Banknote authentication through image features.

For the course MG-GY 9753 Machine Learning for Business.

Fall 2018

Professor H. Kone

Members: Aakriti Gupta, Darina Guseva, Santiago Novoa

Project Repository with code:
https://github.com/smn405/ML9753_Project

Problem Statement

- **Data Set Information:**
  - Data were extracted from images that were taken from genuine and forged banknote-like specimens. For digitization, an industrial camera usually used for print inspection was used. The final images have 400 x 400 pixels. Wavelet Transform tool were used to extract features from images.
- **Attribute Information:**
  - 1. Variance of Wavelet Transformed image (continuous)
    2. Skewness of Wavelet Transformed image (continuous)
    3. Curtosis of Wavelet Transformed image (continuous)
    4. Entropy of image (continuous)
    5. Class The true label for whether or not the banknote is forged (Yes = 1, No = 0).

Our goal is to build and test different classifiers predicting whether a banknote is real.

All implementations are built from scratch without using Scikit-learn Library.

Data Set Information

Number of Instances: 1372

Number of Attributes: 5

Missing Values: 0

Associated Tasks: Classification

Source:

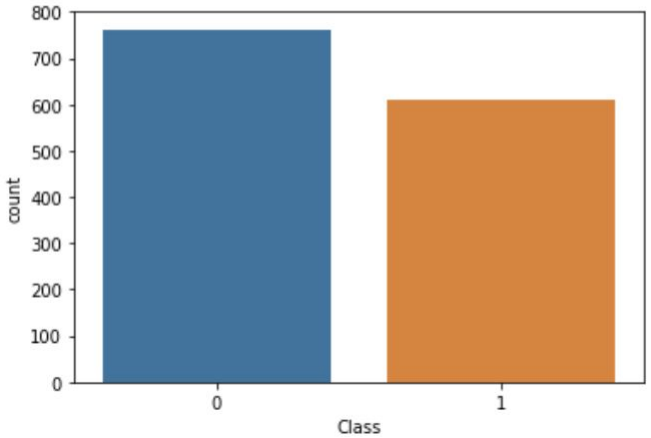
Owner of data: Volker Lohweg, University of Applied Sciences, Ostwestfalen-Lippe.
Donor of data: Helene Darksen, University of Applied Sciences, Ostwestfalen-Lippe.

Date Published: 2013-04-16

Target Variable exploration

	Variance	Skewness	Curtosis	Entropy	Class
0	3.62160	8.6661	-2.8073	-0.44699	0
1	4.54590	8.1674	-2.4586	-1.46210	0
2	3.86600	-2.6383	1.9242	0.10645	0
3	3.45660	9.5228	-4.0112	-3.59440	0
4	0.32924	-4.4552	4.5718	-0.98880	0

```
0    762
1    610
Name: Class, dtype: int64
```



KNN implementation

Min-max normalization is often known as feature scaling where the values of a numeric range are reduced to a scale between 0 and 1. The normalised value of a member of the set of observed values of *x*, is calculated using the following formula:

$$z = \frac{x \text{ - } \min(x)}{\max(x) \text{ - } \min(x)}$$

Where *min* and *max* are the minimum and maximum values in *x* given its range.

	Variance	Skewness	Curtosis	Entropy	Class
count	1372.000000	1372.000000	1372.000000	1372.000000	1372.000000
mean	0.539114	0.587301	0.287924	0.668917	0.444606
std	0.205003	0.219611	0.185669	0.191041	0.497103
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.379977	0.451451	0.159869	0.557821	0.000000
50%	0.543617	0.602168	0.254280	0.723929	0.000000
75%	0.711304	0.770363	0.364674	0.813171	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000

Scikit- Comparison

```
from sklearn import metrics
print('The accuracy of KNN from scratch: ',metrics.accuracy_score(test['Class'], test['Predictions']))
print('The accuracy of KNN from Sci-kit: ',metrics.accuracy_score(test['Class'], test['scikit']))
```

```
The accuracy of KNN from scratch:  0.8220640569395018
The accuracy of KNN from Sci-kit:  1.0
```

```
metrics.confusion_matrix(test['Class'], test['Predictions'])
```

```
array([[143,  28],
       [ 22,  88]])
```

```
metrics.confusion_matrix(test['Class'], test['scikit'])
```

```
array([[171,   0],
       [  0, 110]])
```

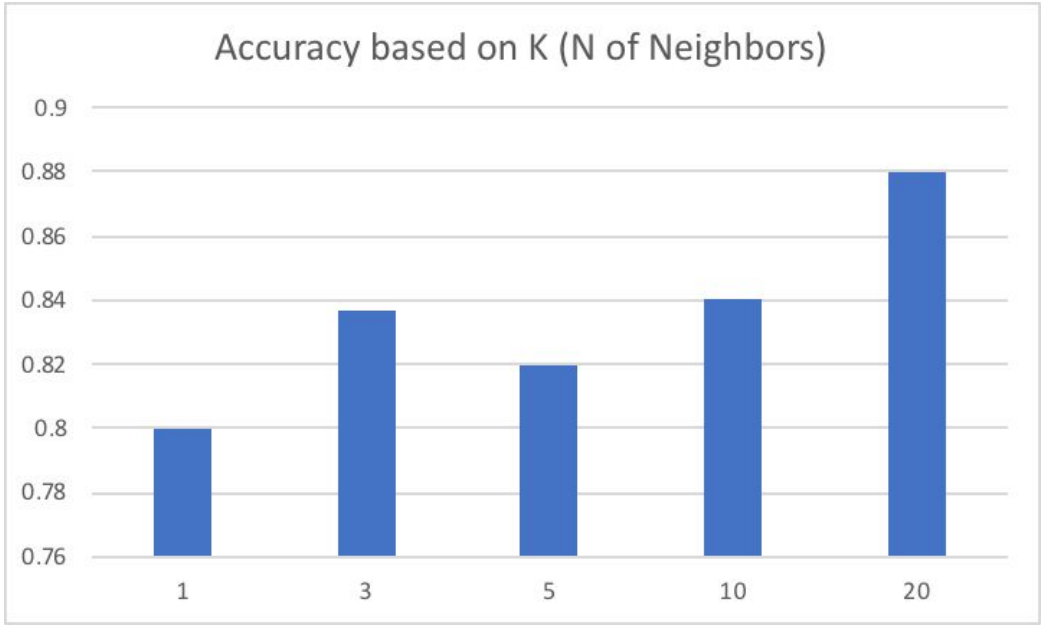
Prediction \ Reference	0	1
0	143	28
1	22	88

Precision: 0.8
Recall: 0.758

Prediction \ Reference	0	1
0	171	0
1	0	110

Precision: 1
Recall: 1

Accuracy vs. N of Neighbors



Time Comparison

In [97]:
```
%%timeit

k = 3
lst= []
for index, row in test.iterrows():
    testSet=  [row["Variance"], row["Skewness"], row["Curtosis"], row["Entropy"]]
    aux = pd.DataFrame(testSet)
    result,neigh = knn(train, aux, k)
    lst.append(result)
```

```
1min 54s ± 2.26 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [89]:
```
%%timeit
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(train.iloc[:,0:4], train['Class'])
# Predicted class
test['scikit']= neigh.predict(test.iloc[:,0:4])
```

```
41.7 ms ± 6.52 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Sci-kit implementation is approx. 2850 times faster

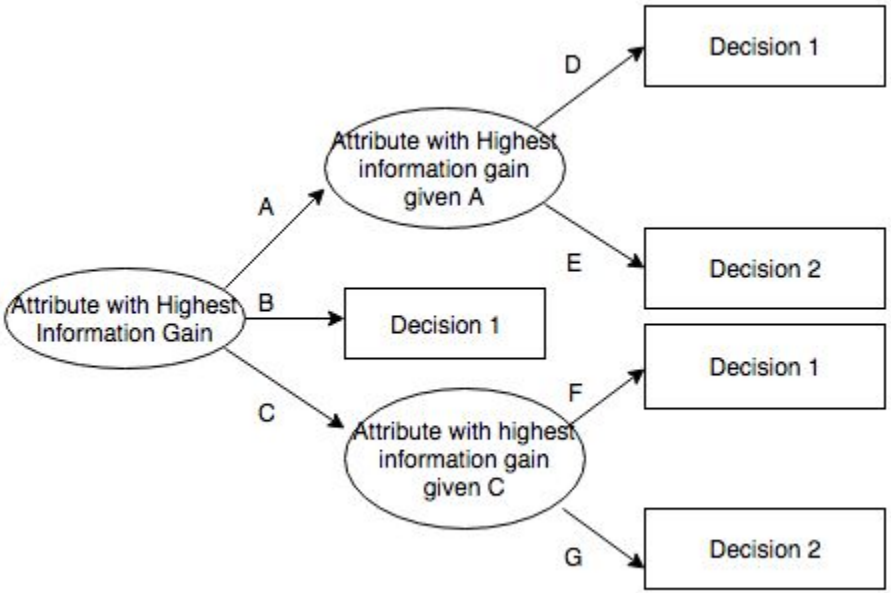
Decision Tree implementation

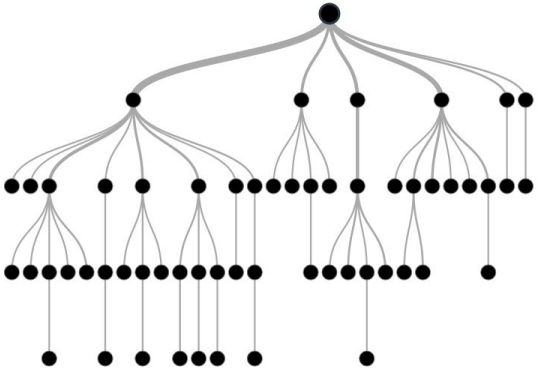
ID3 (Iterative Dichotomiser 3)

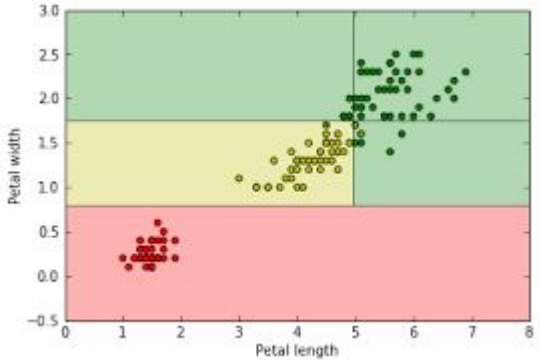
Ross Quinlan

Nodes:

- Leaves (decision)/ terminal nodes
- Inner Node (condition)







Gini index

$$= 1 - \sum_{t=0}^{t=1} P_t^2$$

A Gini score gives an idea of how good a split is by how mixed the classes are in the two groups created by the split. A perfect separation results in a Gini score of 0, whereas the worst case split that results in 50/50 classes.

We calculate it for every row and split the data accordingly in our binary tree. We repeat this process recursively.

Mean Accuracy Results

Minimum Node records	Maximum Depth		
	3	5	7
5	93.285	97.299	98.175
10	93.285	97.299	98.175
15	93.285	97.299	98.248
30	92.847	96.496	96.934

Scikit- Comparison

```
print('Mean Accuracy of our decision tree: %.3f%%' % (sum(scores)/float(len(scores))))
print('Mean Accuracy of scikit is %.3f%%' % clf.score(data_new, banknote))
```

```
Mean Accuracy of our decision tree: 97.299%
Mean Accuracy of scikit is 1.000%
```

From Scratch

	Predicted	0	1.0
True			
0		145	4
1.0		8	117

Scikit

	Predicted	0	1.0
True			
0		152	0
1.0		0	122

Time Comparison

Scikit implementation is approx. **6168** times faster

```
#%%timeit
for i in range(len(dataset[0])):
    str_column_to_float(dataset, i)

n_folds = 5
max_depth = 5
min_size = 10
scores = evaluate_algorithm(dataset, decision_tree, n_folds, max_depth, min_size)
```

```
scratchtime = time.time() - ps
print('Time is : %f' % scratchtime)
```

```
Time is : 13.223908
```

```
%%timeit
clf = DecisionTreeClassifier()
banknote = data['Class'].values.reshape(len(data),1)
data1 = data['Variance'].values.reshape(len(data),1)
data2 = data['Skewness'].values.reshape(len(data),1)
data3 = data['Curtosis'].values.reshape(len(data),1)
data4 = data['Entropy'].values.reshape(len(data),1)
data_new = np.hstack((data1,data2,data3,data4))
clf = clf.fit(data_new, banknote)
```

```
2.14 ms ± 125 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Support Vector Machine

$$c(x, y, f(x)) = \begin{cases} 0, & \text{if } y * f(x) \geq 1 \\ 1 - y * f(x), & \text{else} \end{cases}$$

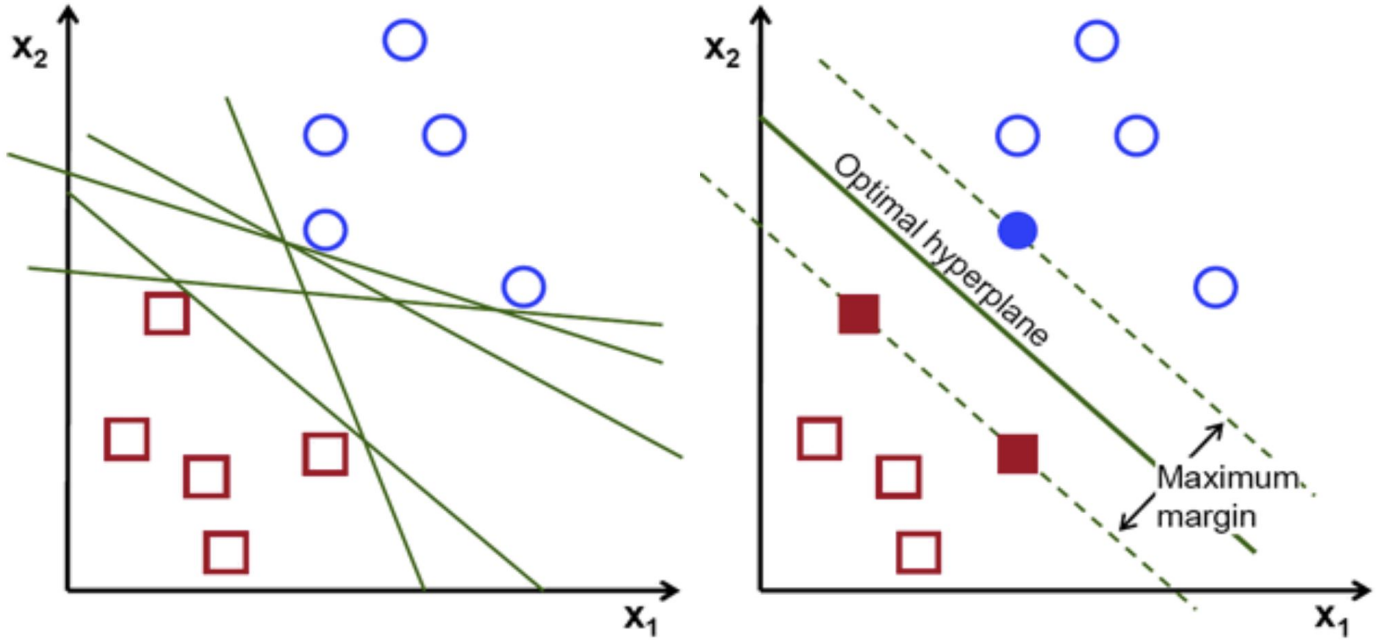
- *A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (*supervised learning*), the algorithm outputs an optimal hyperplane which categorizes new examples. In two dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.*

[a] If $Y_i = +1$; $wx_i + b \geq 1$

[b] If $Y_i = -1$; $wx_i + b \leq 1$

[c] For all i; $y_i (w_i + b) \geq 1$

Intuition Behind SVM



Gradient Update SVM

$$w = w - \alpha \cdot (2\lambda w)$$

Gradient Update — No misclassification

$$w = w + \alpha \cdot (y_i \cdot x_i - 2\lambda w)$$

Fit Method for SVM

```
start_time = time.time()
while(epochs < 10000):
    y = w1 * train_f1 + w2 * train_f2 +  w3 * train_f3 +  w4 * train_f4
    prod = y * y_train
    print(epochs)
    count = 0
    for val in prod:
        if(val >= 1):
            cost = 0
            w1 = w1 - alpha * (2 * 1/epochs * w1)
            w2 = w2 - alpha * (2 * 1/epochs * w2)
            w3 = w3 - alpha * (2 * 1/epochs * w3)
            w4 = w4 - alpha * (2 * 1/epochs * w4)

        else:
            cost = 1 - val
            w1 = w1 + alpha * (train_f1[count] * y_train[count] - 2 * 1/epochs * w1)
            w2 = w2 + alpha * (train_f2[count] * y_train[count] - 2 * 1/epochs * w2)
            w3 = w3 + alpha * (train_f3[count] * y_train[count] - 2 * 1/epochs * w3)
            w4 = w4 + alpha * (train_f4[count] * y_train[count] - 2 * 1/epochs * w4)
        count += 1
    epochs += 1
end_time = time.time() - start_time
print(end_time)
```

Results- Code from Scratch

Hyperparameters:

- Epochs = 10000
- Train-Test Split: 80:20
- Accuracy Score : 0.9709
- Run Time : 276.44 s

	Predicted	-1.0	1.0
True			
-1.0		157	7
1.0		1	110

Results- Sklearn

- Hyperparameters:
- Train-Test Split: 80:20
- Accuracy Score : 1.0
- Run-time: 0.00523 s

	Predicted	-1.0	1.0
True			
-1.0		164	0
1.0		0	111

Overview of Results

	KNN	Decision tree	SVM
Type	Lazy	Eager	Eager
Accuracy	0.82	0.97	0.97
Run time	1 min, 54 sec	13.22 sec	4 min, 36 sec

THANK YOU