**KNN and MLP approaches for classification problems**

*Rodrigo Palma*
Department of Electrical and Computer Engineering, Temple University
rodrigo.palma@temple.edu

**Introduction:** There are two main types of tasks in the machine learning field: supervised and unsupervised. The first type is used when our data is labeled, so we can train our model with the correct labels and afterward check its accuracy. The second type is used when we do not have the labels, in this case, we would cluster the dataset and after that, a human should see the results and decide what are the labels. For classification problems, we use supervised learning because we usually know the labels we want to classify. In this work, we classify two datasets with two different algorithms: a non-neural network and a neural network-based approach. The first algorithm is the K-nearest neighbors (KNN) algorithm, which is a very simple and well-known algorithm used for classification problems. The second algorithm is the Multi-Layer perceptron (MLP), which uses a neural network that is based on layers and neurons, each neuron has an activation function that says if the information will be passed of not to the next layer. These algorithms were trained in a 2-dimensional and 5-dimensional dataset, where the first one has 1000 samples and the second one has 10000 samples.

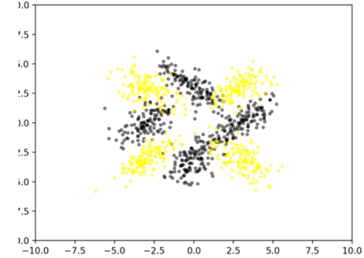

Figure 1. Plot of 2D dev dataset

**KNN Algorithm:** The KNN is a simple and useful supervised algorithm used for classification problems. This algorithm searches for the K-nearest neighbors of a sample and checks their classes. The distance between the samples can be calculated with different techniques such as Manhattan distance, Euclidian distance, or Minkowski distance. The final decision can be a simple majority vote. I decided to use KNN because of the shape of the 2D dataset when plotted (Figure 1). It shows that the points are close together and the neighbors would be from the same class as the sample and consequently, the classification would work well.

To find the best value for K (the number of neighbors), I used the SearchGridCV method from Python's scikit-learn library that iterates over all parameters that I am interested in and returns the best combination. To measure accuracy, I split the train data into into 5 groups to cross-validate them. Both datasets got better results with the Manhattan distance, which was interesting, and I could not explain why that happened. For the K value, the 2D dataset had a better result with 30 neighbors, while the 5D dataset had a better performance with 850 neighbors. For both algorithms, when I used a different number of neighbors, the error rate increased significantly.

To train the model, I used the KNeighborsClassifier function from scikit-learn library (v0.21.3) using python 3.7.

**MLP Algorithm:** Multi-Layer perceptron is a deep, artificial neural network composed of an input layer that receives the input signal, the output layer that makes the final decision about the input data, and N number of hidden layers between these two layers, where the computation engine happens. These neural networks can be used to solve supervised learning problems such as classification. The idea of the network is to set an initial set of parameters (weights and biases), learn with the errors that the output has with the initial parameters, and back-propagate the error so the network can learn the best combination of parameters to achieve the minimum error. The error can be measured in many different ways, such as the root mean squared error. Many decisions must be taken to create an architecture of a network. First, the number of hidden layers, which can be very challenging to find. Second, the number of neurons each layer should have. Third, the initial weights and biases for each hidden layer. Forth, the activation function for each

layer. Fifth, the loss function and the optimizer to improve the weights and biases and, finally, the parameters such as learning rate, epochs, and batch size.

For my architecture, I started with one hidden layer with 128 neurons and the Rectified linear unit (ReLu) activation function. For the weights and bias initialization, I used a random normal function. This setup did not work as I expected because the accuracy was decreasing for each epoch. So, I created a new hidden layer and started to try a different combination of parameters and activation functions. My final architecture has 2 hidden layers, with 1024 and 512 neurons respectively and I sticked with the same weights and biases initialization. I also tried more neurons than these ones, but the accuracy started to decrease probably due to overfitting. I multiplied the input for each layer with its respective weights and added the bias term. After weights and biases, I added an activation function; I tried to use ReLu for both hidden layers, but the performance was very low, so I tried to use SoftMax activation for the first hidden layer, and ReLu for the second hidden layer. This combination had the best accuracy. The output layer uses the sigmoid function to make the final decision. After that, I had to choose a loss function to optimize the weights and biases. This was very tricky because most functions did not work well for my set up. The function that produced the best results was SoftMax cross-entropy with logits, and I also used the L2 regularization. For the optimizer, I tried Gradian Descendent and Adam, this last one worked better in my model. The learning rate started in 0.001 and I had to decrease it exponentially (exponential decay function) every 3 epochs to adjust the learning rate throughout the epochs, otherwise it would overfit. The best batch size was 200 and I used 50 epochs. These last parameters had a huge influence on the result. Learning rates higher than 0.001 overfit the model. To train this network, I used 100% of the train dataset, and to measure the accuracy, I used both dev and eval datasets. I did not use dev and eval datasets to train the model. I used TensorFlow version 1.13.1 with Python version 3.7.

**Results:** The final results are presented in Table 1. First, I ran all algorithms in a MacBook pro with a 2.3GHz 8-core 9th-generation Intel Core i9 processor. KNN ran much faster than MLP (as expected), about 10 seconds in the 5D dataset, while MLP took about four minutes in the same dataset. The two algorithms used in this work got similar results and it is hard to find the best one. The results show that, for the 2D eval dataset, KNN had better performance in time and error rate. Again, for the 2D dataset, it seems that KNN would work quite well due to the geometry of the data. Since the samples are together, the majority vote for the neighbors would end up with the correct label. We can calculate the confidence level of these results compared to the baseline algorithm from last semester. For KNN and the 2D dataset, the train has a confidence level of 71%, the dev has a confidence level of 82% and the eval result has a confidence level of 82%. For KNN and the 5D dataset, the train has a confidence level of 50%, the dev has a confidence level of 63% and the eval result has a confidence level of 57%. For the MLP algorithm and the 2D dataset, the train has a confidence level of 77%, the dev has a confidence level of 82% and the eval result has a confidence level of 70%. For MLP and the 5D dataset, the train has a confidence level of 70%, the dev has a confidence level of 82% and the eval result has a confidence level of 71%.

| | **2D Data** | | |
|---|---|---|---|
| **Algorithm** | **Train** | **Dev** | **Eval** |
| KNN | 7.93% | 7.95% | 7.85% |
| MLP | 7.71% | 7.95% | 8.35% |
| | **5D Data** | | |
| **Algorithm** | **Train** | **Dev** | **Eval** |
| KNN | 36.88% | 37.07% | 36.81% |
| MLP | 36.53% | 36.67% | 36.48% |

Table 1. Error rates for each algorithm in its respective dataset.

**Conclusions:**

Both approaches had similar results, and MLP took longer to train compared to KNN. The error rates found had a confidence level of around 70%, which can be considered a good number to reject the null hypothesis. For the KNN algorithm, the most important parameter that you need to find is the number of

neighbors, because this is what really changed the error rates. However, the MLP was very complicated to build and to get good results out of it. The learning rate and the batch size were crucial to get these results. The same is for the activation function and the number of hidden layers. Only one hidden layer decreased the accuracy and more than two overfit the model so the results for the dev dataset started to decrease. Overall, with the set up used for these two algorithms, the results were very similar, so the simplicity of KNN overcomes the complexity of the MLP network.