Data Mining - 20595

Maman 22

Yehonatan Simian
206584021

Question 1 - Bayesian Inference & Supervised Learning (30%)

Section 1 - Bayesian Inference (7%)

Let's examine a Bayesian inference approach to predication of CKD using the Naïve Bayes algorithm. This approach is called "naïve" because it depends on the so-called "naïve" assumption that the different categories that classify our data-set (age, blood-pressure etc.) are independent (which is usually not true in real world). That way, we can use Bayes' rule to calculate the probability that a new patient has CKD based on it's own values in each category.

For the uninformed reader, Bayes' rule is a method to calculate conditional probability by negating the condition and use the probabilities that the conditions themselves are true. That means we can calculate the probability that a new patient has CKD based on its other classifications, by using 3 different probabilities:

- The probability that he has those classifications provided that he has CKD.
- The probability that he has those classifications.
- The probability that he has those CKD.

Each one of the three probabilities above is easy to calculate based on the information we already have, e.g. if 250 patients have CKD, among the data-set of 400 records, then the probability of having CKD is $250/400 = 0.625$.

This method is easy to implement and fast to calculate (note that the last two out of three probabilities above can be calculated only once and then reused over and over again), but its naiveness is a major downside as for things in real life seldom are independent, all the more so medical characteristics.

Section 2 - Supervised Learning (8%)

Let's examine a supervised learning approach to predication of CKD using the k-NN algorithm. This approach assumes that each and every one of the 400 records, defined in the previous assignment (Maman 21), is a point in a multidimensional space of 23 dimensions (as the 23 categories that classify our data-set). In order to implement this algorithm, the following steps must be considered:

1. **Define a distance function.** This function will define how much are two records "close" to each other. In order to find a reliable function, we need patients with similar values in the same categories to be "close" to each other. We aren't doctor, hence we don't know much about the categories from the medical perspective, thus all categories "weight" the same, i.e. they have the *same impact* on patients being "close". Also, note that it is common to normalize the categories' values before sending them to the distance function. Since we can normalize every category of the data-set, using 0 and 1 values to binary nominal categories, a ***euclidean distance function*** sounds like a good natural choice as the distance function.

2. **Define k.** This part is not trivial, as for there are many considerations to be taken, and there's no "winning technique" to find the best k value. Some of the considerations are:

   - Choosing a small value of k leads to unstable decision boundaries.
   - Choosing a large value can lead to many records that are in an ambiguous "gray area", where it's hard to decide how to classify them.
   - The algorithm can be run over and over again.

   The last bullet above was written to emphasize that *different values* of k can be chosen arbitrarily, and those who produce higher score values will be taken (we'll try to explain why).

3. Split the data-set to train and test records. Then, for each record among the test records:

   - Find its k nearest neighbors using the distance function described in step #1, where the potential neighbors are the train records.
   - Classify that record's class category (CKD or NOT-CKD) as the majority of neighbors; e.g. If $k = 3$, and two of the three nearest neighbors of a test record are classified as CKD, that test record will also be classified as CKD.

This method is easy to implement and also fast to calculate, and its major downside is the binary categories' large impact on the distance.

Section 3 - Approach Selection (5%)

I have decided to choose k-NN as a prediction method of CKD. In my opinion, this is a better than the Bayesian inference approach due to a couple of reasons:

First of all, Bayesian inference is all around relations between different categories; A Bayesian network defines the relations between different categories, but I can't build such network due to my lack of medicinal knowledge, and the naïve Bayes' major downside is its inaccuracy caused by ignoring relations between those categories.

Moreover, k-NN sounds to me like an interesting algorithm I'd like to investigate and explore, especially on the step of choosing k - Will a lower value of k yield better accuracy and precision? Will a higher value be superior? And most important - why? By running the algorithm and plotting some graphs, I might be able to answer those questions, and perhaps also acquire some new insights and understandings of the supervised learning approach.

To summarize, k-NN sounds to me like a better and also more interesting way to predict CKD, hence I have decided to select *supervised learning* to solve the given problem.

Section 4 - Execution & Results (6%)

Again, I will run the algorithm using Python's *sklearn* library, as for it already has a built-in k-NN algorithm. Furthermore, the code is pretty similar to the code from Maman21:

```python
def get_normalized_data():
    db = pd.read_csv(SRC_PATH)  # the prepared dataset from Maman21

    features = db.drop("class", axis=1)
    target   = db["class"]

    scaler = MinMaxScaler() # default scaling is 0 to 1
    normalized_features = scaler.fit_transform(features)
    # target is already normalized... only 0s and 1s

    return normalized_features, target
```

Figure 1: Data normalization process

```python
def main():
    features, target = get_normalized_data()
    train_test_split_list = train_test_split(features, target, test_size=0.33, random_state=42)

    print(f"+-------------------------------------------------------+")
    print(f"| name\t\t\t| accuracy\t| percision\t|")
    print(f"|-------------------------------------------------------|")
    for i in range(1, 10):
        train_knn(i, train_test_split_list)
    print(f"+-------------------------------------------------------+")
```

```python
def train_knn(num_of_neighbors, train_test_split_list):
    false_train, false_test, true_train, true_test = train_test_split_list

    knn = KNeighborsClassifier(num_of_neighbors)
    knn.fit(false_train, true_train)

    print_results(knn, f"k-NN, {num_of_neighbors} neighbor(s)", true_test, false_test)
```

Figure 2: Training process

```python
def print_results(model, name, true_test, false_test):
    prediction = model.predict(false_test)
    accuracy = accuracy_score(true_test, prediction)
    percision = precision_score(true_test, prediction)
    print(f"| {name}\t| {accuracy:.4f}\t| {percision}\t|")

    create_graph(true_test, prediction)
```

```python
def create_graph(true_test, prediction):
    cm = confusion_matrix(true_test, prediction)
    sns.heatmap(cm, annot=True, fmt='d')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()
```

Figure 3: Visualizing process

Question 1 continued on next page. . .

And the results:

```
C:\Users\yonis\OneDrive - Open University of Israel\OpenU
+-----------------------------------------------------------+
| name                  | accuracy      | percision         |
|-----------------------------------------------------------|
| k-NN, 1 neighbor(s)   | 0.9924        | 1.0               |
| k-NN, 2 neighbor(s)   | 0.9621        | 1.0               |
| k-NN, 3 neighbor(s)   | 0.9773        | 1.0               |
| k-NN, 4 neighbor(s)   | 0.9621        | 1.0               |
| k-NN, 5 neighbor(s)   | 0.9621        | 1.0               |
| k-NN, 6 neighbor(s)   | 0.9545        | 1.0               |
| k-NN, 7 neighbor(s)   | 0.9621        | 1.0               |
| k-NN, 8 neighbor(s)   | 0.9621        | 1.0               |
| k-NN, 9 neighbor(s)   | 0.9621        | 1.0               |
+-----------------------------------------------------------+
```

Figure 4: Accuracy & Precision

I've also added the confusion matrices of the highest / lowest scored neighbors, i.e. $k = 1, k = 5$:

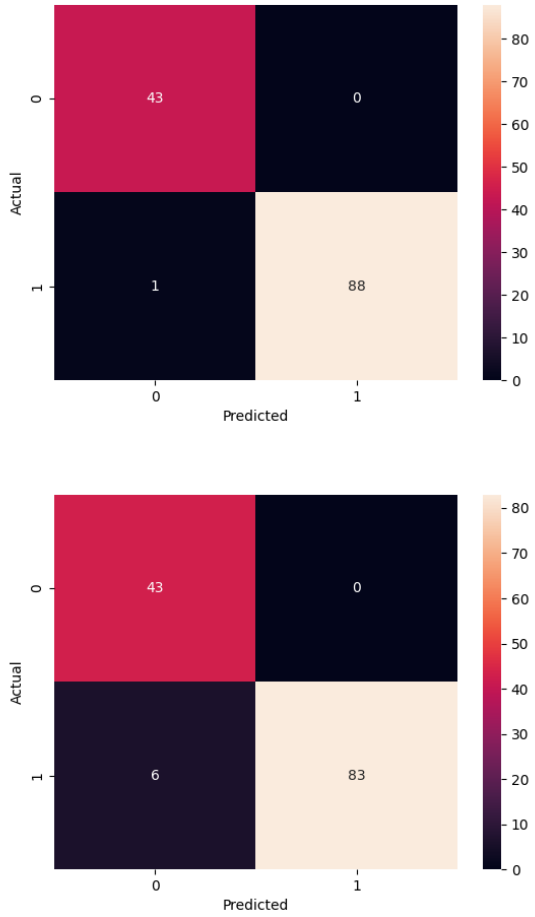


Figure 5: Confusion Matrices

Section 5 - Analysis & Conclusions (6%)

First of all, the results are pretty good, and surely are better than those in Maman21 (reminder: highest accuracy was 0.97, highest precision 0.98, with the same parameters for sklearn's train_test_split). Moreover, we can notice that the best results were produced when $k = 1$ with accuracy of 0.9924, and the rest converges pretty fast to accuracy of 0.9621.

These results lead me to the following conclusions:

- k-NN was a good choice for prediction of CKD, hopefully because of the reasons I mentioned in Section 3.
- Higher values of k produced lower scores; Perhaps the reason is the low amount of records in the data-set, which lead to including "far" neighbors together when a point is lack of near neighbors (low-density areas).
- As a result of the previous conclusion, I'm even more curious to see the results of DBSCAN on this data-set, the algorithm that I'll chose on Question 2, which should classify such low-density areas as *noise* and by that overcome this disadvantages of k-NN and other supervised learning approaches.

Figure 6: Sonic the Hedgehog holding a guitar

Question 2 - Cluster Analysis (30%)

Section 1 - Evaluation Criteria (2%)

Cluster evaluation criteria can be defined using two main different types:

1. **Internal evaluation** - each clustering result is evaluated based on the very own clustered data itself. This method usually assigns the best score to the algorithms that produce clusters with high similarity within a cluster and low similarity between different clusters.

2. **External evaluation** - each clustering result is evaluated based on data that was *not* used for clustering, such as known class labels and external benchmarks.

In other words, internal evaluation summarizes the clusters to a single quality score, and external evaluation compares the clusters to an outer known classification.

Section 2 - Approach Selection (5%)

I have decided to choose DBSCAN (Density-based spatial clustering of applications with noise) as a prediction method of CKD. This method is the most popular among density based clustering methods, which rely on the principal that records with the same classification form areas with higher density, thus a density-based cluster can be formed by ”connecting” close records. Those clusters can be recognized easily by iterating over the records, and recursively bind them to clusters (or mark them as ”noise”), somewhat similar to find connected components in a graph.

Note that, similar to k-NN , a distance function must be chosen. Furthermore, records are resembled to points in a multidimensional space, therefore these similarities can help us compare the two methodologies (k-NN and DBSCAN), and perhaps discover interesting insights about this kind of classification. In other words, these two methods look to me like the same approach from different perspectives (supervised and unsupervised). The main difference between the two methods is that DBSCAN can recognize nontrivial ”shapes” of data in the multidimensional space of records, ”shapes” that cannot be observed using k-NN like ”crescent moon shape” and ”doughnut shape”.

The informed reader might notice that I have not mentioned the k-means algorithms, which might sound inappropriate when exploring the unsupervised equivalent of k-NN ; The reason is that DBSCAN overcomes k-NN ’s downside of having to arbitrarily choose k. DBSCAN also overcomes the noises that affect algorithms such as k-means and k-NN , hence it is my choice of unsupervised CKD prediction algorithm.

Section 3 - Cluster Analysis Steps (13%)

The steps for cluster analysis, using my chosen DBSCAN method, are:

1. **Data preparation** - this has already been done and explained on Maman 21. I will use the same data-set that was already prepared by me.

2. **Distance function definition** - as mentioned in Question 1 Section 1, the *euclidean distance function* is a good natural choice for my purposes, hence it'll be chosen as the distance function for both k-NN and DBSCAN algorithms. Note that binary categories will be treated as 0 and 1, as described in the distance function definition.

3. **Parameters definition** - indeed the chosen distance function is the *euclidean distance function*, which takes two points in any dimensional space and calculates the distance between them, but additional parameters are needed for our function as well. This is because calculating the distance is not enough; Our function must also determine if that distance is sufficient for the two points to be in one cluster. Furthermore, DBSCAN can detect noise - records that not belong to any cluster. To summarize, the distance function must take two parameters:

   - ε - Maximal distance for two point in a cluster.
   - MinPts - Minimal number of points to be considered a cluster.

   Different values of those parameters can (and will be) explored during the execution.

4. **Cluster formation** - the algorithm is pretty simple:

   1) For every record that has not been marked yet:
   2) Recursively find the close records according to ε (similar to BFS, DFS).
   3) If their amount exceeds MinPts, mark them all as a unique cluster.
   4) Otherwise, mark them as noise.

   Note that core and border points are not mentioned in the algorithm. This is because of my decision to follow the algorithm description as shown in class, which is a bit different than the one taught in the books. This way or another, the idea of forming clusters is clear, so we can proceed to the next step.

5. **Results analysis** - this will be explained in detail in the next two sections; But as part of the algorithm itself, it's good to mention that visualization of the results can't help us this time to find better results. This is because there are too much features in the data-set, causing the visualization to be in a 23-dimensional graph or giving up features in order two visualize only 2 features in a 2-dimensional graph - two options that I don't like. In other words, this time the are two main differences in evaluating results:

   1) We're not gonna split the data to "train" and "test", and that's because...
   2) Evaluating unsupervised clustering algorithms like DBSCAN can be subjective and dependent on the specific characteristics of my data-set and the goals of my analysis.

   The second difference lead me to use another measurements rather than accuracy and precision, as for they are not relevant on unsupervised clustering algorithms:

   1) **Silhouette**: the most common score for clustering algorithms. Ranges from -1 to 1, where higher values are better, but there's no fixed threshold that defines a "good" score.

2) **Calinski-Harabasz** together with

3) **Davies-Bouldin**: these two scores have no fixed range, but in general a higher value suggests well-separated and compact clusters, indicating a better clustering solution.

With that being said, I can now proceed to the code and its execution results.

Section 4 - Execution & Results (6%)

Again, the code is pretty similar to last execution, even with the two new differences:

```python
def main():
    db = pd.read_csv(SRC_PATH) # the normalized dataset from last question

    scaler = MinMaxScaler()
    db_scaled = scaler.fit_transform(db)

    for i in range(1, 6):
        eps = i / 10
        for min_samples in range(3,10,2):
            print(f"+--------------------------------+")
            print(f"| Epsilon: {eps}\t| MinPts: {min_samples}\t|")
            print(f"|--------------------------------|")
            train_dbscan(eps, min_samples, db_scaled)
        print(f"+--------------------------------+")
```

```python
def train_dbscan(eps, min_samples, db_scaled):
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)
    dbscan.fit(db_scaled)
    print_results(dbscan, db_scaled)

def print_results(dbscan, db_scaled):
    labels = dbscan.labels_
    num_clusters = len(set(labels)) - (1 if -1 in labels else 0)
    num_noise_points = list(labels).count(-1)

    print(f"| There are {num_clusters} clusters\t\t|")
    print(f"| There are {num_noise_points} noise points\t|")
    if num_clusters == 0:
        print(f"| Scores aren't relevant\t|")
        print(f"| when there are 0 clusters.\t|")
    else:
        silhouette_avg = silhouette_score(db_scaled, labels)
        CH_score = calinski_harabasz_score(db_scaled, labels)
        DB_score = davies_bouldin_score(db_scaled, labels)
        print(f"| Silhouette: {silhouette_avg:.3f}\t\t|")
        print(f"| Calinski-Harabasz: {CH_score:.1f}\t|")
        print(f"| Davies-Bouldin: {DB_score:.2f}\t\t|")
```

Figure 7: DBSCAN python code

And the results:

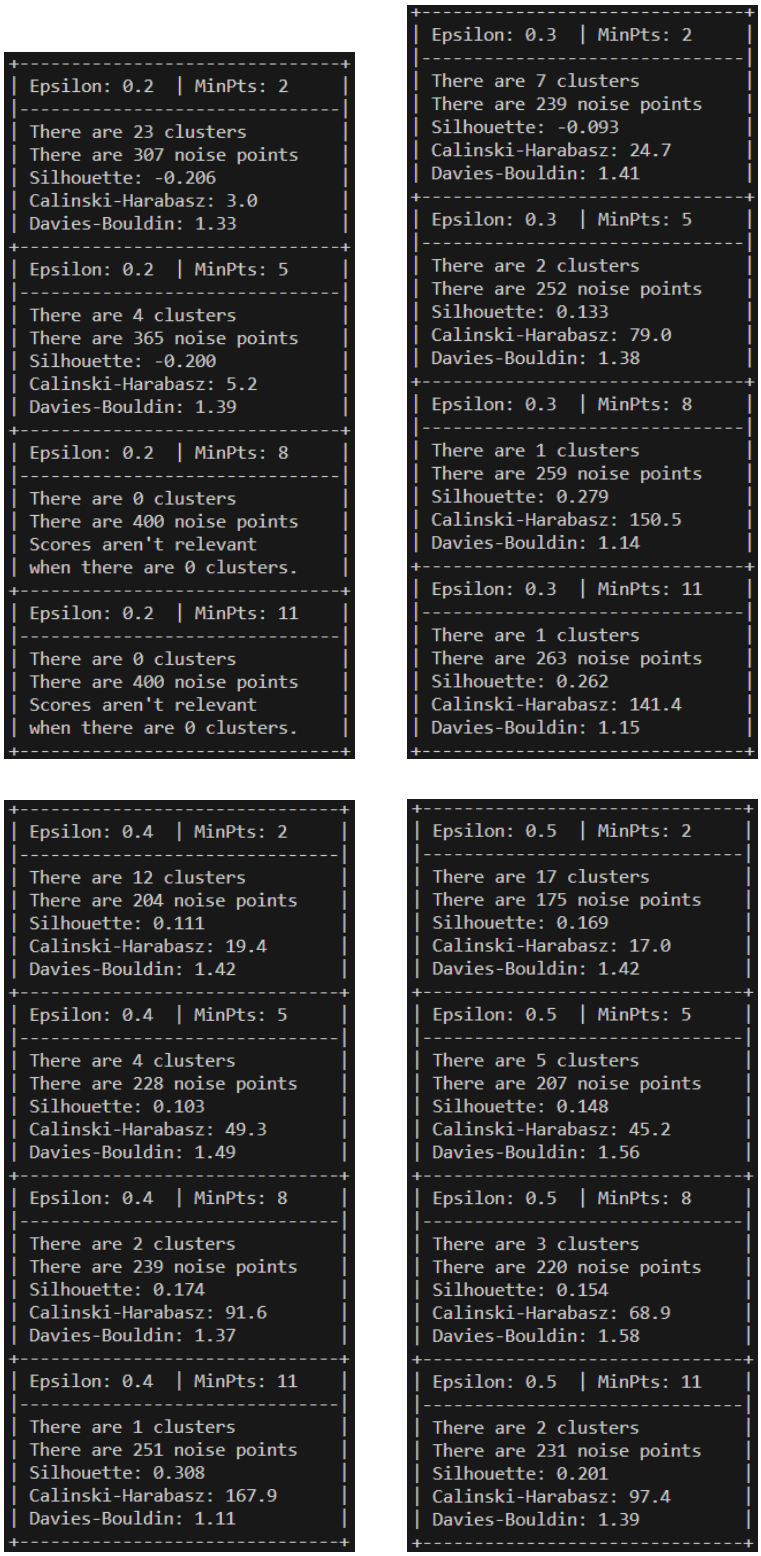


Figure 8: DBSCAN results

I dropped the eps=0.1 results because they where all 0 clusters. Actually, as can be seen, higher values of epsilon have produced better results in general (in some cases, eps=0.4 produces better than 0.5). Also, although Davies-Bouldins' score was pretty stable over the entire run (with eps=0.5 being slightly better), the Calinski-Harabasz score rose much higher with higher values of MinPts.

This caused me to want to explore some higher values of epsilon and MinPts, so I altered the code a bit:

```
32  def main():
33      db = pd.read_csv(SRC_PATH) # the normalized dataset from last question
34
35      scaler = MinMaxScaler()
36      db_scaled = scaler.fit_transform(db)
37
38      for i in range(3, 10, 2):
39          eps = i / 10
40          for min_samples in range(10, 31, 10):
41              print(f"+--------------------------------+")
42              print(f"| Epsilon: {eps}\t| MinPts: {min_samples}\t|")
43              print(f"|--------------------------------|")
44              train_dbscan(eps, min_samples, db_scaled)
45          print(f"+--------------------------------+")
```

Figure 9: Altered DBSCAN python code

And here are the new results:

```
+--------------------------------+
| Epsilon: 0.5  | MinPts: 10     |
|--------------------------------|
| There are 2 clusters           |
| There are 231 noise points     |
| Silhouette: 0.201              |
| Calinski-Harabasz: 97.4        |
| Davies-Bouldin: 1.39           |
+--------------------------------+
| Epsilon: 0.5  | MinPts: 20     |
|--------------------------------|
| There are 1 clusters           |
| There are 251 noise points     |
| Silhouette: 0.308              |
| Calinski-Harabasz: 167.9       |
| Davies-Bouldin: 1.11           |
+--------------------------------+
| Epsilon: 0.5  | MinPts: 30     |
|--------------------------------|
| There are 1 clusters           |
| There are 251 noise points     |
| Silhouette: 0.308              |
| Calinski-Harabasz: 167.9       |
| Davies-Bouldin: 1.11           |
+--------------------------------+
```

```
+--------------------------------+
| Epsilon: 0.7  | MinPts: 10     |
|--------------------------------|
| There are 3 clusters           |
| There are 209 noise points     |
| Silhouette: 0.185              |
| Calinski-Harabasz: 73.4        |
| Davies-Bouldin: 1.67           |
+--------------------------------+
| Epsilon: 0.7  | MinPts: 20     |
|--------------------------------|
| There are 2 clusters           |
| There are 225 noise points     |
| Silhouette: 0.221              |
| Calinski-Harabasz: 101.9       |
| Davies-Bouldin: 1.41           |
+--------------------------------+
| Epsilon: 0.7  | MinPts: 30     |
|--------------------------------|
| There are 1 clusters           |
| There are 251 noise points     |
| Silhouette: 0.308              |
| Calinski-Harabasz: 167.9       |
| Davies-Bouldin: 1.11           |
+--------------------------------+
```

```
+--------------------------------+
| Epsilon: 0.9  | MinPts: 10     |
|--------------------------------|
| There are 4 clusters           |
| There are 190 noise points     |
| Silhouette: 0.210              |
| Calinski-Harabasz: 62.5        |
| Davies-Bouldin: 1.66           |
+--------------------------------+
| Epsilon: 0.9  | MinPts: 20     |
|--------------------------------|
| There are 2 clusters           |
| There are 223 noise points     |
| Silhouette: 0.227              |
| Calinski-Harabasz: 103.2       |
| Davies-Bouldin: 1.43           |
+--------------------------------+
| Epsilon: 0.9  | MinPts: 30     |
|--------------------------------|
| There are 1 clusters           |
| There are 251 noise points     |
| Silhouette: 0.308              |
| Calinski-Harabasz: 167.9       |
| Davies-Bouldin: 1.11           |
+--------------------------------+
```

Figure 10: New DBSCAN results

These results are indeed better, and I'll try to explain why on the next section.

Question 2 continued on next page. . .

Section 5 - Analysis & Conclusions (6%)

Overall, the results were pretty... bad. Which kinda makes me sad, because I thought DBSCAN would overcome k-NN 's weaknesses. The best scores were produced when epsilon ranged between 0.4 to 0.9, and MinPts were generally high. Moreover, the number of noise points was way too high than I expected, with *all* different values of epsilon and MinPts that I tried.

It is worth mentioning that the highest result was *too* similar in different situation - 4 different epsilon values with 3 different MinPts values have produced the *exact same* scores on all three evaluation measurements. I could think that my code has flows with it, but after reviewing it again and consulting online, it seemed pretty fair to me, so the only guess left is... the data-set was too small.

So, the first conclusion is that DBSCAN might not be a good algorithm to run on small data-sets. I hoped for it to overcome the noise that k-NN suffered from, but eventually this noise was way too high for DBSCAN to deal with, and it lowered the results to pretty low scores (compared to the 0.99 accuracy that k-NN have produced).

Furthermore... well, there is no furthermore this time. I can't really know what were the causes for these low scores. Plotting graphs is possible when separating features, but it's much work for a relatively small section of this assignment. Same for exploring different values of epsilon and MinPts.

At this point, all that's left to say is:

Figure 11: Sonic the Hedgehog with weird... hands?

Question 3 - Artificial Neural Network (30%)

Section 1 - Architecture Definition (5%)

I have chosen to use a Feed-forward Neural Network. In order to define a neural network's architecture, one must have a few considerations:

- **Neurons connection** - neural networks are defined using neurons in layers. Two layers are very clear, and so is the number of neurons in those layers:

  1. The input layer must consist of 23 neurons, as the 23 categories we have in our data base; This means there will be one input neuron that represents the normalized age value of a patient, ranged from 0 to 1, one input neuron to represent the normalized blood pressure value, and so on.
  2. The output layer must consist of 1 neuron whose value represent the probability that the patient has CKD.

  The rest of the layers (a.k.a. "hidden layers"), and the number of neurons in each layer, can be chosen arbitrarily. These number are far more flexible, i.e. different numbers can produce different results, and there is no one perfect answer. As been said before - we are not doctors, and my knowledge in medicine is very minimal; Thus these numbers will be chosen arbitrarily without much explanation. I have chosen *2 layers* with *128 neurons* in each layer. The reason is because this numbers are beautiful (I'm a musician, and musicians *love* powers of 2).

- **Information flow** - the information will be passed from each neuron to each neuron in the following layer, meaning the network will be *fully connected*. It might be a good reason to disconnect some neurons; For example, different neurons in the hidden layers might represent the blood quality (as the result of different neurons input neurons that represent blood qualities such as blood pressure and sugar), so these neurons should be disconnects from the neurons that represent age and appetite; However, this is much work for a work that is not about medicine, so I'll leave that aside.

- **Activation function** - there are a few activation functions that can be chosen for this network. It has been said that when a specific mathematical model is lacking, a sigmoid function is often used; Since I indeed lack of mathematical model, the *sigmoid function* will be my choice of this network's activation function.

Note that I haven't talked about the weights and the biases of each neuron; Those will start at random and will be redefined during the learning process.

Section 2 - Parameters Definition (5%)

The ~~optimization~~ learning process must rely on the following parameters:

- **Cost function** - again, an arbitrary choice must be done, since there is no one perfect answer for the cost function's identity. So... any volunteers in the audience? C'mon I wanna see more hands raised! Hmmm, let's see... well... wait... I think... great, over there! You, little function, what's your name? *Mean Squared Error*, did I spell it correctly? Wonderful, nice to meet you Mean Squared Error. Can I call you MSE? Superb. You seem like a descent function, I hope you don't mind sharing your formula with us. Did I here you say $\frac{1}{2}\sum_i \left(a_i - E_i^r\right)^2$ where a_i is the result of the activation function on the i-th neuron and E_i^r is the desired output of some training sample r? How delightful. I can assume that your gradient on the sample r must be $\nabla_a = (a - E^r)$. Oh, you little chicky function, I'm use we're gonna use you a lot! Thank you for volunteering, now let's proceed for the next parameter.

  NOTE Only after writing this cute little paragraph, i.e. when I wrote the code itself, I have noticed that sklearn's MLPClassifier doesn't even give an ability to choose the cost function. So I investigated a bit and found out that "For binary classification with MLPClassifier, the model uses the **log_loss** function as the default loss function. It corresponds to logistic regression with cross-entropy loss.". *Moreover, the MSE function is not used in MLPClassifier, but it does in MLPRegressor. That being said, I'll trust sklearn to know better than me what cost function is better, as for it was written by professionals who know better than me.*

- **Batch size** - since our dataset is rather small (only 400 records that sum up to less than a mega-byte of information), a batch learning would be optimal for our purpose, as it does not suffer from its main disadvantage; Thus makes the advantages of on-line and mini-batch learning redundant. In other words, there will be only one batch that will consist of the entire train portion of the dataset.

- **Learning rate** - Slowwwwwwww. I mean... hi. Due to the small size of our dataset, a slow learning rate should lead to good results in a descent amount of time, and not suffer from the main disadvantage of slow learning rates. As for the actual number - I'll play with it along the different executions. In my opinion, 0.0069 to 0.0420 seem like a... *nice* learning rate range.

Section 3 - Execution & Evaluation (10%)

Again, Sklearn came to the rescue when I searched for a neural network implementation in Python:

```python
def main():
    db = pd.read_csv(SRC_PATH) # the normalized dataset from last question

    features = db.drop("class", axis=1)
    target   = db["class"]

    train_test_split_list = train_test_split(features, target, test_size=0.33, random_state=42)

    print(f"+-----------------------------------------------------+")
    print(f"| learning rate\t\t| accuracy\t| precision\t|")
    print(f"|-----------------------------------------------------|")
    for i in range(4):
        rate = (69 + i * 117) / 100000 # magic numbers oohhhhhh
        train_neural_network(rate, train_test_split_list)
    print(f"+-----------------------------------------------------+")
```

```python
def train_neural_network(rate, train_test_split_list):
    false_train, false_test, true_train, true_test = train_test_split_list

    model = MLPClassifier(
        hidden_layer_sizes=(128, 128),  # two layers
        activation='logistic',          # sigmoid function
        solver='adam',                  # gradient descent
        batch_size=int(400*0.67),       # no mini-batch
        learning_rate_init=rate,        # learning rate
        max_iter=1000,                  # number of ephocs
        random_state=42,                # random weights
    )
    model.fit(false_train, true_train)

    print_results(model, rate, true_test, false_test)
```

```python
def print_results(model, rate, true_test, false_test):
    prediction = model.predict(false_test)
    accuracy = accuracy_score(true_test, prediction)
    percision = precision_score(true_test, prediction)
    print(f"| {rate:.5f}\t\t| {accuracy:.4f}\t| {percision}\t\t|")

    show_loss_curve(model)
    show_confusion_matrix(true_test, prediction)
    show_roc_curve(model, true_test, false_test)
```

Figure 12: Neural network python code

Question 3 continued on next page. . .

This time, three visualization types will be generated:

```python
def show_loss_curve(model):
    plt.plot(model.loss_curve_)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss Curve')
    plt.show()
```

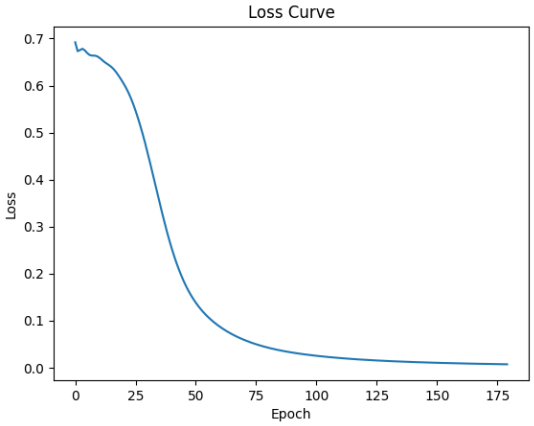
```python
def show_confusion_matrix(true_test, prediction):
    cm = confusion_matrix(true_test, prediction)
    sns.heatmap(cm, annot=True, fmt='d')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()
```

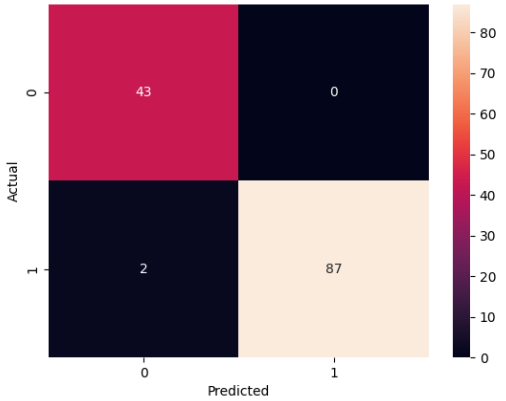
```python
def show_roc_curve(model, true_test, false_test):
    y_pred_proba = model.predict_proba(false_test)[:, 1]
    fpr, tpr, thresholds = roc_curve(true_test, y_pred_proba)
    auc_score = roc_auc_score(true_test, y_pred_proba)

    plt.plot(fpr, tpr, label='ROC Curve (AUC = {:.2f})'.format(auc_score))
    plt.plot([0, 1], [0, 1], 'k--')  # Plotting the random guess line
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()
```

Figure 13: pyramided python visualization code

And the results:

learning rate	accuracy	precision
0.00069	0.9848	1.0
0.00186	0.9848	1.0
0.00303	0.9848	1.0
0.00420	0.9848	1.0





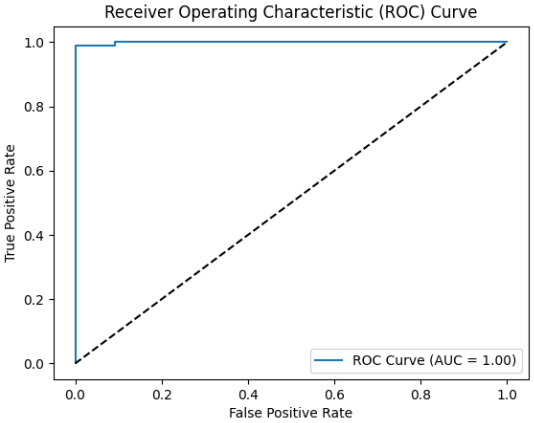


Figure 14: Neural network results

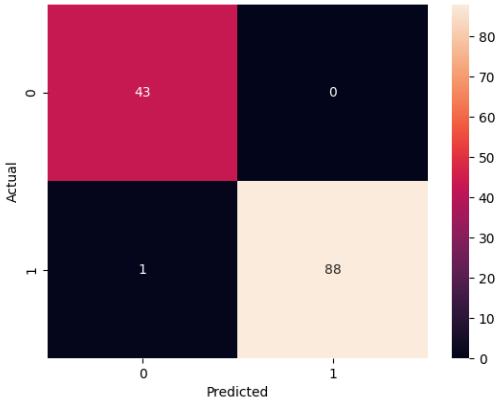
One specific line in MLPClassifier's documentation drew my attention: "Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better."

Since my dataset is relatively small, I have decided to use lbfgs (Limited-memory BFGS) solver instead of gradient descent in order to see if it yields better results. I've altered my code a bit:
And the results were indeed better:

```python
def train_neural_network(rate, train_test_split_list):
    false_train, false_test, true_train, true_test = train_test_split_list

    model = MLPClassifier(
        hidden_layer_sizes=(128, 128),  # two layers
        activation='logistic',          # sigmoid function
        solver='lbfgs',                  # Limited-memory BFGS
        # batch_size=int(400*0.67),       # no mini-batch in lbfgs
        # learning_rate_init=rate,        # no learning rate in lbfgs
        max_iter=1000,                  # number of ephocs
        random_state=42,                # random weights
    )
    model.fit(false_train, true_train)
```

Figure 15: Neural network code using lbfgs

learning rate	accuracy	precision
0.00069	0.9924	1.0
0.00186	0.9924	1.0
0.00303	0.9924	1.0
0.00420	0.9924	1.0



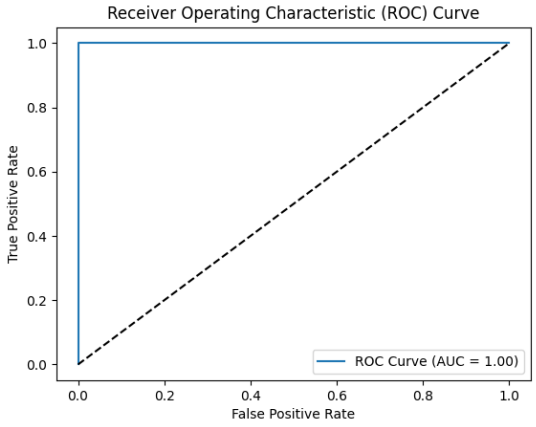


Figure 16: Neural network results using lbfgs

Note that loss curve does not appear on Figure 16, as for it is not relevant if we're not using gradient descent. Further analysis on Section 5; But first:

Question 3 continued on next page. . .

Section 4 - Wrong Classification Report (6%)

In order to find exceptional cases where a wrong classification has been made, I've added a new Python function:

```
83  def print_incorrect_results(prediction, true_test):
84      incorrect_indices = np.where(prediction != true_test)[0]
85      for record in incorrect_indices:
86          print(record)
```

Figure 17: Wrong classification detection code

I've ran it on both *adam* and *lbfgs* solvers, using all of the different learning rates that have been shown in the previous section.

Two specific records have been detected on the 'adam' solver, records #24 and #63, and one of them (record #24) was also detected on the 'lbfgs' solver.

Looking at their categories' values, I haven't noticed anything unusual. Record #24 for example had some very similar values to record #23's values.

Section 5 - Result Analysis & Conclusions (10%)

First of all, the results were really good! My first conclusion is that reading the documentation is important. If I wouldn't read sklearn's documentation, I wouldn't know about lbfgs (which was much faster than adam) and wouldn't be able to receive such good accuracy and precision results. Which leads me to the second part of my analysis:

Using the lbfgs solver I was able to receive the exact same accuracy and precision values that have been received using k-NN with one neighbor. Knowing that the number 0.9924 was received by having one wrong classification, I had to go back to the k-NN code and see which record was wrongly classified, and... ***et voilà!*** The unruly record was #24. So majestic, the very last classification algorithm I tried gave the same results as the very first, having the same unruly record; We've come full circle, the circle of life I tell ya... so symbolic.

Now to analysing the graphs: the **loss curve** had converged rather fast. Although the default number of ephocs given by sklearn is 200, changing it to 1000 didn't matter at all as for it converged even before.

The three results graphs, shown on Figure 14, where the same graphs for all of the 4 learning rates. It seems that I've chosen slow enough learning rates that could converge pretty fast, even though they all suffered from wrong classifications of records #24 and #63. Changing the learning rate parameter did not alter the result graphs, that's why I did not add a set of graphs to each learning rate; But changing the size of the layers did make the lost curve to converge faster (which makes sense). I did not add that graph because I wanted to keep the assignment clean and focused, with *learning rate* being the only parameter whose different values I explored and shown as a visualization of my neural network execution.

Question 3 continued on next page. . .

Let's move on; I think that the most important conclusion in this assignment comes from the fact that almost every algorithm that I used, on both Maman 21 and Maman 22, suffered from one main issue: the dataset was *too small*. Indeed, every machine learning algorithm must have enough data to train on in order to yield good results on testing data; Hence the **number one priority** in searching for a classification algorithm for such a small dataset, is choosing an algorithm that can deal with small datasets.

Any other conclusions that I thought about were already explored on Maman 21, therefore will be detailed more on the next and final question of this assignment, Question 4. With that being said, were ready to see the final figure of Sonic the Hedgehog for this assignment, and go ahead to the final Question where we summarize the entire project, compare that two Mamans and draw the final conclusions of the entire saga.

Figure 18: Sonic the Hedgehog ~~playing~~ being a guitar

Question 4 - Summary and Conclusions (10%)

Let's start with a poem:

Data Mining Poem / Y.S

What could I tell, what to unfold,
Data mining storied that yet to be told;
With algorithms as my loyal steed,
I rode through fields of knowledge's seed.

Decision trees, towering and grand,
Like nature's own creations, they stand;
Supervised learning, a guiding light,
Leading me through the dark of night.

Clustering and association, their paths I traced,
Seeking order in chaos, seeking grace;
And behold, the neural networks arose,
A symphony of nodes, a web that grows.

Yet, as I delved into this data maze,
With each question asked, a new one plays.
What truth lies hidden in these coded strings?
What knowledge emerges when the algorithm sings?

Sorry for the bad image quality, I tried over an hour to insert it as a part of the documents itself, but LaTeX is kinda hard to control when new interesting fonts are involved, so I had to insert it as an image instead.

So, let's compare the results of Maman 21 and Maman 22. We'll start by listing them, from best to worst:

Algorithm	Best Accuracy	Best Precision	Comments
Neural Network	0.992	1.0	lbfgs
k-**NN**	0.992	1.0	1 neighbor
Decision Tree #1	0.985	0.989	Cart + random forest
Decision Tree #2	0.969	0.977	C4.5 + random forest
DBSCAN	-	-	Bad results in general

Now, the main conclusion has been said in this assignment over and over - 400 records is **not** a big enough dataset to train on! I repeat it because it's very important, and relies on facts that have been known for decades; We must not forget that machine learning and specifically neural network algorithms have been known for decades, but only recently have they become so common, due to larger datasets being more available, along with greater computing power to rely on.

Talking about computing power - this small dataset did not suffer from long executions of the code (although neural networks with gradient descent *did* produce some noticeable execution delays), but in larger datasets this **must** come as a consideration when choosing a classification algorithm.

Now to measuring results - because this part is not trivial at all; I have used accuracy and precision and my main objective when trying to yield better results, but not necessarily this is the best answer. On decision trees and neural networks it was very easy and convenient to check the model's success - we just split the dataset to train and test records, then trained the former and tested the latter records separately, knowing that one specific outcome must be provided by the model. This was **not** the same for the unsupervised DBSCAN algorithm - where classifying on the trained model was not so unambiguous. There are a few reasons for that, some of them are also related to k-NN :

- How can we decide what's "close" and what's "far" in terms of records similarity?
- Even when experimenting different parameters values and finding some values that yield better results - how would we know if those values fit larger datasets?
- The last ambiguousness regards DBSCAN only, as for it suffers a problem no other algorithm has, among the algorithms that we explored - what do the clusters represent? Unlike the other algorithms, it is not promised that clusters represent records with the same classification - which leads for accuracy and precision being irrelevant score for this algorithm.

As a personal expression, I can say that neural network and k-NN were my favorite algorithms among all those whom I explored; Maybe their higher scores where not only a result of them fit better to our dataset, but also a result of my *better understanding* of them. Either way, I think it's time for our last meme section. **Until the next time!**

Question 4 continued on next page. . .

Figure 19: Data science memes... for the final time :)