NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Assignment 3

TDT4173: Machine Learning and Case-Based Reasoning

Submitted by
Vittorio Triassi

November 2019

1 Theory

1.1 What is the core idea of *deep* learning? How does it differ from *shallow* learning? Support your arguments with relevant examples.

With the increase of the computational power, also the way problems have been addressed has changed. *Deep learning* takes advantage of huge amount of available data, which thanks to more powerful hardware, allows us to perform complex computations. Usually, when we talk about deep learning, we are referring to Neural Networks and more specifically to Deep Neural Networks. The main benefit coming from deep learning, which is also why it differs from *shallow learning* is that the complexity of the model allows us to perform an *automatic feature extraction* from the data. On the other hand in shallow learning, there is no such feature learning and *handcrafted features* are used instead. Overall, it really depends on the amount and the type of data we are dealing with. For instance, if we have images or voice signals, deep learning comes in handy because it is possible to recognize useful patterns in the hidden layers of our network. At this point, we might think that a deep learning approach is always to be preferred. The truth is that it is not like that because training deep models is difficult and there are other side effects that arise. If we are addressing a problem for which we already know features that perform well on our data, a shallow model is always better. Typical examples of deep learning models can be Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN) and autoencoders, while Support Vector Machines (SVM) and decision trees can be seen as typical models of shallow learning.

1.2 Describe the comparison between following machine learning techniques: *k-NN, decision tree, SVM* and *deep learning*. Also, discuss the situations, for each of these techniques, as *why* and *when* NOT to prefer them.

When dealing with machine learning problems, it can be difficult to identify the right tecnique to use. The main aspects we should keep in mind generally are: (1) How much data do we actually have? (2) How are we dealing with overfitting? (3) Is the problem linearly separable? (4) What about the performance of our model?

In the case of *k-nearest neighbors* (k-NN) algorithm, we are dealing with a classification algorithm in which the basic idea is to classify a new instance considering the k closest data points. Without going into details, we can say that the following algorithm is quite simple to understand and is non-parametric, which means that it does not make any specific assumption on the distribution of the data. Unfortunately, it can take up a lot of memory and that is why is not recommended for huge datasets. It works quite well when we have a small number of dimensions instead.

Also *decision trees* are non-parametric. They aim at solving classification problems as well, by using a tree structure in which the leaves represent the expected classes. The internal nodes behave in such a way that they split the instance space in subspaces. One problem with decision trees is that they easily run into overfitting. On the other hand they are fast and we do not have to take care of tuning them with a lot of parameters as it happens with other tecniques (e.g. SVMs).

In *Support Vector Machines* (SVMs), we try to find a hyperplane that is able to separate two classes of the data by finding the largest margin. SVMs have a good accuracy compared to other tecniques but are very hard to tune because they involve a lot of parameters. In contrast to other tecniques, SVMs perform really well. Probably only an ensemble like a random forest can do better. One of the big limitations with SVMs is that they can not address multi-class problems and in the end it can be seen as a binary classifier.

As it has previously mentioned, *deep learning* can be linked to Deep Neural Networks. Also with such networks we want to perform classification problems. In this case, having a lot of data really helps the behavior of the model. A few drawbacks are the time required for training the model (since it gets really deep and the number of parameters increases a lot) and the difficulty when choosing the right topology.

1.3 Discuss when should we use ensemble methods in context of machine learning? Explain briefly any 3 types of ensemble machine learning methods.

Ensemble methods, and more specifically *ensemble learning*, comes in handy when we want to improve the performance of our model. What we do is not to simply rely on one prediction, instead we combine predictions made by several models. In other words, rather than training just one model, we train several of them and combine their hypotheses to come up with just one that is more accurate. This approach is really useful when speed time is not our concern. In fact, if we want to achieve the best score possible, ensemble methods allow us to obtain the best outcome. That is also why they are quite spread in data science competitions. The most common tecniques when dealing with ensemble methods are: *bagging*, *boosting* and *stacking*.

Bagging can be defined as a parallel ensemble because each model is built in such a way that is independent from the others. The following tecnique is usually used when we aim at reducing the variance of the learners. The basic idea in bagging is to create subsets of data that come from training samples randomly drawn with replacement. The final goal is to aggregate the learners trained on the following samples. If we are dealing with a regression task, we simply average the results of the learners and take that as the final hypothesis. If we are performing a classification task instead, we select the predicted class through a voting mechanism, where the class that has been voted most is the one we choose as final prediction.

Boosting is another of the most used ensemble methods in machine learning. The following method is mainly used to reduce the bias in the model. The idea in this case is to perform a sequential process in which the learners are stacked one by one and together they form the ensemble. One of the most known approaches for boosting is the *AdaBoost* algorithm. In AdaBoost, we start with the same weights for the whole dataset during the first round. Then, in the following rounds, we put more emphasis on the samples that were previously misclassified and decrease the ones that were correctly classified instead. By doing so, the final classifier is the combination of all the learners using a majority voting.

The third and last ensemble method considered is *stacking*. In bagging and boosting we have always performed the ensemble by using one learning algorithm. Differently from that, when considering the stacking, we can train several models by using different algorithms and in the end we combine their results to make a prediction.

2
Programming

2.1 k-NN implementation from scratch

As far as the k-Nearest Neighbors algorithm, we will divide the discussion into two parts. First, the k-NN algorithm with a regression approach will be performed and finally the same will be done addressing the problem as a classification task.

k-Nearest Neighbors with regression

In order to compute the **k-Nearest Neighbors** with the regression approach, it is important for us to establish a distance measure. Such measure will be responsible for computing the k smallest distances between our test example and all the previously stored data points. In our case, we will use as test example, the 124^{th} example of the file containing the training data. Moreover, we are interested in computing the k-NN when $k = 10$.

Since the Euclidean distance is the most used and it is a L2 norm, we decide to use `numpy.linalg.norm`, which by default implements for the `ord` parameter the norm 2 (see Figure 2.1). After we have defined how to compute the k smallest distances between our test example and our dataset we are ready to perform the **k-Nearest Neighbors** based on mean (see Figure 2.2). The first 10 results are shown in Figure 2.3.

It is worth pointing out that at least with the chosen test example, the first smallest distance in our k-NN will have a distance equal to zero. This is quite reasonable, since the distance from a point and itself is clearly zero.

```python
# testExample = new data point
# k = number of neighbors we are interested in
def computeDistances(X, testExample, k):

    distances = []

    for i in range(len(X_reg)):
        distances.append([np.linalg.norm(X[i] - testExample), i])

    # sort the list to have the smallest distances in the top
    distances = sorted(distances, key = lambda x: x[0])
    # take only the first k distances
    distances = distances[:k]

    return distances
```

Figure 2.1: Compute the k smallest distances.

```python
# testExample = new data point
# k = number of neighbors we are interested in
def kNNRegression(X, testExample, k):

    distances = computeDistances(X, testExample, k)

    print(*distances, sep = "\n")
    print("\n")

    samples = []

    for i in range(len(distances)):
        samples.append(distances[i][1])

    s = 0
    for i in samples:
        s += y_reg[i]
    prediction = s / k

    return prediction
```

Figure 2.2: k-NN regression implementation.

Now we use our k-NN implementation and predict on a test example:

```
k_reg = 10
testExample_reg = X_reg[123]
pred_reg = kNNRegression(X_reg, testExample_reg, k_reg)

print("k-Nearest Neighbors (regression) with k = " + str(k_reg))
print("Test example = " + str(testExample_reg) + ". Prediction: " + str(pred_reg))
```

```
[0.0, 123]
[0.17320508075688762, 126]
[0.20000000000000018, 72]
[0.22360679774997816, 133]
[0.22360679774997896, 146]
[0.30000000000000016, 73]
[0.3464101615137755, 63]
[0.3605551275463984, 83]
[0.3605551275463989, 127]
[0.3741657386773947, 54]


k-Nearest Neighbors (regression) with k = 10
Test example = [6.3 2.7 4.9]. Prediction: [1.6]
```

Figure 2.3: k-NN prediction.

k-Nearest Neighbors with classification

This time, the k-NN algorithm is addressed as a classification problem. In fact, we want to classify the k nearest data points to our test example considering a majority vote approach. We do not need to define the distance measure again since it will be the same used before.

Since we can run into predictions that output "ties", a random prediction among the classes we think might satisfy our problem is drawn.

Moreover, since the distances were initially computed and stored with several decimal digits, we decided to round them until three, otherwise it would have been quite hard to show any interesting result given the size of our dataset.

In Figure 2.4 the k-NN classification algorithm implementation is shown and in Figure 2.5, the prediction with $k = 10$ is provided.

```python
def kNNClassification(X, testExample, k):

    distances = computeDistances(X, testExample, k)

    for i in range(len(distances)):
        distances[i][0] = round(distances[i][0], 3)

    print(*distances, sep = "\n")
    print("\n")

    counts = Counter(x[0] for x in distances)
    minimum_distance = counts.most_common(k)[0][0]

    # array used to store predictions in
    # the case we have ties in 'distances'
    predictions = []

    for i in range(len(distances)):
        if(minimum_distance == distances[i][0]):
            predictions.append(y[distances[i][1]])

    # we draw a random prediction among the classes
    # we think might satisfy our problem
    prediction = random.choice(predictions)

    return prediction
```

Figure 2.4: k-NN classification implementation.

```python
k = 10
testExample = X[123]
pred = kNNClassification(X, testExample, k)

print("k-Nearest Neighbors (classification) with k = " + str(k))
print("Test example = " + str(X[123]) + ". Prediction: " + str(pred))
```

```
[0.0, 123]
[0.173, 126]
[0.245, 146]
[0.361, 127]
[0.361, 72]
[0.374, 133]
[0.412, 83]
[0.424, 111]
[0.436, 138]
[0.48, 54]


k-Nearest Neighbors (classification) with k = 10
Test example = [6.3 2.7 4.9 1.8]. Prediction: [1]
```

Figure 2.5: k-NN classification implementation.

2.2 k-NN vs SVM vs Random Forest on sklearn digit dataset

For the purpose of this task, the `sklearn` implementations of k-NN, SVM and RandomForest on the `sklearn.datasets.load_digits` dataset were used. Since we did not have specific constraints on the parameters to provide in input to our classifiers, we thought about setting quite default values. This means that as far as the k-NN classifier, we used `n_neighbors = 5`; with SVC, that stands for Support Vector Classifier, `gamma=0.001` was used and finally, as regards to RandomForest we set `n_estimators=100` (see Figure 2.6). By doing so, we should obtain the default performance for such models.

```
11 classifiers = [[kNN(n_neighbors=5), "k-Nearest Neighbour"],
12                [SVC(gamma=0.001), "Support Vector Machine"],
13                [RFC(n_estimators=100), "Random Forest"]]
```

Figure 2.6: Classifiers and their signatures.

The first step to carry out was the split of our dataset in training and test examples. In order to do that, we used the `train_test_split` function and decided to have a `split_size = 0.30` which means that our test examples were the 30% of the whole dataset. Then, we flatten our training and test examples, since the `fit` method does not accept 3-D examples. Now, we proceed by fitting our models and showing the accuracy we get (see Figure 2.7).

```
Accuracy k-Nearest Neighbour: 0.9925925925925926
Accuracy Support Vector Machine: 0.9907407407407407
Accuracy Random Forest: 0.975925925925926
```

Figure 2.7: Classifiers accuracy.

It turns out that all the classifiers perform really well on unseen examples. The best performance is obtained with the k-NN classifier, followed by the SVC. The RandomForest performs just slightly worse instead. In order to have a better insight on how we are doing on our data, it can be a good idea to plot the *confusion matrix* for each classifier.

A confusion matrix is a table that helps us to visualize how our classifier performs on test examples for which we already know the true labels. In our case, we decided to plot both: a normalized and a not normalized version of the confusion matrix as shown in Figures 2.8, 2.9 and 2.10.

The interesting aspect that came out from this task is that thanks to such libraries, it gets way faster to get a deeper understanding of the addressed problem. This is because we are not implementing everything from scratch but we are rather using implementations that let us to go straight into the problem comparing different models at the same time. Something that seems to be really important when addressing ML/DS problems, is the ability to show the results obtained. As far as the classifiers

used in the following task, we had the chance to play a bit with all the parameters it is possible to specify in their implementations. The results plotted in the confusion matrices aforementioned are not the results of any particular tuning. Despite that, trying different configurations, very good performance was achieved. These parameters strictly depend on the size of our dataset and on the problem we are trying to address. So, rather than plotting several tests, we thought about trying the models a bit and just providing the default configurations.

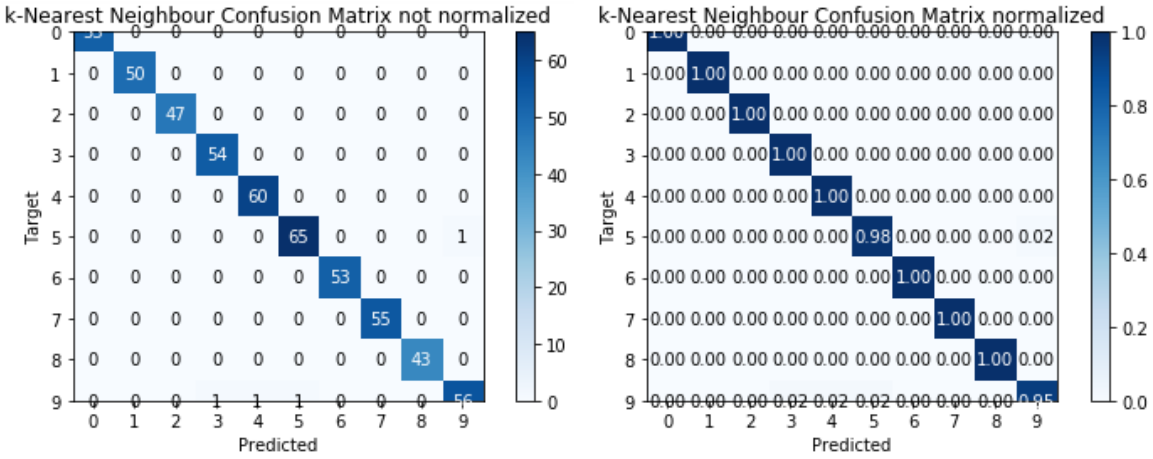

Figure 2.8: Confusion Matrix for k-NN.

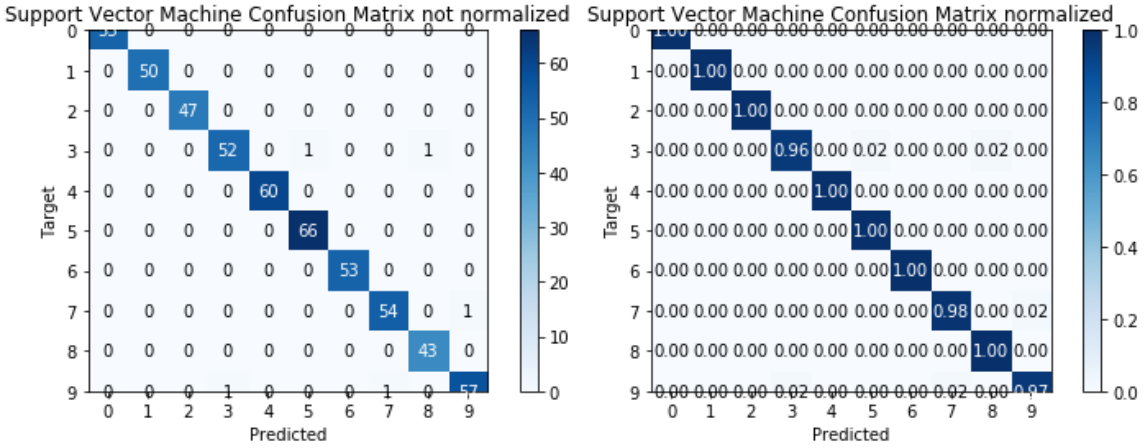

Figure 2.9: Confusion Matrix for SVC.

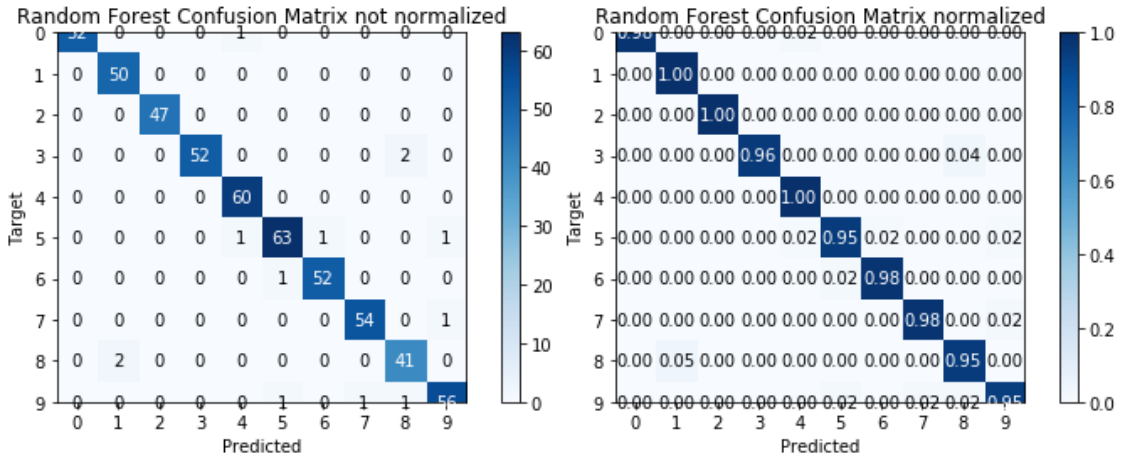

Figure 2.10: Confusion Matrix for RandomForest.

2.3 Neural Network from scratch

For the following task, we were asked to solve the XOR problem by using a Neural Network (NN) created from scratch. A very common problem with NNs, is the need to increase the non-linearity to be able to generalize and consequently to correctly predict over specific datasets. In our case, we want to solve a basic but at the same time very effective problem, and to do so we start asking ourselves how many layers we need in our network. It turns out that in order to be able to predict hypotheses for the XOR problem, it is not possible to use just a single neuron. The reason is that we can not plot one line that is able to separate our data points in such a way that the predictions are correct. It seems clear then, that we need to increase the complexity of the function we want to predict and to achieve that we need a more complex architecture. The impacts that such a thing has on our network is the need to stack a hidden layer made up of at least two neurons. So we would end up with a Neural Network as follows:

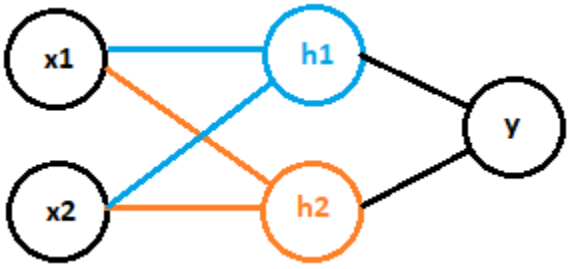

Figure 2.11: Neural Network architecture.

In Figure 2.11 we can see that we have two input features x_1 and x_2 (two values from the XOR truth table) and one hidden layer with two activations h_1 and h_2 that let us to come up with a prediction in the output layer y. This is the minimum we need in order to be able to predict any combination for this problem. In our

implementations, we can set the number of hidden neurons that is initially set equal to 2 but can be modified with any other value. In our task, we do not have a specific dataset but we would rather want to learn a function that predicts the XOR table. That is why the first step is the creation of our truth table (that is going to represent the training examples) with the respective true labels (the value for which the two input features are true or false) as shown in Figure 2.12. Then, we carry on by defining the functions for the sigmoid, the derivative of the sigmoid, the forward and backpropagation.

```
X = np.array([[0,0],
              [0,1],
              [1,0],
              [1,1]])

y = np.array([[0],[1],[1],[0]])
```

Figure 2.12: Creation of the XOR truth table.

As far as parameters such as the learning rate and the number of iterations, several combinations were carried out. In the end, we obtained quite stable results when the traning went over 20000 iterations with a learning rate equal to 0.3. When trying to predict, we obtained the results shown in Figure 2.13. As we can see, our configuration let us to predict with quite good performance. As regards to the *Loss function* and how this was minimized, we plot the number of iterations over the cost. In Figure 2.14, it is quite clear that the function gets correctly minimized and converges towards zero updating properly the weights of the model.

```
Classes:
 [[0 1 1 0]]
Predictions :
 [[0.01843785 0.95232588 0.95232594 0.07880215]]
```

Figure 2.13: Predictions and expected classes

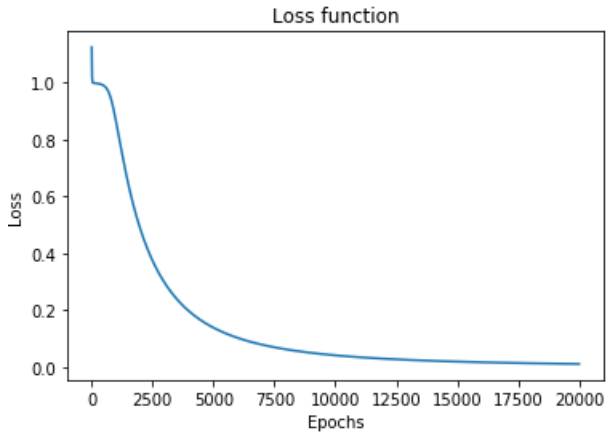

Figure 2.14: Loss function