AI project report

Anh Kiet DUONG

November 2022

Contents

1 TP1

1.1 K nearest neighbor

1.1.1 Recall the general principle

K nearest neighbor (k-NN) [1] is a non-parametric supervised learning. For each test point, I calculate their distance from the points of the training set using the euclidean distance formula.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{1}$$

Where d is distance, $(x_1, y_1); (x_2, y_2)$ respectively are the coordinates of the 2 points need to calculate the distance. Then I take k training points with the smallest distance, k is a user-defined constant. At this step, I have many ways to choose the label for the test data, the most common is to choose the label with the highest frequency.

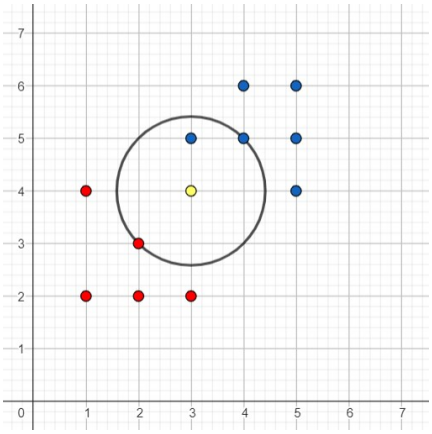


Figure 1: Illustrate the kNN algorithm. The blue points and the red points are the training data. Yellow point is testing data, in this case k=3 and the label of the test data is blue.

1.1.2 Version 1

Code a normal loop to compute the distances then sort them and pick out k points with the closest distance.

```python
def distance(x, y):
    return math.sqrt((x[0]-y[0])**2 + (x[1]-y[1])**2) #Euclidean Distance

def kppv_1(X_test, X_train, labels, K):
    clas = []
    for i in X_test.T:
        distance_label = []
        for j, label in zip(X_train.T, labels):
            distance_label.append([distance(i, j), label]) # lists of distance to node
            ↪ Ai and node Ai's label

        distance_label = sorted(distance_label, key=lambda X_test:X_test[0])[:K] # sort
        ↪ the list of distance and get top k results
        distance_label = [i[1] for i in distance_label] # get labels of top k nodes
        clas.append(max(distance_label,key=distance_label.count)) # get label appear max
        ↪ times in top k nodes
    clas = np.array(clas).astype(int) # set type
    return clas

affiche_classe(test, kppv_1(test, data, label, K), 2) # visualize
```

1.1.3 Version 2

Use sklearn's built-in library [2], then declare a model *KNeighborsClassifier* and train it.

```
def kppv_2(X_test, X_train, labels, K):
    knn = KNeighborsClassifier(n_neighbors = K) # create model
    knn.fit(X_train.T, labels) # fit model
    clas = knn.predict(X_test.T) # predict
    return clas

affiche_classe(test, kppv_2(test, data, label, K), 2) # visualize
```

1.1.4 Version 3

Use techniques to optimize computation speed. First, use multithreading to calculate the distance between points. Then, instead of having to sort like version 1, I just need to use the algorithm to choose the first k distances.

```
def kppv_3(X_test, X_train, labels, K):
    clas = []
    distances = pairwise_distances(X_test.T, X_train.T, metric="euclidean", n_jobs=-1) #
    ↪  compute the distance
    idxs = np.argpartition(distances, K)[:, :K] # get top k nodes by distance

    for i in range(len(X_test.T)):
        distance_label = [labels[j] for j in idxs[i]] # get label of top k nodes
        clas.append(max(distance_label,key=distance_label.count)) # get label appear max
        ↪  times in top k nodes

    clas = np.array(clas).astype(int) # set type
    return clas

affiche_classe(test, kppv_3(test, data, label, K), 2) # visualize
```

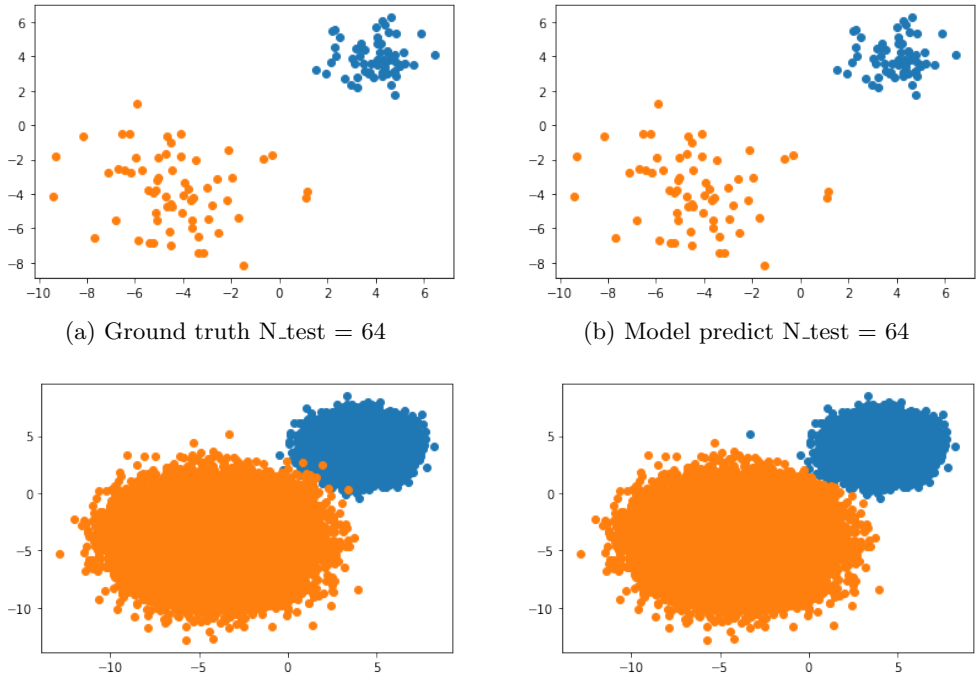
1.1.5 Conclude

I will visualize the test cases when the number of test data $N_test = 64$ and $N_test = 65536$ to validate the algorithm. In all tests with any values of N_test, all 3 versions gave the same results so I only show 1 representative image of kNN algorithm. Next to it is the ground truth label of the test data. We can see when $N_test = 64$ the model works very well. However, when N increases ($N_test = 65536$), some random data is far away from the center, then the model is confused and leads to some wrong results.

Compare the execution time of the 3 versions we can see that version with some optimization algorithms suggested by me is faster than other versions (including sklearn version). In the case of small N, since manual installation methods require some init steps, the speed is a little slower. However, as n increases gradually my algorithm shows more and more its speed.

Version	N=64	N=4096	N=65536	N=1048576
V1	301ms	7.11s	54s	12m
V2	207ms	658ms	2.98s	43.6s
V3	322ms	621ms	1.47s	22.4s

Table 1: Compare the execution time of the 3 versions. V1, V2, V3 are version 1, version 2, version 3 respectively and N is the number of test data.



(a) Ground truth N_test = 64

(b) Model predict N_test = 64

(c) Ground truth N_test = 65536

(d) Model predict N_test = 65536

Figure 2: Ground truth label of the test data and label predict by kNN algorithm.

1.2 Perceptron

1.2.1 Recall the general principle

Perceptron [3] is a parametric supervised learning. The algorithm is illustrated as shown in Figure 3. Where x_1, x_2, X_3 are inputs and w_1, w_2, w_3 are the corresponding coefficients, respectively. An activation function f is, f could be $f(x) = x$, $f(x) = tanh(x)$,... In the case of a classification problem, a threshold value is set to determine the output of the model, a python code that illustrates:

```
y = 1 if f(x) > threshold else 0
```

And to train the model we use a gradient descent vector to update the weights W with the formula:

$$W = W + \alpha \frac{\partial L}{\partial W}. \tag{2}$$

Where L is loss value, α is defined learning rate. Nowadays, the development of perceptrons puts many layers on top of each other and each layer has many nodes. This is the same premise for the development of AI and deep learning today.

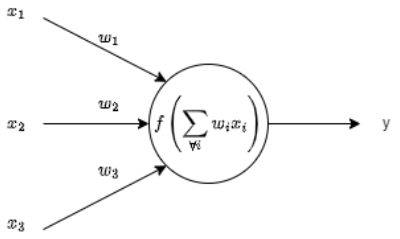


Figure 3: Illustration of perceptron

1.2.2 Implement

The implemented code of perceptron:

```
def loss(y_pred, y_true):
    return ((y_pred - y_true)**2).mean(axis=0) # loss function

def sigmoid_array(x):
    return 1 / (1 + np.exp(-x)) # compute sigmoid

def perceptron(x,w,active):
    y = np.multiply(x.reshape(x.shape[::-1]), w.reshape(-1))
    if (active):
        return np.tanh(np.sum(y, axis = 1)) # compute tanh if set active
    else:
        return sigmoid_array(np.sum(y, axis = 1)) # else compute sigmoid

def apprentissage(x, yd, active):
    mdiff = [] # losses
    lr = 1e-3 # learning rate
    epochs = 10000 # number of epochs

    x = np.vstack([np.ones([data.shape[1]]), data]) # add bias
    w = np.random.random_sample([1, x.shape[0]]) # random init weight

    for e in trange(epochs):
        yp = perceptron(x, w, active) # compute y predict
        mdiff.append(loss(yp, yd)) # append loss
        w += lr*np.mean((yd - yp)*x, axis = 1) # update weight lr*grad

    return w, mdiff
w, mdiff=apprentissage(data, label, activation)
```

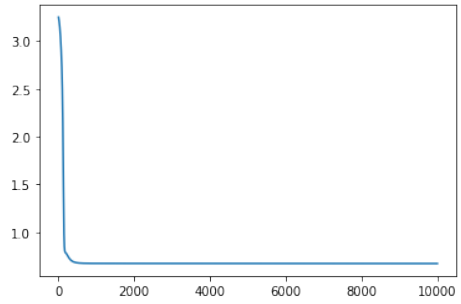
In the settings we set the learning rate to 0.001 and the number of epochs to 10000. In Figure 3 we can see that in the early epochs, the loss tends to decrease rapidly. However, in the epochs after the loss, the loss almost never decreases, usually because the model does not have enough weights, high learning rate, not enough training data or is too good. In my opinion in this case the model is too good, it can classify the training set well.

1.3 Conclude

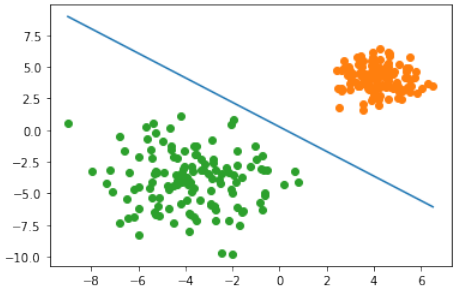
The perceptron algorithm gives better **online** performance than the kNN algorithm. Because when training (**offline**) the model is extracted from the training data the weights for a function. Then, for each test data, we just need to re-execute the function with those weights. Whereas kNN for each test we have to iterate over a list of training data. In contrast, the kNN model does not take time to train. 2. Depending on the nature of each problem, we should choose the appropriate model.

Algo	**Training**	**Testing**
kNN	0	207ms
Perceptron	2.53s	52ms

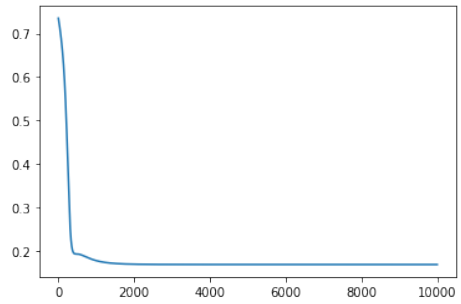
Table 2: Compare kNN and perceptron online/offline time



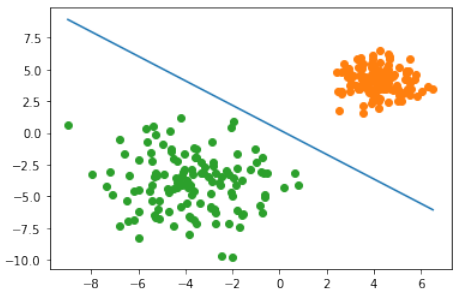
(a) Model with tanh activation loss each epoch



(b) Model with tanh activation result



(c) Model with sigmoid activation loss each epoch



(d) Model with sigmoid activation result

Figure 4: Perceptron model training and validation.

2 TP2

First I read the data and perform the split data for train and validation.

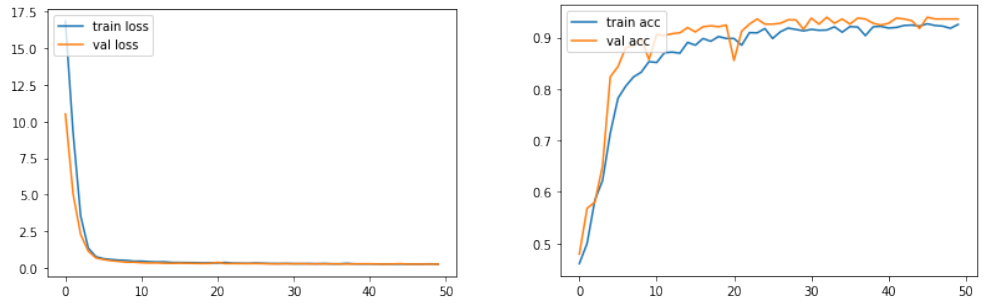
```
df = pd.read_csv("/content/Uses_Cases/Spam/Spam detection - For model creation.csv", sep
↪  = ";") # read train dataset
print(df.describe()) # summary data
y = (df["GOAL-Spam"].values == "Yes").astype(int) # get label
X = df.iloc[:,1:].values # remove label from train data
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2,
↪  random_state=42) # split dataset for validation
```

2.1 Build an M1 model based on a neural network architecture without hidden layers

Since in the previous section 1.2.2 we have installed perceptron, now for simplicity I will install with TensorFlow library. Model 1 declaration code with TensorFlow [4]:

```
m1_input = tf.keras.layers.Input(shape = (X.shape[1])) # create input layer
m1_output = tf.keras.layers.Dense(1, activation = "sigmoid")(m1_input) # output layer
↪  (no hidden layers)
m1 = tf.keras.Model(inputs = [m1_input], outputs = [m1_output]) # Model from input to
↪  output layer
```

Figure 5 shows the difference between training and validation. Thereby we can see, in the epoch (0, 30) the model is underfit and in the later stage to the 50th epoch the model is fitting. Here in the training process the model is not overfit, and the result of the model is also quite high, so if there is overfit, it is not really serious.



(a) Training and validation loss each epoch

(b) Training and validation accuracy each epoch

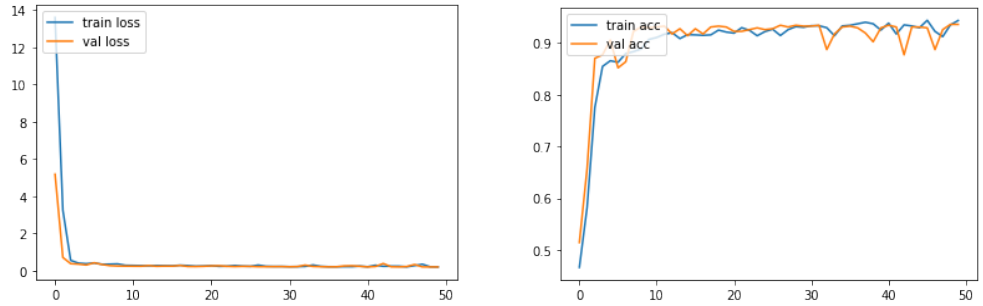
Figure 5: Model 1 training and validation.

2.2 Build an M2 model based on a neural network architecture with a hidden layer

Model 2 declaration code with TensorFlow [4]:

```
m2_input = tf.keras.layers.Input(shape = (X.shape[1])) # create input layer
m2_headModel = tf.keras.layers.Dense(16, activation='elu')(m2_input)  # one hidden
↪  layer, 16 nodes and elu fucntion
m2_output = tf.keras.layers.Dense(1, activation = "sigmoid")(m2_headModel) # output
↪  layer
m2 = tf.keras.Model(inputs = [m2_input], outputs = [m2_output])  # Model from input to
↪  output layer
```

The 2^{nd} line of this version is different from the previous version. Because I have to add a hidden layer to the model. In this case is the hidden layer with 16 nodes and the activation function is the elu function.



(a) Training and validation loss each epoch

(b) Training and validation accuracy each epoch

Figure 6: Model 2 training and validation.

Figure 6 shows the difference between training and validation. Thereby we can see, in the epoch (0, 5) the model is underfit and in the later stage to the 50th epoch the model is fitting. Here in the training process the model is not overfit, and the result of the model is also quite high, so if there is overfit, it is not really serious.

2.3 Build an M3 model based on a naive Bayes network type classifier

2.3.1 Recall the general principle

Naive Bayes is supervised learning algorithms, and have been around since the 2nd half of the 18th century. Bayes' theorem states the following relationship:

$$P\left(y|\overline{x_1...x_n}\right) = \frac{P\left(y\right)P\left(x_1...x_n|\,y\right)}{P\left(x_1...x_n\right)} = \frac{P\left(y\right)\prod_{i=1}^{n}P\left(x_i|\,y\right)}{P\left(x_1...x_n\right)}. \tag{3}$$

In case the features are linear rather than some label, we use Gaussian Naive Bayes and assumed that the likelihood of the features is to be Gaussian. Then we can compute:

$$P\left(x_i\,|y\right) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}\left(\frac{x_i-\mu}{\sigma}\right)^2} \tag{4}$$

2.3.2 Implement

My Gaussian Naive Bayes implement:

```python
class NaiveBayes():
    def __init__(self):
        self.std = {} # store std
        self.mean = {} # store mean

    def f(self, x, m, s): # Gauss prob
        return math.exp(-0.5*((x-m)/s)**2)/(s*math.sqrt(2*math.pi))

    def fit(self, X_train, y_train, var_smoothing=1e-3):
        for i in list(set(y_train)): # for each label
            self.std[i] = []
            self.mean[i] = []

            X_train_temp = X_train[y_train == i] # get X where label(X) = i
            for col in range(X_train.shape[1]): # for each column
                self.std[i].append(max(var_smoothing, np.std(X_train_temp[:, col]))) # save std of label
                self.mean[i].append(np.mean(X_train_temp[:, col])) # save mean of label

    def predict(self, X_test):
        cls = []
        for x in X_test: # for each test
            probs = []
            for label in self.mean.keys(): # for each label
                p = 1 # init log prob
                for i in range(len(self.mean[label])): # for each column
                    p *= self.f(x[i], self.mean[label][i], self.std[label][i]) # Gauss prob
                probs.append((label, p)) # append log prob of label
            cls.append(max(probs, key = lambda x: x[1])[0]) # get class with max log prob
        return cls

my_m3 = NaiveBayes() # create my Naive bayes model (base on var_smoothing, num_feaure we have difference score)
my_m3.fit(X_train, y_train) # train the model
print(np.mean(my_m3.predict(X_valid) == y_valid)) # Validation the model
```

Sklearn Gaussian Naive Bayes:

```
m3 = GaussianNB() # create Naive bayes model
m3.fit(X_train, y_train) # train the model
print(m3.score(X_valid, y_valid)) # Validation the model
```

Because we have case that standard deviation σ equal to 0.0 so we need *var_smoothing* to prevent the denominator being 0. Since this number is a user-entered constant, the algorithms can give different results like the table 3. On the other hand, sklearn's algorithms also include input feature selection algorithms $n\_features\_in\_$. Hence it can take less input than the number of existing features.

Model	Validation	Testing
My code	0.8134	0.8163
Sklearn	0.8201	0.8226

Table 3: Gaussian naive bayes performance

2.4 Evaluate the performance of the three models using the test provided

The evaluate code:

```
df_test = pd.read_csv("/content/Uses_Cases/Spam/Spam detection - For prediction.csv") #
↪  read test dataset
df_test.describe() # summary data
y_test = df_test["Spam"].values # get label
X_test = df_test.iloc[:,:-1].values # get data
m1_pred = (m1.predict(X_test) > 0.5).astype(int).flatten() # predict by m1
m2_pred = (m2.predict(X_test) > 0.5).astype(int).flatten() # predict by m2
m3_pred = m3.predict(X_test) # predict by m3
print("M1 accuracy:", np.mean(m1_pred == y_test))
print("M2 accuracy:", np.mean(m2_pred == y_test))
print("M3 accuracy:", np.mean(m3_pred == y_test))
```

Since we cannot fix the TensorFlow random seed, we will have different results each time we run M1 and M2, because the generated random value is different. Table 4 shows the performance of the model and calculated by accuracy metric. Although the results are not fixed, we can see that M1, M2 have higher results than M3. In which M2 is slightly better than M1.

Model	Run 1	Run 2	Run 3	Run 4
M1	0.9277	0.9309	0.9277	0.9317
M2	0.9451	0.9521	0.9482	0.9505
M3	0.8226	-	-	-
My M3	0.8163	-	-	-

Table 4: Testing result of model 1 (M1), model 2 (M2), moedl 3 (M3).

2.5 Conclude

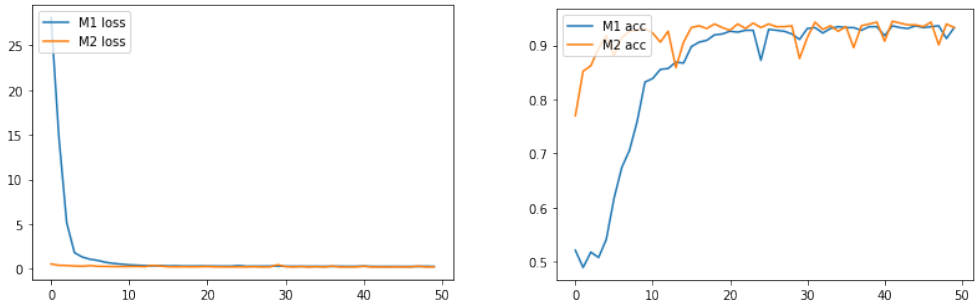
In the table 4 we can see that M1, M2 have better accuracy. However in the table 5 we can see that M1 and M2 took longer to train. Depending on the requirement of each problem, we should choose the appropriate algorithm.

We can see that the accuracy of M2 is higher than that of M1 but not too much. But M2 has a much higher complexity and capacity than M1. So let's see what is the difference between M1 and M2. Figure 7 we can see that M2 has much better than M1 in some first epochs. Prove that M2 has better learning ability than M1. After that both have roughly equal performance and almost reach the limit despite more training. Therefore, in each problem, we need to consider choosing the number

Model	Training	Testing
M1	13.2s	270us
M2	17.8s	271us
M3	13.4ms	702us
My M3	7.59ms	431us

Table 5: Compare model training and testing time

of layers of the model, increasing the number of layers **maybe** increase the accuracy of the model, but also increase the complexity significantly.



(a) Loss of M1 and M2

(b) Accuracy of M1 and M2

Figure 7: Model 1 and model 2 compare.

References

[1] E. Fix and J. L. Hodges, "Discriminatory analysis. nonparametric discrimination: Consistency properties," *International Statistical Review/Revue Internationale de Statistique*, vol. 57, no. 3, pp. 238–247, 1989.

[2] O. Kramer, "Scikit-learn," in *Machine learning for evolution strategies*, pp. 45–53, Springer, 2016.

[3] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

[5] T. Nguyen-Quang, T.-D. H. Nguyen, T.-L. Nguyen-Ho, A.-K. Duong, N. Hoang-Xuan, V.-T. Nguyen-Truong, H.-D. Nguyen, and M.-T. Tran, "Hcmus at mediaeval 2020: Image-text fusion for automatic news-images re-matching," 2020.

[6] A.-K. Duong, H.-L. Nguyen, and T.-T. Truong, "Large margin cotangent loss for deep similarity learning," in *2022 International Conference on Advanced Computing and Analytics (ACOMPA)*, pp. 40–47, IEEE, 2022.

[7] A.-K. Duong, H.-L. Nguyen, and T.-T. Truong, "Enhanced face authentication with separate loss functions," *arXiv preprint arXiv:2302.11427*, 2023.

[8] M. S. A. Toofanee, S. Dowlut, M. Hamroun, K. Tamine, V. Petit, A. K. Duong, and D. Sauveron, "Dfu-siam a novel diabetic foot ulcer classification with deep learning," *IEEE Access*, 2023.

[9] M. S. A. Toofanee, M. Hamroun, S. Dowlut, K. Tamine, V. Petit, A. K. Duong, and D. Sauveron, "Federated learning: Centralized and p2p for a siamese deep learning model for diabetes foot ulcer classification," *Applied Sciences*, vol. 13, no. 23, p. 12776, 2023.

[10] A.-K. Duong and V.-H. Huynh, "Bilinear cnns model and test time augmentation for screening viral and covid-19 pneumonia," *SSICT2020*, p. 112, 2020.

[11] A. K. DUONG, V.-H. HA, H. BARHOUMI, and A. AROUI, "Federated learning with siamese neural network," 2023.