_graph(s_2_1.clf, '8_'+str(samples))

    #with open('./timefile', 'a') as f:
        #f.write(str(i*100) + ' samples in ' + str(time.time() - start_time) + ' sec
        #f.write(str(samples) + ',' + str(classifier_time) + '\n')

    with open('./treesize_file', 'a') as f:
        f.write(str(samples) + ',' + str(len(threshold)) + '\n')

    count = 0
    letter_perf = np.zeros((26,2))
    for letter in range(len(test_X)):
        prediction = s_2_1.clf.predict(test_X[letter])
        actual = test_Y[letter]
        if actual == prediction:
            letter_perf[actual,0] += 1
        else:
```

```
            letter_perf[actual,1] += 1
        count +=1


    lfname = ('./letterdata' + str(samples))
    #with open(lfname, 'w') as lf:
        #lf.write(str(letter_perf))
        #lf.close()

    #s_2_1.draw_graph(s_2_1.clf, lfname)

    if classifier_time > 60*mins:
        break
    print count
```

Visualization and data analysis in MATLAB:

code/letter_data.m

```
% Import data
for i=1:198 %last column (199) does not have enough samples
    rmBrackets = 'sed ''s/\[//g;s/\]//g'' letterdata';
    system([rmBrackets, [int2str(i*100) ' >> tmp']])
    raw = importdata('tmp');
    error(:,i) = raw(:,2)./(raw(:,1)+raw(:,2))
    system('rm tmp');
end

% Plot Results
figure(1);clf;
    subplot(3,1,1)
        hold on
        %TODO: Shade between lines rather than dotted lines
        x = linspace(1,198,198);
        mean_error = mean(error);
        mean_plus_std = mean(error)+std(error);
        mean_minus_std = mean(error)-std(error);
        fill( [x fliplr(x)], [mean_plus_std fliplr(mean_minus_std)], 'b');
        alpha(.05);
        plot(x, mean_error, 'k', 'LineWidth', 2)
        plot(x, mean_plus_std, 'b')
        legend('StdDev','Mean')
        plot(x, mean_minus_std, 'b')

        title('Error', 'FontSize', 14)
        set(gca,'XTickLabel',{'0%', '10%', '20%', '30%', '40%', '50%'...
                              , '60%', '70%', '80%', '90%', '100%'});
        xlabel('Percent of Total Data used for Training')
        ylabel('Error on Test Set')


    subplot(3,1,2)
        time = importdata('timefile');
        plot(x,time([1:198],2));
```

```
36          title('Training Time', 'FontSize', 14)
37          set(gca,'XTickLabel',{'0%', '10%', '20%', '30%', '40%', '50%'...
38                                , '60%', '70%', '80%', '90%', '100%'});
39          xlabel('Percent of Total Data used for Training')
40          ylabel('Seconds')
41
42      subplot(3,1,3)
43          hold on
44          tree_size = importdata('treesize');
45          plot(x,tree_size([1:198],2));
46          title('Tree Size', 'FontSize', 14)
47          set(gca,'XTickLabel',{'0%', '10%', '20%', '30%', '40%', '50%'...
48                                , '60%', '70%', '80%', '90%', '100%'});
49          xlabel('Percent of Total Data used for Training')
50          ylabel('Nodes')
51
52
53  figure(2)
54      meanError = mean(error(:,[40:end]),2);
55      bar(meanError)
56      set(gca,'Xtick',[1:26])
57      set(gca,'XTickLabel',{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'...
58                          , 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't'...
59                          , 'u', 'v', 'w', 'x', 'y', 'z'});
60      hold on;
61      h=errorbar([1:26], meanError, std(error(:,[40:end]),1,2));
62      set(h,'linestyle','none', 'color', 'r');
63      axis([0 27 0 0.30])
64      title('Mean Letter Error Rates (20% to 95% Training mix)', 'FontSize', 14)
```

Appendix B: Results

Figure 1: *Decision tree predicting animal type*

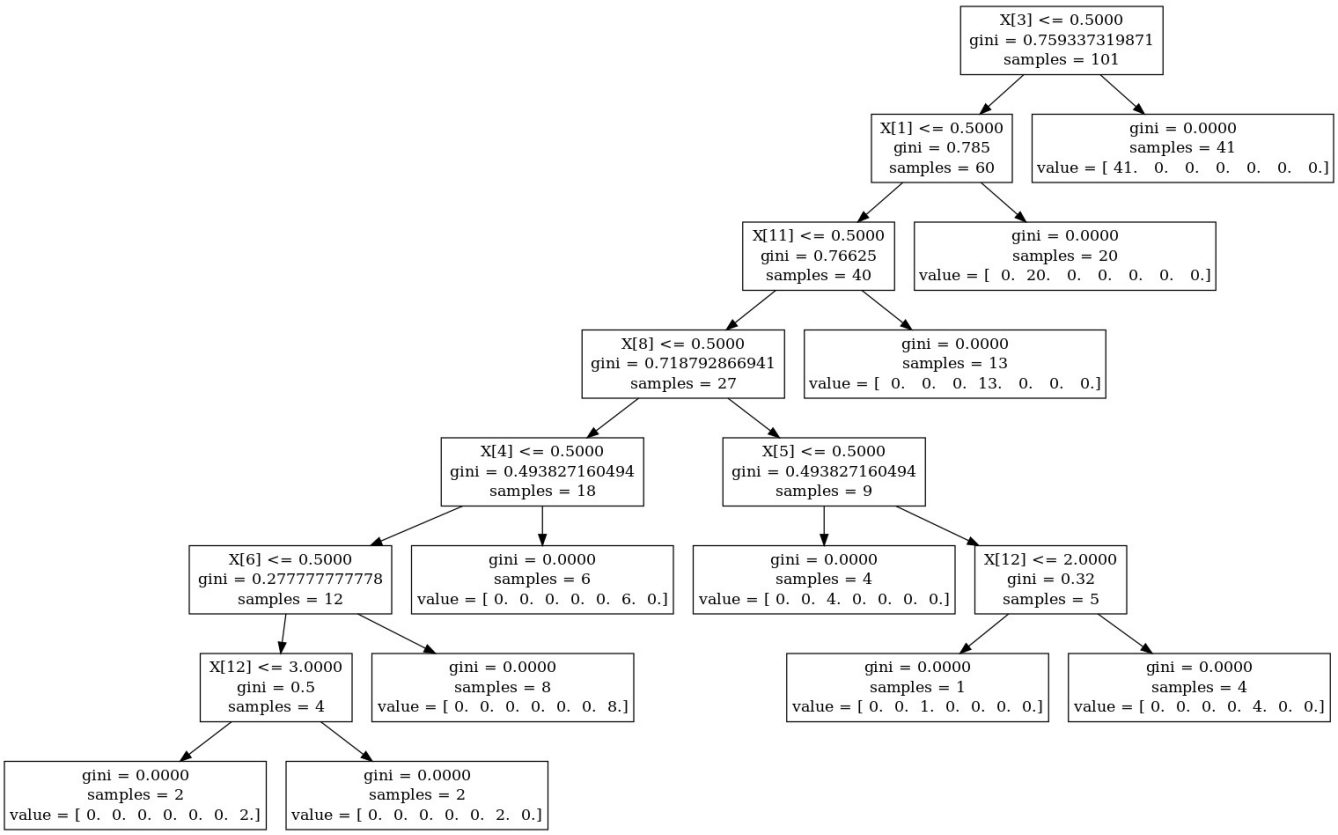

Figure 2: *Decision tree predicting animal type*

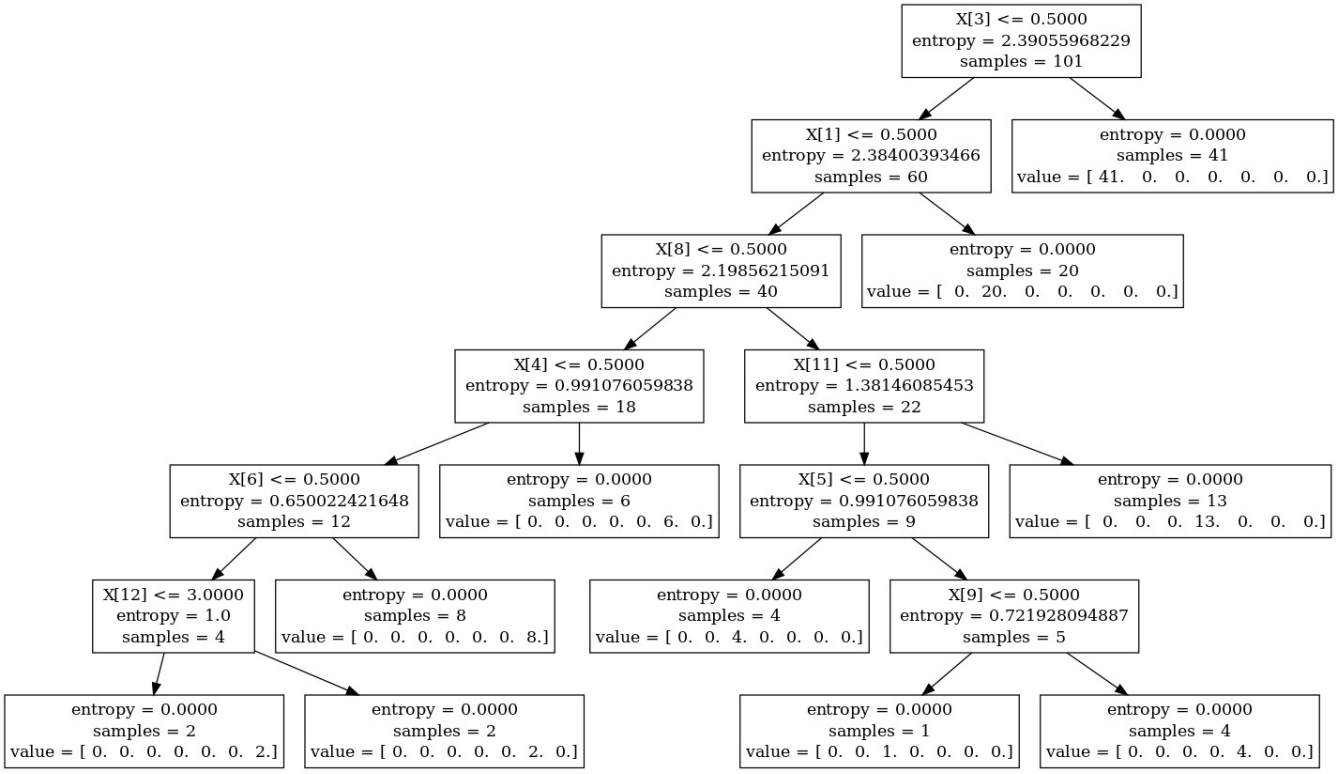

Figure 3: *Decision tree predicting animal type incl. name*

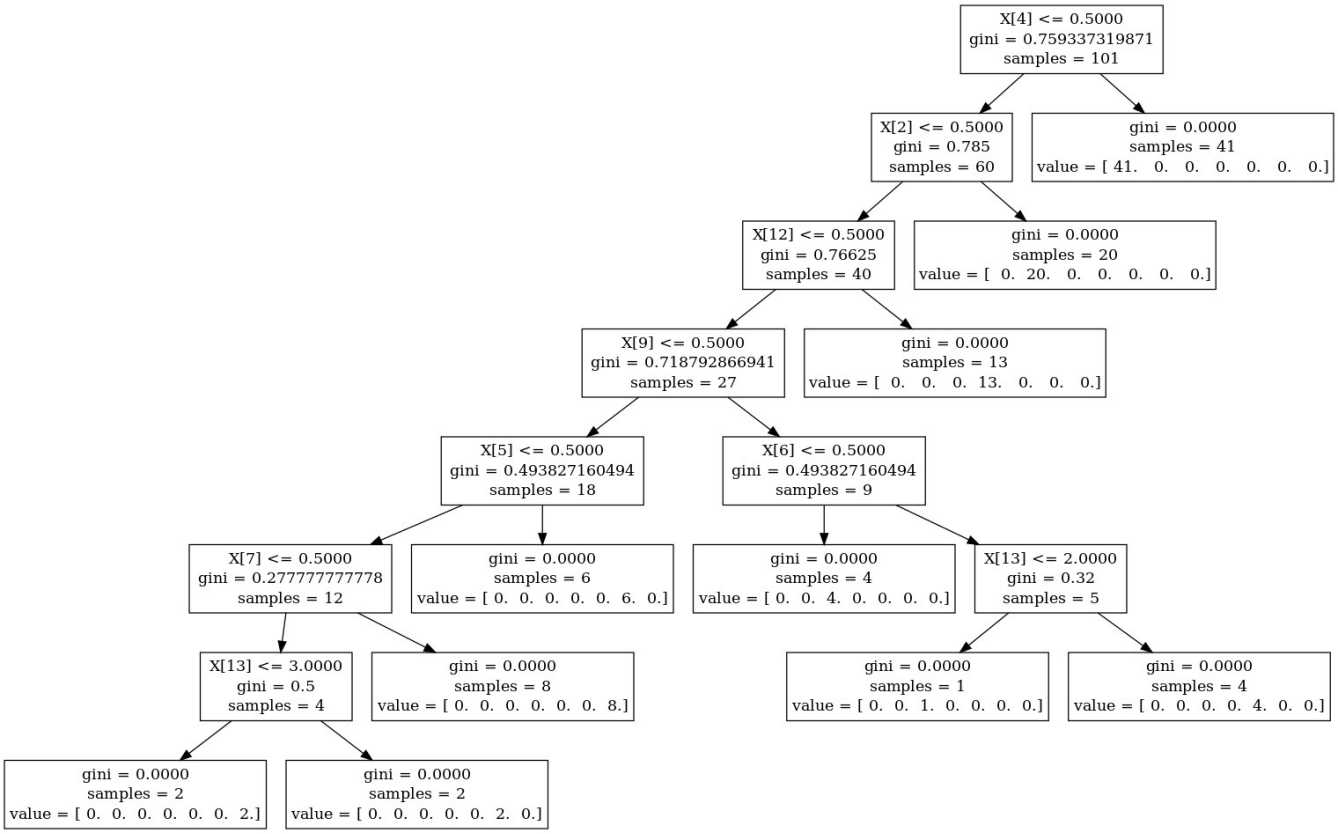

Figure 4: *Decision tree predicting animal type incl. name*

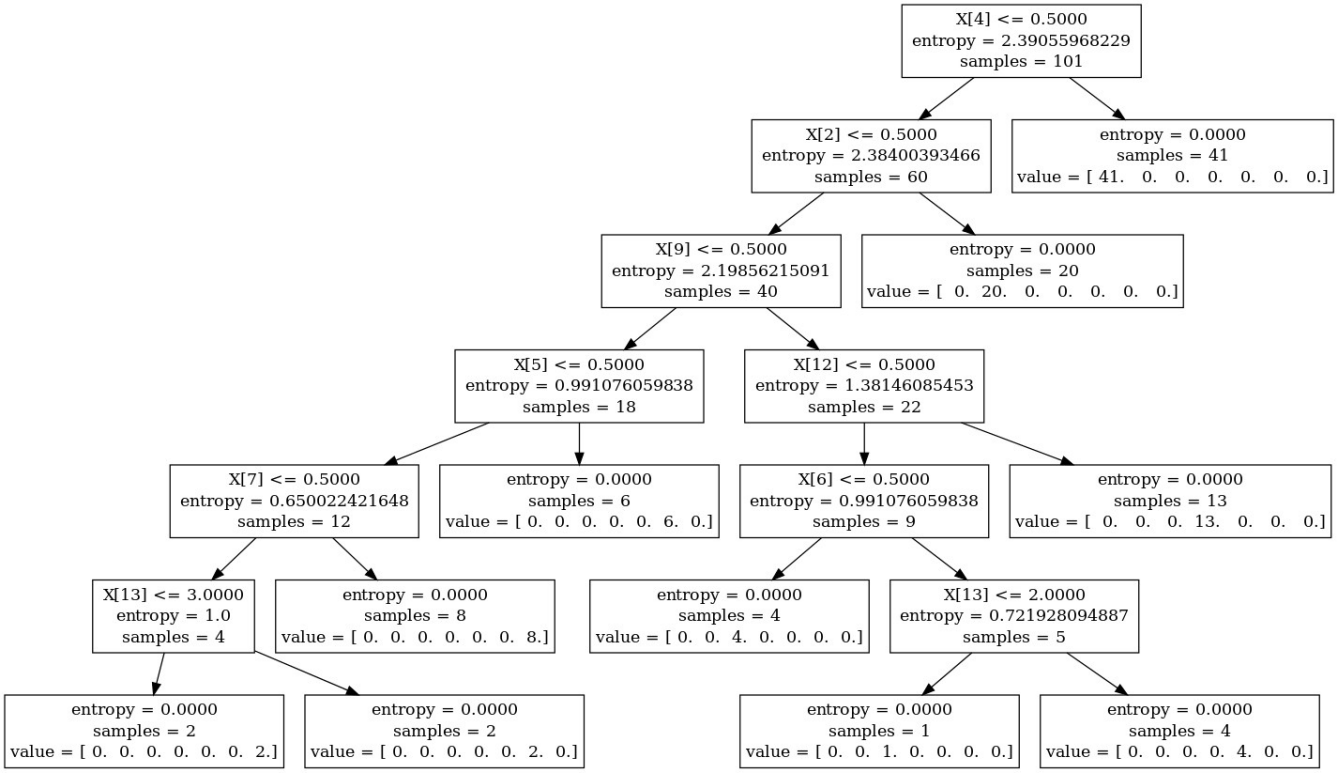

Figure 5: *Decision tree predicting whether animal is airborne*

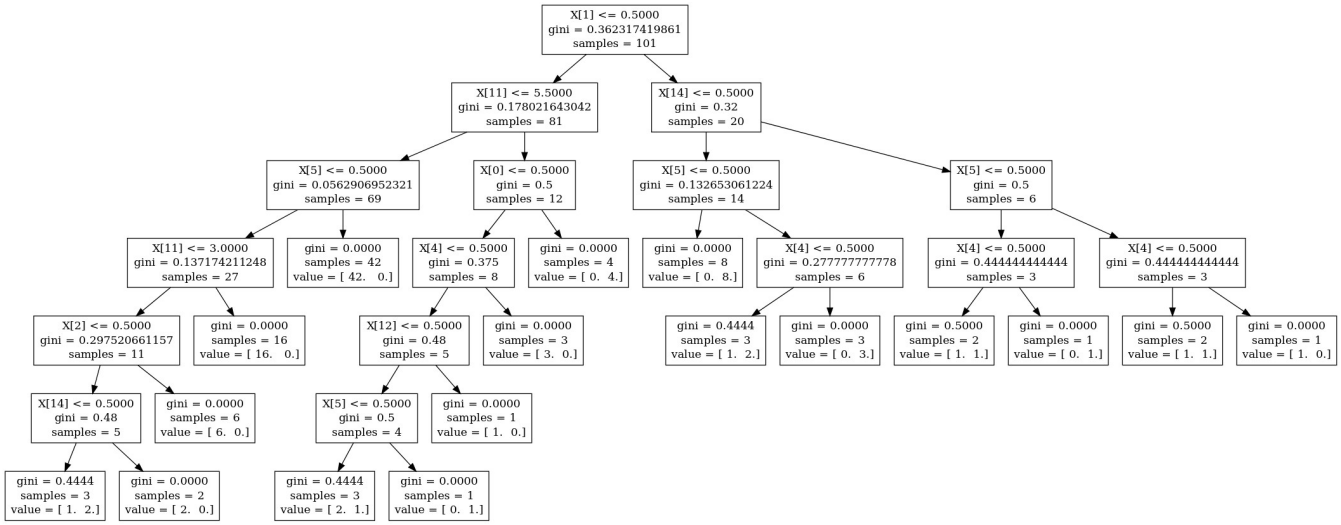

Figure 6: *Decision tree predicting animal name*

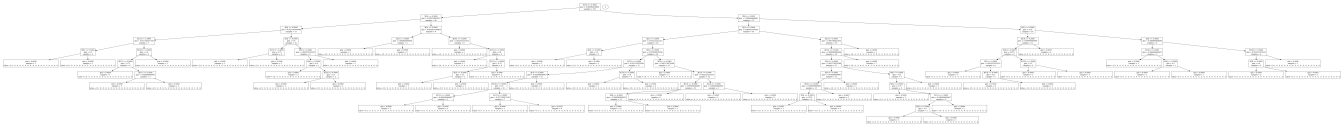

Figure 7: *Decision tree predicting letters: mean letter error rates*

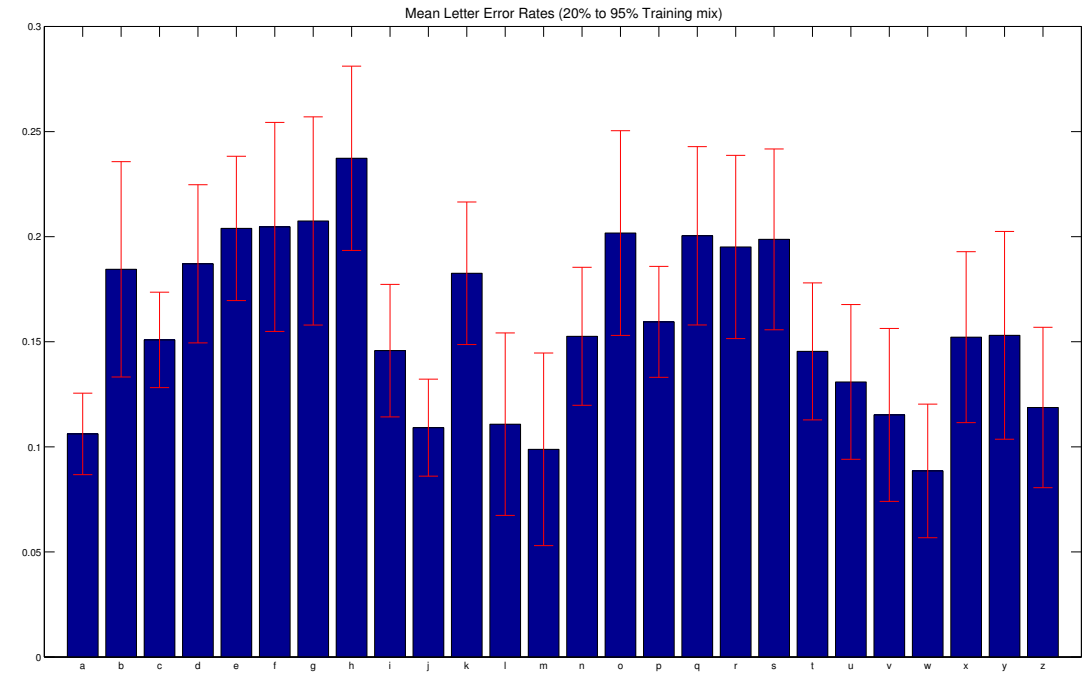

Figure 8: *Performance on letters*