Assignment 2

Tomas Langebaek Carrizosa. Group members: Richmond Horikawa, Nathan Hu, Chenxiang Zhang, Amalia Riegelhuth

1. Text Book Exercises

1.1 PRML 8.16

Consider the problem of finding

$$p(x_n \mid x_N)$$

for the nodes in Figure 8.38. Then the message parsing algorithm of section 8.4.1 allow us to get the following results with marginalization over the variables

$$p(x_n = \sum_{x_1} ... \sum_{n-1} \sum_{n} p(x)$$

For one computation we have that

$$p(x) = \sum p(x, \hat{x})$$

So, if we want to get the conditional probability for each message we know that this is true:

$$p(x_n \mid x_N) = p(x_n, x_N)/p(x_n)$$

So, if the message is passed throw the chain, we can compute the conditional probability using the marginalization over the other variables and the join distribution, that come from the previous message. Therefore, the parsing algorithms solves the problem efficiently.

1.2 PRML 8.27

Table 1: Table (a) provides the join probability respecting the constraints of the variables for this exercise.

(a)

–	x1	x2	x3	–
y1	0	0.1	0.3	0.4
y2	0.1	0.1	0.05	0.25
y3	0.3	0.05	0	0.35
–	0.4	0.25	0.35	1

1.3 ESL 8.4

Consider the bagging method of Section 8.7:

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} f^{*b}(x)$$

Let our estimate $f(x)$ be the B-spline smoother $\mu(x)$ of Section 8.2.1:

$$\hat{\mu}^* = h(x)^T(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{y}^*$$

Consider the parametric bootstrap of equation (8.6):

$$\mathbf{y}_i^* = \hat{f}(x) + \epsilon_i^*$$

applied to this estimator. Show that if we bag f(x), using the parametric bootstrap to generate the bootstrap samples,the bagging estimate $\hat{f}_{bag}(x)$ converges to the original estimate $f(x)$ as $B \to \infty$.

$$\text{Let } h(x)^T(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T = z \text{ , so } \hat{\mu}^* = z\mathbf{y}^* = \hat{f}(x)$$

$$\text{because } \mathbf{y}_i^* = \hat{f}(x) + \epsilon_i^* \text{ we get}$$

$$z(\hat{f}(x) + \epsilon_i^*)$$

$$= z(zy + \epsilon_i^*$$

$z^2y + z\epsilon_i^*$ and assuming that the function z squared behaves like z:

$$zy + z\epsilon_i^*$$

$$\lim_{B\to\infty} \frac{1}{B}\sum_{b=1}^{B} f^{*b}(x) + \epsilon_i^*$$

$$= \lim_{B\to\infty} \frac{1}{B}\sum_{b=1}^{B} f^{*b}(x) + \lim_{B\to\infty} \frac{1}{B}\sum_{b=1}^{B} \epsilon_i^*$$

$$= \lim_{B\to\infty} \hat{f}(x) + \lim_{B\to\infty} 0$$

$$= \hat{f}(x)$$

1.4 ESL 10.1

Derive expression (10.12) for the update parameter in AdaBoost.

$$(\epsilon^{\beta} - \epsilon^{-\beta})\sum_{i=1}^{N} \omega^{(m)} I(y_i \neq G(x_i)) + \epsilon^{-\beta} \cdot \sum_{i=1}^{N} \omega^{(m)}$$

$$\text{let } K_1 = \sum_{i=1}^{N} \omega^{(m)}, \; K_2 = I(y_i \neq G(x_i))$$

so we have that

$$(\epsilon^{\beta} - \epsilon^{-\beta}) \sum_{i=1}^{N} \omega^{(m)} I(y_i \neq G(x_i)) + \epsilon^{-\beta} \cdot \sum_{i=1}^{N} \omega^{(m)} = (\epsilon^{\beta} - \epsilon^{-\beta}) K_1 K_2 + \epsilon^{-\beta} K_1 = f(x)$$

$$\frac{df}{d\beta} = (\epsilon^{\beta} + \epsilon^{-\beta}) K_1 K_2 - \epsilon^{-\beta} K_1 = 0$$

$$= (K_1 K_2 \epsilon^{\beta} + K_1 K_2 \epsilon^{-\beta} - K_1 \epsilon^{-\beta}) \cdot \epsilon^{\beta} = 0 \cdot \epsilon^{\beta}$$

$$= K_1 K_2 \cdot \epsilon^{2\beta} + K_1 K_2 - K_1 = 0$$

$$\epsilon^{2\beta} = \frac{K_1 - K_1 K_2}{K_1 K_2}$$

$$\epsilon^{2\beta} = \frac{\frac{K_1 - K_1 K_2}{K_1}}{\frac{K_1 K_2}{K_1}}$$

$$\epsilon^{2\beta} = \frac{1 - \frac{K_1 K_2}{K_1}}{\frac{K_1 K_2}{K_1}}, \ \frac{K_1 K_2}{K_1} = err$$

$$\epsilon^{2\beta} = \frac{1 - err}{err}$$

$$\beta = \frac{1}{2} \cdot \ln \frac{1 - err}{err}$$

2. Graphical Models

2.1

What is p(Intelligence = 1; Letter = 1; SAT = 1)?

$$p(I = 1; L = 1; S = 1) = \frac{p(I = 1 \cap L = 1 \cap S = 1)}{p(L = 1 \cap S = 1)}$$

$$p(L = 1 \mid G) p(G \mid D, I = 1) p(D) p(S = 1 \mid I = 1) p(I = 1):$$

Marginalization:

$$\sum_{G,D} I = 1, L = 1, S = 1, G, D =$$

$$0,9 \cdot 0,9 \cdot 0,6 \cdot 0,8 \cdot 0,3 = 0,117$$

$$+0,6 \cdot 0,08 \cdot 0,6 \cdot 0,8 \cdot 0,3 = 0,007$$

$$+0,01 \cdot 0,02 \cdot 0,6 \cdot 0,8 \cdot 0,3 = 0,0000288$$

$$+0,9 \cdot 0,5 \cdot 0,4 \cdot 0,8 \cdot 0,3 = 0,043$$

$$+0,6 \cdot 0,3 \cdot 0,4 \cdot 0,8 \cdot 0,3 = 0,017$$

$$+0,01 \cdot 0,2 \cdot 0,4 \cdot 0,8 \cdot 0,3 = 0,000192$$

$$\sum = 0,184$$

$$\sum_{G,D,I} L = 1, S = 1, G, D, I =$$

$$+0,9 \cdot 0,3 \cdot 0,6 \cdot 0,05 \cdot 0,7 = 0,00567$$

$$+0,6 \cdot 0,4 \cdot 0,6 \cdot 0,05 \cdot 0,7 = 0,00504$$

$$+0,01 \cdot 0,3 \cdot 0,6 \cdot 0,05 \cdot 0,7 = 0,000063$$

$$+0,9 \cdot 0,9 \cdot 0,6 \cdot 0,8 \cdot 0,3 = 0,11664$$

$$+0,6 \cdot 0,08 \cdot 0,6 \cdot 0,8 \cdot 0,3 = 0,006912$$

$$+0,01 \cdot 0,02 \cdot 0,6 \cdot 0,8 \cdot 0,3 = 0,0000288$$

$$+0,9 \cdot 0,05 \cdot 0,4 \cdot 0,05 \cdot 0,7 = 0,00063$$

$$+0,6 \cdot 0,25 \cdot 0,4 \cdot 0,05 \cdot 0,7 = 0,0021$$

$$+0,01 \cdot 0,7 \cdot 0,4 \cdot 0,05 \cdot 0,7 = 0,000098$$

$$+0,9 \cdot 0,5 \cdot 0,4 \cdot 0,8 \cdot 0,3 = 0,0432$$

$$+0,6 \cdot 0,3 \cdot 0,4 \cdot 0,8 \cdot 0,3 = 0,01728$$

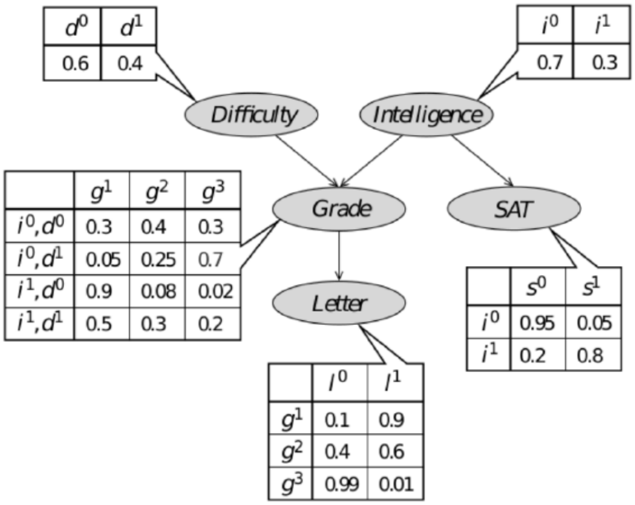
$$+0,01 \cdot 0,2 \cdot 0,4 \cdot 0,8 \cdot 0,3 = 0,000192$$

$$\sum = 0,197$$

$$\frac{p(I = 1 \cap L = 1 \cap S = 1)}{p(L = 1 \cap S = 1)} = \frac{0,1842}{0,197} = 0,931$$

2.2

Convert the directed graph in Figure 1 to a factor graph.



d^0	d^1
0.6	0.4

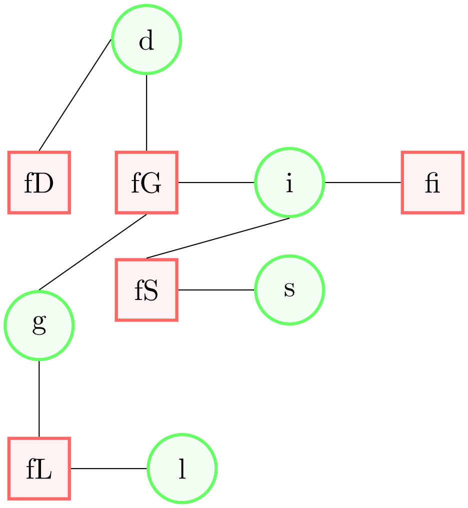
i^0	i^1
0.7	0.3

	g^1	g^2	g^3
i^0,d^0	0.3	0.4	0.3
i^0,d^1	0.05	0.25	0.7
i^1,d^0	0.9	0.08	0.02
i^1,d^1	0.5	0.3	0.2

	s^0	s^1
i^0	0.95	0.05
i^1	0.2	0.8

	l^0	l^1
g^1	0.1	0.9
g^2	0.4	0.6
g^3	0.99	0.01

Figure 1: Student network graph



Student network factor graph

2.3

For this part (Sum product and max sum Algorithms), I didn't mange to show the tables correctly, so they are far down the document, mixed with the 3rd exercises. For the sum-product algorithm, only the backwards step for finding the marginal probability of S is shown.

$\mu fd \rightarrow d(d):$

d	$f(d)$
0	0.6
1	0.4

$\mu d \rightarrow f(g):$

d	$f(d)$
0	0.6
1	0.4

$\mu fi \rightarrow i(I):$

d	$f(d)$
0	0.7
1	0.3

$\mu s \rightarrow f(s):$

		s	
		0	1
i	0	0.95	0.05
	1	0.2	0.8

$\mu fs \rightarrow i(I):$

		s	
		0	1
i	0	0.95	0.05
	1	0.2	0.8

$\mu i \rightarrow f(g):$

d	$f(d)$
0	0.7
1	0.3

$\mu f_g \rightarrow g(G) : \sum_d \sum_i f_g(i,d,g)\mu d \rightarrow f_g(d)\mu_i \rightarrow f_g(I):$

		s	
		0	1
i	0	0.95	0.05
	1	0.2	0.8

d	i	
0	0	0.6·0.7 → 0.42
1	0	0.6·0.3 → 0.18
1	0	0.4·0.4 → 0.28
1	1	0.4·0.3 → 0.12

d	i	g=1	g=2	g=3
0	0	0.3·0.42	0.4·0.42	0.3·0.42
1	0	0.9·0.18	0.08·0.18	0.02·0.18
1	0	0.05·0.28	0.25·0.28	0.7·0.28
1	1	0.5·0.12	0.3·0.12	0.2·0.12

	g=1	g=2	g=3
→	0.126	0.168	0.126
→	0.162	0.0144	0.0036
→	0.014	0.07	0.196
→	0.06	0.036	0.024

g	$p(g)$
1	0.362
2	0.2884
3	0.3496

$$\mu_g \rightarrow \mu_f l : \sum_g \sum_f f_g(l,g) f_g(L) :$$

g	$p(g)$
1	0.362
2	0.2884
3	0.3496

g	l	
1	0	0.362·0.1 → 0.0362
2	0	0.2884·0.4 → 0.11536
3	0	0.3496·0.99 → 0.246104
1	1	0.362·0.9 → 0.3258
2	1	0.2884·0.6 → 0.17304
3	1	0.3496·0.01 → 0.003496

l	$p(l)$
0	0.497664
1	0.502336

$$\mu_i \rightarrow f(g) : \sum_i \sum_s f_s(s,i) f_i(I) \rightarrow$$

i	s	
0	0	0.7·0.95 → 0.665
1	0	0.7·0.05 → 0.035
1	0	0.3·0.2 → 0.06
1	1	0.3·0.8 → 0.24

p(s=0) = 0.665+0.06=0.725

p(s=1) = 0.035+0.024=0.275

s	$p(s)$
0	0.725
1	0.275

3. Combining multiple learners

The goal of this exercise is to predict whether an email is spam or not using various ways of combining multiple learners. For this you will need the spam dataset described in ESL, which can be downloaded from https://archive. ics.uci.edu/ml/datasets/spambase. We will use classification trees as the base learner.

3.1

The data was divided in the following way: For the simple classification tree and bagging ratios of 25% for testing and validation were used. The other 50% was used for training. This are the recommended ratios according to the slides of the first lecture of the course. For the boosting algorithms the ratios were different. Due to the fact that the selected algorithm was AdaBoost, the validation, training and testing data had to be of the same size for the implementation, therefore ratios of 25%, 25%, and 25% of the total data were used for all training, validation, and testing sets. This is the code used for splitting the data:

```
indices = list(range(0,len(data)))
testAndValidation, trainingData = train_test_split(indices, test_size = 0.5)
halfTrainingData, otherHalf = train_test_split(trainingData, test_size = 0.5)
testData, validationData = train_test_split(testAndValidation, test_size =
    0.5)
```

Then X and Y for train, validation and test were taken in each case.

3.2

Implement, train and compare results of classifying spam

3.2.1 Classification tree method

For this implementation the scikit-learn tree classifier was used. The validation and training sets were used to find the correct hyper parameter (tree depth) for this data. Then, a tree classifier alone was used to get the final result with the testing data. The code and procedure used when finding the optimal tree depth is the following:

```
# loop with 27 iterations using validation data for finding the best tree
    depth.
for i in range(1, 27, 1):
    # the tree is initialized
    classifier = tree.DecisionTreeClassifier(random_state=0, max_depth=i)
    # the tree is trained
    classifier = classifier.fit(Xtrain, Ytrain)
    # a prediction is made with the validation set
    tree_prediction = classifier.predict(Xvalidation)
    # the score for this prediction is calculated
    score = accuracy_score(tree_prediction, Yvalidation)
    # the maximun score and its tree depth stored
    if score > maxAcc:
        maxAcc = score
        maxDepth = i
    treeDepth.append(i)
    treeAccuracy.append(score)
```

The final result of this step is shown in the following graph:

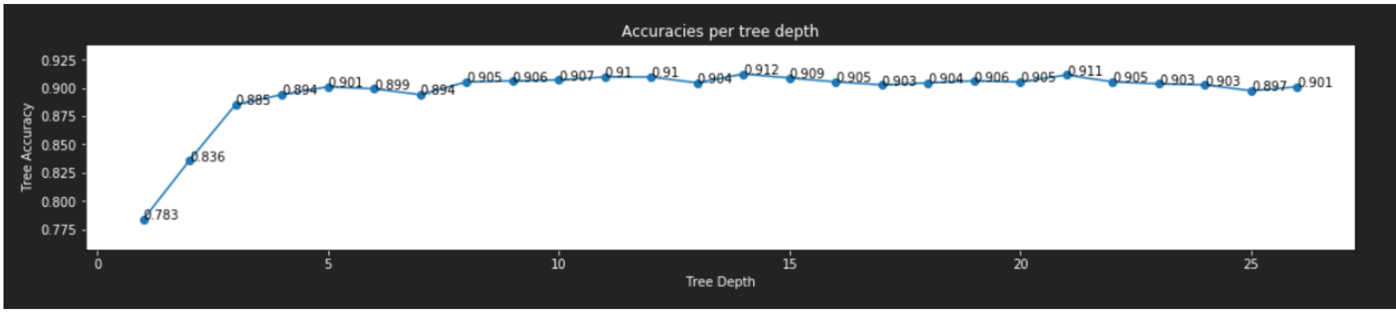


Figure 2: Accuracies per tree depth fora simple tree classifier

The final result for validation is shown bellow: Numbers were highlighted in purple.

```
Maximum Accuracy: '0.912'
For depth: '14'
```

With this parameter for tree depth, another simple tree was used. It was trained and tested with the testing data. The code used and its result was the following:

```
classifier = tree.DecisionTreeClassifier(random_state=0, max_depth=14)
classifier = classifier.fit(Xtrain, Ytrain)
tree_prediction = classifier.predict(Xtest)
score = accuracy_score(tree_prediction, Ytest)
print('Accuracy:',score)
```

```
Accuracy: '0.906'
```

Which is consistent with the validation.

3.2.2 Bagging

For bagging the same scikit-learn tree classifier was used. Samples where taken from the training data repeatedly and a consensus of the predictions was made. The same validation data was used in each iteration to get consistency in the testing. To find the hyper parameters this was done with the validation set, for different iterations. The selection for the size of samples was 63%. Literature corresponding Bagging says that this is a good sample size in average.

```
sampleSize = int(Xvalidation.shape[0]*0.63)
```

This is the procedure used for validation:

```
# for loop for validation
for maxbagging in range(0,100):
    totalResults = np.array(np.full(Xvalidation.shape[0], 0))
    ValidationRows = np.random.randint(Xvalidation.shape[0], size=Xvalidation.shape[0])
    XsampleValidate = Xvalidation[ValidationRows,:]
    YsampleValidate = Yvalidation[ValidationRows]
    # Bagging as done with multiple repetitions
    for i in range(maxbagging):
        # Here a random sample from the training data is taken
        randomRowsTraining = np.random.randint(Xtrain.shape[0], size=sampleSize)
        XsampleTrain = Xtrain[randomRowsTraining,:]
        YsampleTrain = Ytrain[randomRowsTraining]
        # As before, a simple decision tree was used
        classifierBagging = tree.DecisionTreeClassifier(random_state=1, max_depth=14)
        classifierBagging = classifierBagging.fit(XsampleTrain, YsampleTrain)
        tree_prediction = classifierBagging.predict(XsampleValidate)
        auxMatrix = [totalResults, np.array(tree_prediction)]
        # The results for each sample is added to the total and then, the consensus is taken
        totalResults = np.sum(auxMatrix, axis=0)

    totalResults = np.divide(totalResults, maxbagging)
    # This is the way in which the consensus was calculated

    for i in range(len(totalResults)):

        if(totalResults[i]<0.5):
            totalResults[i] = 0
        else:
            totalResults[i] = 1

    score = accuracy_score(totalResults, YsampleValidate)
```

The result is the following:

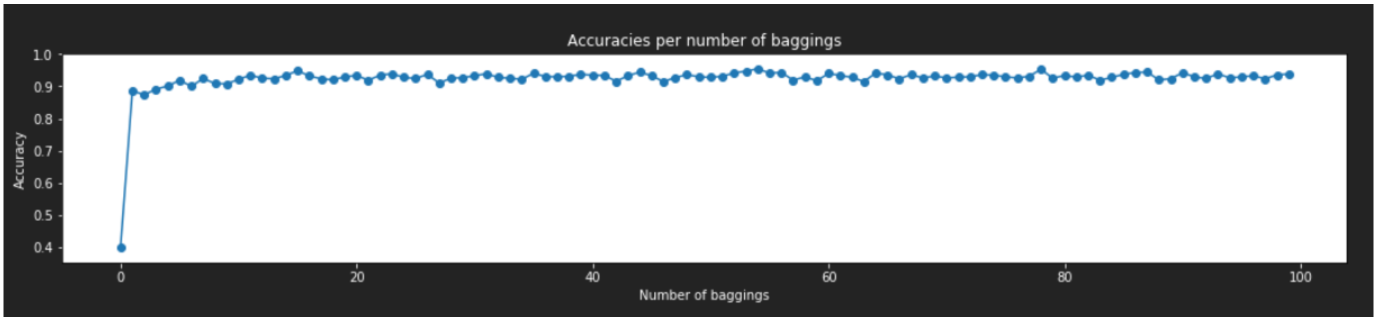


Figure 3: Validation: Accuracies per bagging number

```
Maximum Accuracy: '0.955'
For number of bags: '54'
Sample size (63% size of data): 724
```

This is the result of the testing data set with the fixed hyper parameter. The code is the same as before, without the external loop.

```
Accuracy: '0.935'
```

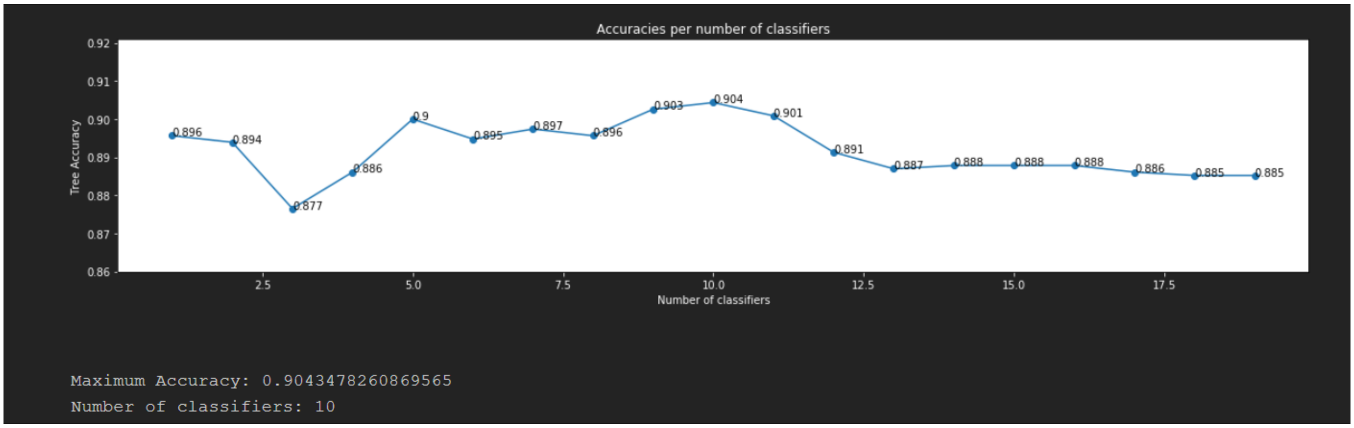
This is also consistent with the validation.

3.2.3 Boosting algorithm - AdaBoost.M1

The boosting alogorith implemented is this part is AdaBoost.M1. For this implementation, different values of iteration were tested. The tree depth, the maximun for the excercise 3.2.1 was used. The revelenta code of the implementatios is the following:

```
treeDepth = 1
NClassifiers = []
treeAccuracy = []
maxClassifier = 0;
maxAcc = 0;

trainSize = Xtrain2.shape[0]
validationSize = Xvalidation.shape[0]
classifierAda = tree.DecisionTreeClassifier(random_state=0, max_depth=14)
for k in range(1, 20):

    addedClassifiers = np.full((validationSize), 0.0)
    Weights = np.full(trainSize, 1/trainSize)
    n = 0
    for observation in range(k):
        n += 1
        classifierAda.fit(Xtrain2, Ytrain2, sample_weight = Weights)
        prediction = classifierAda.predict(Xvalidation)
        incorrect = (prediction != Yvalidation)
        incorrectWeightsSum = 0;
        for i in range(len(incorrect)):

            if np.take(incorrect, i):
                incorrectWeightsSum += Weights[i]
        error = 1-accuracy_score(prediction,Yvalidation)
        alpha = np.log( (1 - error) / error)
        for j in range(len(incorrect)):
            if np.take(incorrect, j):
                Weights[j] = Weights[j]*np.exp(alpha)

        prediction = np.multiply(prediction, alpha)
        addedClassifiers = [sum(x) for x in zip(prediction, addedClassifiers)]

    addedClassifiers = np.divide(addedClassifiers, n)

    for i in range(len(addedClassifiers)):

        if(addedClassifiers[i]<0.5):
            addedClassifiers[i] = 0
        else:
            addedClassifiers[i] = 1

    score = accuracy_score(addedClassifiers, Yvalidation)

    if score > maxAcc:
        maxAcc = score
        maxClassifier = k
    NClassifiers.append(k)
    treeAccuracy.append(score)
```

In this way, the optimal hyper parameter was found. Then the testing was done with the testing data. These are the results for hyper parameters and testing:



```
Maximum Accuracy: 0.9043478260869565
Number of classifiers: 10
```

Figure 4: Validation for Ada Boost

And for testing, with this parameter (10), the result is the following:

```
Accuracy: '0.935'
```

In this case the test showed better results than the validation.

3.2.4 Conclusion

When we used the simple tree classifier the accuracy was 0.912 for a tree depth of 14. This result (the hyper parameter) might be due to the fact that the data we used had multiple inputs for just one value of Y. In contrast, when the tree was bagged, better results were achieved. For example we got an accuracy of 0.955 for 54 repetitions. As it can be see in the graph of bagging, results stay high even if multiple bags are take. But overall they tend to get lower. This might be due to an over fit. Finally in the AdaBoost algorithms, for this particular implementation, we didn't get better results than in bagging. Multiple approaches we made in the calculation of the error and the final sum of all the previous classifiers. Even though this was a problem, an accuracy of 0.935 was achieved which is better than the classification tree alone, but not enough to be higher than the bagging approach. Overall every single one of the strategies used produced good results in their own way, and are a very good example of what kind of results can be achieved by combining multiple learners.