Machine Learning: Assignment #3

Due on May 13, 2021

Prof. Graham Taylor

Jean Hounkpe, CTO VideoLogic

1 Theory

1.1 Solution 1.1

Decision trees recursively divide the training data space into regions that are parallel to the features and assign them to their most prevalent class. This process allows to generate a simple decision boundary when the data are easily separable along one or more features ($X_1 < 10$, $X_2 > 150$, etc.). With this same type of data, the 1-nearest neighbour algorithm would produce a very complex decision boundary especially when some examples are among instances of different classes. This is illustrated by Figure 1 and Figure 2.

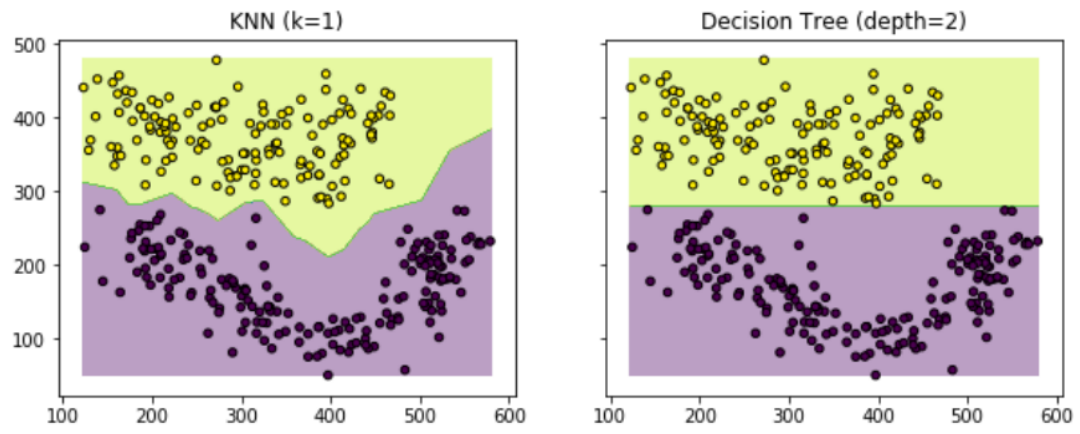


Figure 1: Decision boundary: simple (Decision tree) Vs complex (1-KNN)

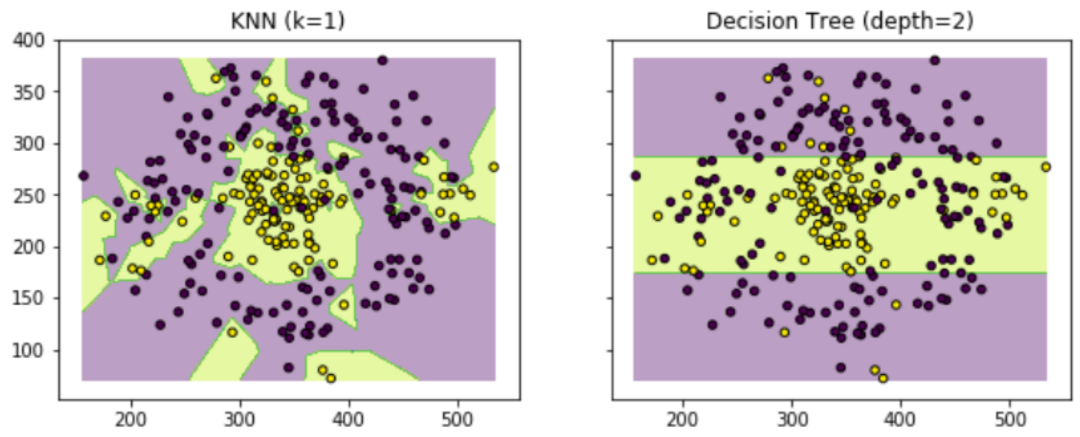


Figure 2: Decision boundary: simple (Decision tree) Vs complex (1-KNN)

On the other hand, when all the training data are perfectly linearly separable, but not along the features, then the 1-nearest neighbour generates a simpler decision boundary than the decision trees (Figure 3).

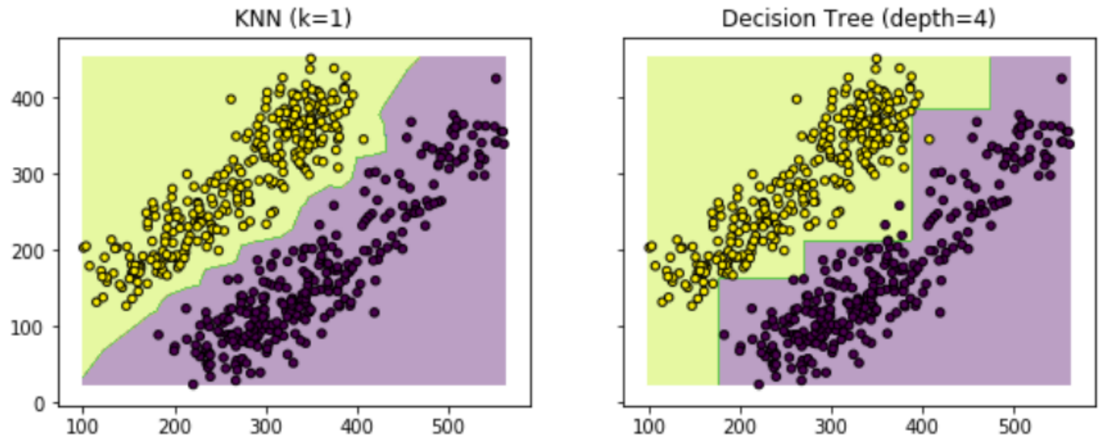


Figure 3: Decision boundary: complex (Decision tree) Vs simple (1-KNN)

1.2 Solution 1.2

Let's re-work the analysis in equations 4.3 to 4.6 of the texbook to show that the perceptron also performs better on negative examples ($y = -1$) after a weight update.

So we have a negative example: $y = -1$. We compute an activation a, and make an error: $ya < 0 \Rightarrow a > 0$. We now update our weights and bias. Let's call the new weights $w'_1, \dots, w'_D, b'$. Suppose we observe the same example again and need to compute a new activation a':

$$\begin{aligned} a' &= \sum_{d=1}^{D} w'_d x_d + b' \\ &= \sum_{d=1}^{D} (w_d - x_d) x_d + b - 1 \\ &= \sum_{d=1}^{D} w_d x_d + b - \sum_{d=1}^{D} x_d^2 - 1 \\ &= a - \left(\sum_{d=1}^{D} x_d^2 + 1 \right) < a \end{aligned}$$

Thus, the new activation is always at least the old activation minus one. Since this was a negative example, we have successfully moved the activation in the proper direction.

1.3 Solution 1.3

Let's modify section 4.5's convergence proof in the textbook to find the bound if x has a norm of at most R instead of 1.

In the perceptron convergence proof, the use of the assumption that all x have a norm of at most 1 is done by going from line 4.14 to line 4.15. This becomes with the new assumption:

$$\begin{aligned}
\left\|w^{(k)}\right\|^2 &= \left\|w^{(k-1)} + yx\right\|^2 && \text{def. of } w^{(k)} \ (4.13)\\
&= \left\|w^{(k-1)}\right\|^2 + y^2\left\|x\right\|^2 + 2yw^{(k-1)}.x && \text{quadratic rule (4.14)}\\
&\leq \left\|w^{(k-1)}\right\|^2 + R^2 + 0 && \text{assumption and } a < 0 \ (4.15)
\end{aligned}$$

Thus, the squared norm of $w^{(k)}$ increases by at most R^2 every update. Therefore: $\left\|w^{(k)}\right\|^2 \leq kR^2$.

Now putting this together with the fact that $w^*.w^{(k)} \geq k\gamma$ and because w^* being a unit vector implies $\left\|w^{(k)}\right\| \geq w^*.w^{(k)}$, we have for $\gamma > 0$:

$$R\sqrt{k} \geq \left\|w^{(k)}\right\| \geq w^*.w^{(k)} \geq k\gamma \quad \Rightarrow \quad R^2k \geq k^2\gamma^2 \quad \Rightarrow \quad \frac{R^2}{\gamma^2} \geq k$$

The bound is then $k \leq \frac{R^2}{\gamma^2}$.

2 Perceptron implementation

2.1 Solution 2.1

[nextExple]:
```
def nextExample(self, X, Y):
                 """
                 X is a vector training example and Y is its associated class.
                 We're guaranteed that Y is either +1 or -1.  We should update
                 our weight vector and bias according to the perceptron rule.
                 """

                 # check to see if we've made an error
                 if Y * self.predict(X) <= 0:    ### SOLUTION-AFTER-IF
                     self.numUpd  = self.numUpd  + 1

                     # perform an update
                     self.weights = self.weights + Y * X

                     self.bias    = self.bias + Y
```

[2]:
```
runClassifier.trainTestSet(perceptron.Perceptron({'numEpoch': 1}),
                                    datasets.TennisData)
```

```
Training accuracy 0.6428571428571429, Test accuracy 0.6666666666666666
```

[3]:
```
runClassifier.trainTestSet(perceptron.Perceptron({'numEpoch': 2}),
                                    datasets.TennisData)
```

```
Training accuracy 0.8571428571428571, Test accuracy 1.0
```

2.2 Solution 2.2

[4]:
```
runClassifier.trainTestSet(perceptron.PermutedPerceptron({'numEpoch': 1}),
                                    datasets.TennisData)
```

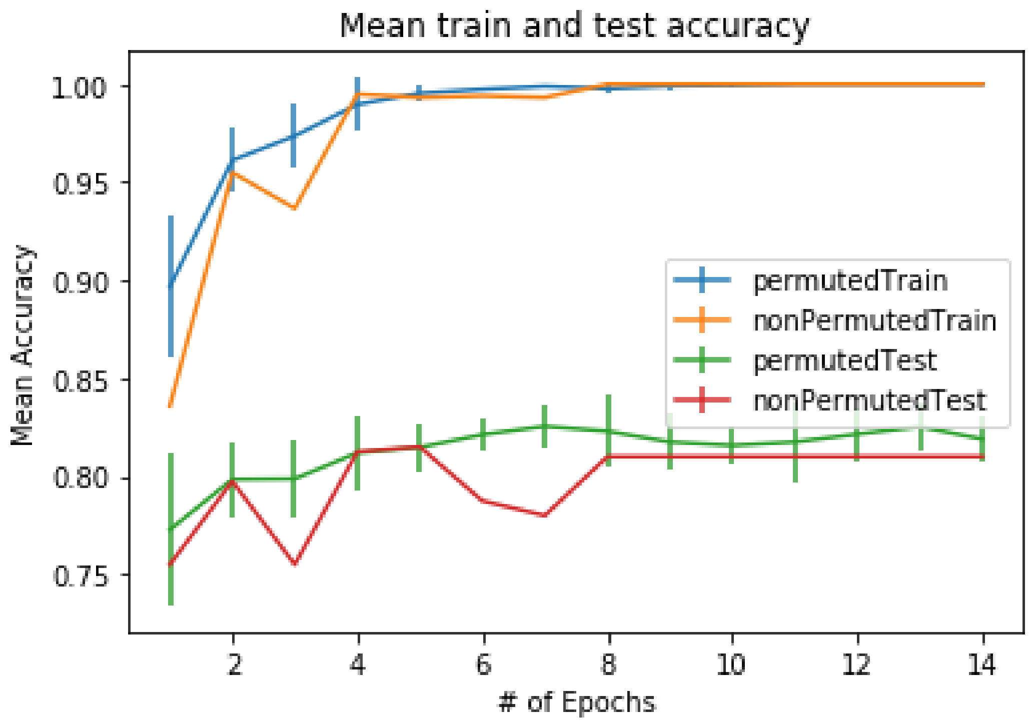
```
Training accuracy 0.6428571428571429, Test accuracy 0.5
```

[5]:
```
runClassifier.trainTestSet(perceptron.PermutedPerceptron({'numEpoch': 1}),
                                    datasets.TennisData)
```

```
Training accuracy 0.7857142857142857, Test accuracy 0.8333333333333334
```

[6]:
```
runClassifier.trainTestSet(perceptron.PermutedPerceptron({'numEpoch': 1}),
                                    datasets.TennisData)
```

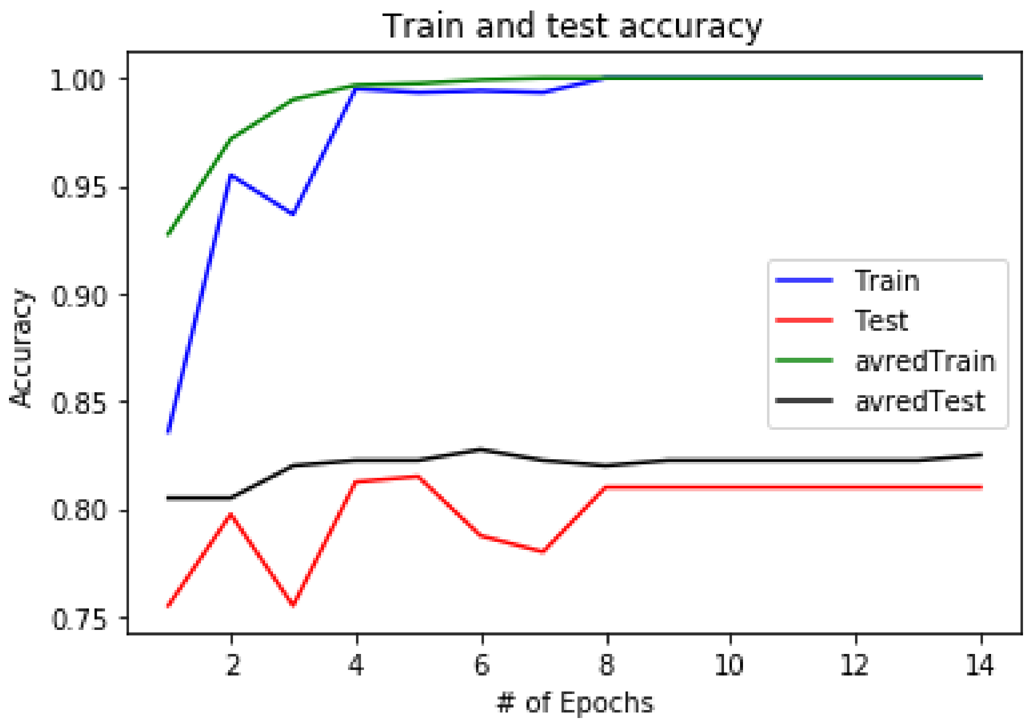
```
Training accuracy 0.7857142857142857, Test accuracy 0.6666666666666666
```



We can see that in addition to being more accurate on the testing set, the permuted perceptron converges faster as a function of the number of epochs than the non-permuted perceptron. The effect of variability in perceptron predictions during training due to the permutation of sample data also decreases rapidly with the increase in the number of epochs.

The fact that the data is re-permuted at each iteration has the effect of making the perceptron to more frequently update the weights and therefore converging more quickly. Also presenting the data in a different order at each iteration will have the advantage of allowing the perceptron to learn from more examples which increases its ability to better generalize.

2.3 Solution 2.3



By construction, the vanilla perceptron algorithm updates the weights each time the system misclassifies the next example, which results in the system placing more importance on later examples than earlier ones. The order in which the samples are presented to the perceptron is therefore crucial to the final weights learned by the system. The average perceptron aims at solving this problem by keeping in memory all the previous weights (these put more importance on the previous examples) and by using as final weights the average value of all those previous weights.

As illustrated in the figure, the average perceptron is almost always better than the vanilla perceptron, in the sense that it is like being trained on more samples which allows it to perform better on the test data.