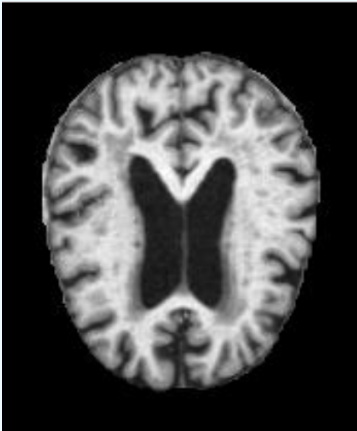


# RAPPORT

# PROJET

# ML

Nom du projet	Projet Machine Learning : Classification d'image	Date du rapport 05/04/2023
Description	Le but de notre projet est de concevoir un algorithme d'apprentissage supervisé afin de déterminer le stade de la maladie d'Alzheimer de différents patients	
Préparé par	<ul style="list-style-type: none"><li>• Amine EL GAOUZI</li><li>• Jérémy LUCAS</li><li>• Matthieu SFEIR</li><li>• Oumaima BOUNHAR</li></ul>	

## Sommaire

1. Les données
2. Les différents modèles utilisés
3. Arbres de décision
4. Random forest
5. Réseau de neurones
6. Méthodes KNN
7. Comparaisons des différents modèles

## Les données .1

Les données que nous utiliserons proviennent de la plateforme KAGGLES, une communauté en ligne dédiée à l'apprentissage automatique et à la science des données.

Nous avons utilisé un ensemble de données contenant des images d'IRM de cerveaux classées selon quatre niveaux de démence d'Alzheimer.

Le premier niveau de démence est appelé “**Mild Demented**”. Dans cet état, le patient commence à oublier les mots et les choses qu'il doit faire. Il a également des problèmes de concentration et de travail. Ce stade est souvent diagnostiqué à l'aide de tests de capacité cognitive. L'ensemble de données mis à notre disposition pour ce niveau de démence contenait 717 images d'entraînement et 179 images.

Le deuxième niveau de démence est appelé “**Moderate Demented**”. À ce stade, le patient commence à oublier les activités qu'il a effectuées récemment, ainsi que les histoires anciennes. Il peut avoir des difficultés à sortir seul et perdre son empathie. L'ensemble de données contient 52 images d'entraînement et 12 images de test pour ce niveau de démence.

Le troisième niveau de démence est appelé “**Very Mild Demented**”. À ce stade, le patient commence à oublier où il a déposé ses affaires et les noms des autres. Il est difficile de détecter si le patient est “Very Mild Demented” à l'aide d'un test de capacité cognitive. L'ensemble de données contient 1792 images d'entraînement et 448 images de test pour ce niveau de démence.

Enfin, le dernier niveau de démence est appelé “**Non Demented**”. Comme son nom l'indique, le patient n'est pas atteint de la maladie d'Alzheimer. L'ensemble de données contient 2560 images d'entraînement et 640 images de test pour ce niveau de démence.

Au total, nous avons à notre disposition 5121 images d'entraînement et 1279 images de test. Chaque image est normalisée en taille et en position. Les images sont de taille 176 x 208 pixels et centrée entre hémisphère gauche et droite.

Pour entraîner et tester les différents modèles, les images ont été transformées en matrices numpy 176x208 nuance de gris. La précision des modèles a été calculée en utilisant la méthode de cross-validation vu en cours avec cinq plis. Cela signifie que les données d'apprentissage sont divisées en cinq sous-ensembles (ou “folds”) de taille égale. Le modèle d'apprentissage est ensuite entraîné sur quatre de ces sous-ensembles et testé sur le cinquième sous-ensemble. Cette opération est répétée cinq fois, de manière à ce que chaque pli soit utilisé comme ensemble de test une fois. Les scores de performance (par exemple, l'exactitude ou la précision) sont enregistrés pour chaque itération et utilisés pour évaluer la performance globale du modèle.

## Les différents modèles comparés .2

---

Nous souhaitons comparer différents modèles de classification, ici appliqués au cadre des images. Pour cela, nous avons quatre modèles : les arbres de décisions, la méthode Random

forest, les réseaux de neurones et le modèle KNN.

Au cours de ce rapport, nous présenterons en détails chacun de ces modèles de classification.

## Arbres de décisions .3

---

Nous avons commencé par utiliser un arbre de décision sur notre modèle.

Voici l'implémentation de l'arbre de décision :

```
IMAGE_SIZE = [176, 208]
class_map = {0:"MildDemented",1: "ModerateDemented",2:"NonDemented",3:"VeryMildDemented"}
image_dirs = ["AlzheimersDataset\\train\\MildDemented", "AlzheimersDataset\\train\\ModerateDemented", "AlzheimersDataset\\train\\NonDemented", "AlzheimersDataset\\train\\VeryMildDemented"]
# Chargement des images et des labels d'entrainements
X_train = []
Y_train = []
for i, image_dir in tqdm(enumerate(image_dirs)):
    for file_name in os.listdir(image_dir):
        if file_name.endswith(".jpg"):
            image_path = os.path.join(image_dir, file_name)
            image = Image.open(image_path)

            image = image.resize((IMAGE_SIZE[0], IMAGE_SIZE[1]))
            X_train.append(np.array(image))
            Y_train.append(i)
```

Ce code est écrit en Python et il s'agit d'un script pour charger et prétraiter des images d'IRM de cerveau provenant du jeu de données Alzheimer de Kaggle.

Tout d'abord, nous devons charger et prétraiter les données provenant de Kaggle. Pour cela, nous avons utilisé les bibliothèques python suivantes : os ( permet d'accéder au fichier) , numpy (permet de stocker les images sous format de tableau de pixel) , matplotlib.pyplot (permet de visualiser les données) , PIL (permet de prétraiter les données).

Nous accédons dans un premier temps aux images grâce à la variables images\_dirs contenant les paths, puis grâce à PIL nous chargeons les images prétraitées dans la liste X\_train et leur label dans la liste Y\_train. Nous avons effectué la même opération pour les données à tester stockées dans X\_test et Y\_test.

Ensuite nous entraînons la base de données sur le modèle de la bibliothèque sklearn :

```
54 #Création de l'arbre de décision
55 clf = DecisionTreeClassifier()
56
57 # Entraînement de l'arbre de décision
58 clf.fit(X_train.reshape(len(X_train), -1), Y_train)
59
60 # Prédire les classes des images de test
61 Y_pred = clf.predict(X_test.reshape(len(X_test), -1))
62
63 # Calculer l'accuracy
64 accuracy = accuracy_score(Y_test, Y_pred)
65 print("Accuracy Binarytree: {:.2%}".format(accuracy))
66
67 # Créer un estimateur de l'arbre de décision
```

```

68 clf_2 = DecisionTreeClassifier()
69
70 # Effectuer la validation croisée sur les données d'entraînement
71 scores_clf = cross_val_score(clf_2, X_train.reshape(len(X_train), -1), Y_train, cv=10)
72
73 # Calculer la précision moyenne de la validation croisée
74 print("Accuracy Binarytree cross validation: {:.2%}".format(scores_clf.mean()))
75

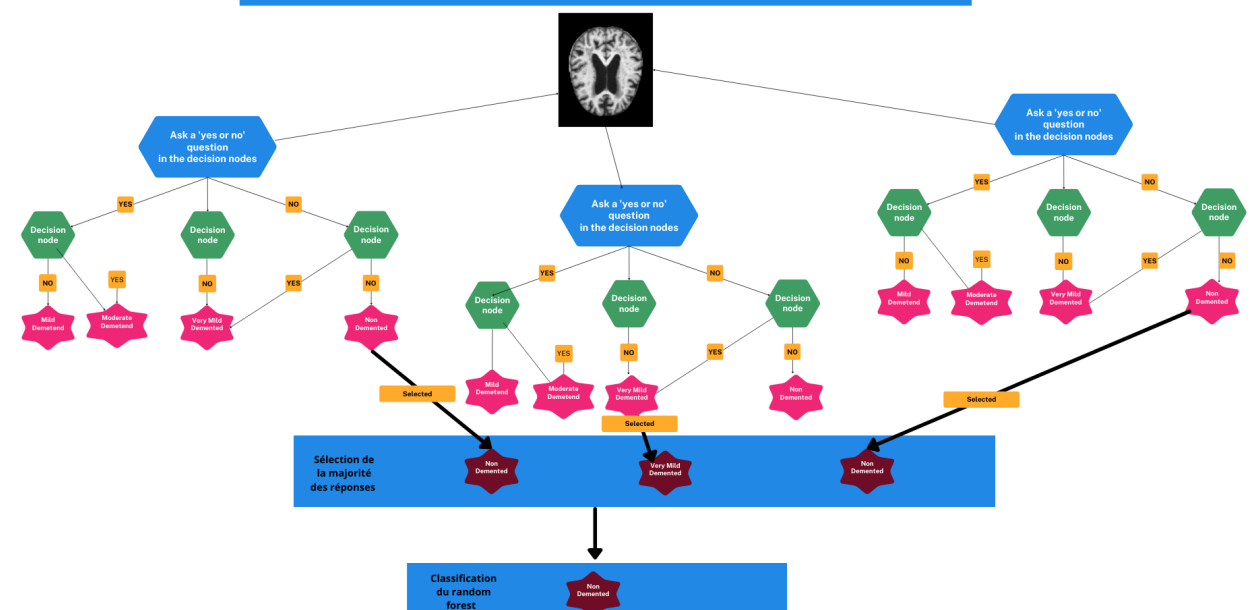
```

## Random Forest .4

Une random forest est un algorithme de classification basé sur la construction d'un grand nombre d'arbres de décision qui fonctionnent ensemble pour prendre une décision. Dans chaque arbre de décision, des tests sont effectués sur les caractéristiques décisives de l'image (par exemple, la nuance de gris des pixels) et les réponses conduisent à des branches spécifiques de l'arbre. Chaque arbre prend une décision et la forêt choisit la décision qui a été prise par la majorité des arbres.

L'utilisation d'une random forest est pertinente dans le cas de données d'IRM de cerveau classées selon 4 niveaux de démence d'Alzheimer, car les images sont hautement complexes et comportent de nombreuses caractéristiques décisives. Les arbres de décision peuvent être très efficaces pour séparer les différentes classes de démence, car ils peuvent être entraînés pour poser des questions spécifiques sur les caractéristiques de chaque image. En combinant de nombreux arbres de décision, la random forest peut capturer la complexité des images et améliorer la précision de la classification.

### Random forest



En outre, la méthode de cross-validation utilisée pour évaluer la performance des modèles est une technique fiable pour estimer l'exactitude de la prédiction sur de nouveaux ensembles de données. En utilisant 5 plis, le modèle est entraîné sur 4 plis et testé sur le cinquième pli, ce qui permet de détecter les éventuels problèmes de sur-apprentissage (overfitting) et de généraliser la prédiction à de nouveaux ensembles de données.

Enfin, l'utilisation de numpy pour transformer les images en matrices de nuance de gris permet de manipuler efficacement les données et de les intégrer dans les algorithmes de classification. En utilisant une random forest, il est possible d'extraire des caractéristiques importantes des images et de les utiliser pour prédire le niveau de démence d'Alzheimer avec une précision élevée.

Voici l'implémentation de la Random Forest :

Nous chargeons les données de Kaggle avec la même méthode que pour l'arbre de décision. Ensuite, nous utilisons nos données pour entraîner notre modèle de Random Forest :

```
92 # Créer une instance de RandomForestClassifier
93 rf = RandomForestClassifier(n_estimators=100, random_state=42)
94 rf_2 = RandomForestClassifier(n_estimators=100, random_state=42)
95 # Entraîner le modèle sur les données d'entraînement
96 rf.fit(X_train.reshape(len(X_train), -1), Y_train)
97 # Prédire les classes des données de test
98 Y_pred = rf.predict(X_test.reshape(len(X_test), -1))
99 # Effectuer la validation croisée sur les données d'entraînement
100 scores_rf = cross_val_score(rf_2, X_train.reshape(len(X_train), -1), Y_train, cv=10)
101 # Calculer l'accuracy
102 accuracy = accuracy_score(Y_test, Y_pred)
103 print("Accuracy RandomForest cross validation:" {:.2%}.format(scores_rf.mean()))
104 print(" Accuracy RandomForest {:.2%}".format(accuracy) )
105
```

Le nombre d'arbres dans une Random Forest est un hyperparamètre important qui peut affecter la précision du modèle. En général, plus le nombre d'arbres est élevé, meilleure sera la performance de la Random Forest, jusqu'à un certain point où l'ajout de plus d'arbres n'améliore plus significativement la précision du modèle.

Cela est dû au fait que chaque arbre de décision dans une Random Forest est construit à partir d'un sous-ensemble aléatoire des données d'entraînement, et en utilisant un sous-ensemble aléatoire des caractéristiques pour chaque division de l'arbre. En augmentant le nombre d'arbres dans la forêt, la variance du modèle diminue car les prédictions moyennes de chaque arbre convergent vers une prédiction plus précise et stable.

Cependant, un grand nombre d'arbres peut entraîner un surapprentissage du modèle, c'est-à-dire que le modèle peut commencer à mémoriser les données d'entraînement plutôt que de généraliser correctement les relations entre les caractéristiques et les étiquettes de classe. Dans ce cas, la précision du modèle sur les données de test peut commencer à diminuer.

En pratique, le nombre optimal d'arbres dans une Random Forest dépend des données spécifiques sur lesquelles le modèle est formé et doit être sélectionné en utilisant une

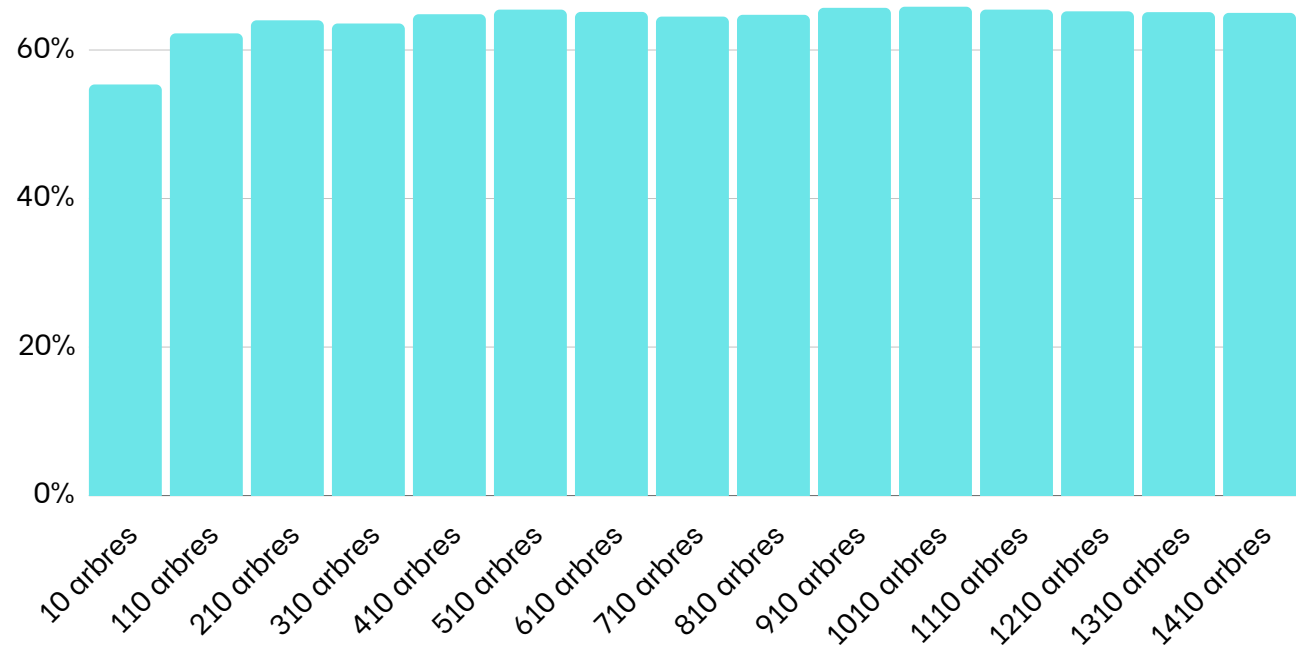
méthode d'optimisation d'hyperparamètres.

Nous avons tenté d'optimiser le nombre d'arbre de décision à générer dans notre forêt. Notre méthode d'optimisation consiste à choisir une valeur initiale (10 arbres) pour le paramètre, à évaluer la précision à cette valeur, puis nous augmentons la valeur du paramètre, de manière à se rapprocher de la valeur optimal de précision. Le processus est répété jusqu'à ce qu'une valeur suffisamment proche de la valeur maximale soit atteinte.

Voici l'implémentation du code :

```
for n_estimators in tqdm(range( 10, 2000, 100)):  
    rf = RandomForestClassifier(n_estimators=n_estimators, random_state=42)  
    # Entraîner le modèle sur les données d'entraînement  
    rf.fit(X_train.reshape(len(X_train), -1), Y_train)  
    # Prédire les classes des données de test  
    Y_pred = rf.predict(X_test.reshape(len(X_test), -1))  
    # Calculer l'accuracy  
    accuracy = accuracy_score(Y_test, Y_pred)  
    print(" Accuracy RandomForest for number forests = {}: {:.2%}".format(n_estimators,accuracy) )
```

Résultats :



Entre 10 et 400 arbres, la précision augmente énormément, puis entre 500 et 1000 la précision stagne. A plus 1000 arbres la précision commence à légèrement décroître.

## Réseau de neurones .5

Nous avons décidé de comparer les précédents principes de classification avec un nouveau modèle, le réseau de neurones. Ce modèle est l'un des plus connus lorsque que nous parlons de Machine Learning.

## Prérequis :

Pour le réseau de neurones, nous avons utilisé les bibliothèques Python TensorFlow, ainsi que Keras. Ces bibliothèques permettent de réaliser de l'apprentissage automatique sur Python. Elles ont des fonctions qui nous permettent de ne pas développer tout le schéma d'un réseau de neurones, c'est-à-dire que ces bibliothèques ont déjà implémenté les fonctions qui nous seront utiles pour la création du réseau de neurones, l'entraînement et le test de celui-ci.

```
import os
import tensorflow as tf
import matplotlib.pyplot as plt

try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
    print('Device:', tpu.master())
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
except:
    strategy = tf.distribute.get_strategy()
print('Number of replicas:', strategy.num_replicas_in_sync)

print(tf.__version__)
```

Pour faciliter la modification de “constantes”, nous initialisons une partie de code avec toutes ces informations.

```
AUTOTUNE = tf.data.experimental.AUTOTUNE
BATCH_SIZE = 16 * strategy.num_replicas_in_sync
IMAGE_SIZE = [176, 208]
EPOCHS = 100
```

## Chargement des Images :

Pour commencer, nous allons charger le dossier initial des données, en pensant à séparer les données d'entraînement du modèle et les données de validation du modèle. Nous pouvons réaliser cette opération grâce à Keras, et l'utilisation de la fonction “image\_from\_directory” qui vient chercher les données souhaitées dans notre projet. Comme nous l'avons précisé dans la présentation des données, nous devons redimensionner les images, c'est pour cela que dans la fonction qui importe les données nous avons ajouté le paramètre “image\_size”, en lui indiquant la taille souhaitée. Ici; la taille souhaitée est 176x208. Nous avons fait cela pour les deux datasets.

```

train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    "../MachineLearningProject/AlzheimersDataset/train",
    validation_split=0.2,
    subset="training",
    seed=1337,
    image_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
)

val_ds = tf.keras.preprocessing.image_dataset_from_directory(
    "../MachineLearningProject/AlzheimersDataset/train",
    validation_split=0.2,
    subset="validation",
    seed=1337,
    image_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
)

```

## Création des niveaux :

Nous avons créé les différentes classes dans lesquelles nous allons classer les données.

```

class_names = ['MildDemented', 'ModerateDemented', 'NonDemented', 'VeryMildDemented']
train_ds.class_names = class_names
val_ds.class_names = class_names

NUM_CLASSES = len(class_names)

```

Puisque nous réalisons une classification de données non continues, nous devons “prévenir” que ce ne sera pas un modèle continu. Pour cela, nous utilisons la méthode “one-hot”. De plus, nous utilisons la fonction “prefetch” pour rendre l’appel des images plus efficace.

```

def one_hot_label(image, label):
    label = tf.one_hot(label, NUM_CLASSES)
    return image, label

train_ds = train_ds.map(one_hot_label, num_parallel_calls=AUTOTUNE)
val_ds = val_ds.map(one_hot_label, num_parallel_calls=AUTOTUNE)

train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

```

## Création du modèle :

Nous allons réaliser la fonction qui va permettre de construire le modèle. Le modèle que l’on souhaite réaliser est un Réseau de Neurones Convolutif. Pour cela, nous allons décrire les différents blocs contenus dans ce modèle. Il y a un bloc convolutif et un bloc de densité. Pour



préciser ces blocs, nous utiliserons comme fonction d'activation le modèle Rectified Linear Unit (ReLU) et pour la fonction de pertes, nous utiliserons le modèle Cross Entropy.

```
def conv_block(filters):
    block = tf.keras.Sequential([
        tf.keras.layers.SeparableConv2D(filters, 3, activation='relu', padding='same'),
        tf.keras.layers.SeparableConv2D(filters, 3, activation='relu', padding='same'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.MaxPool2D()
    ])
    return block

def dense_block(units, dropout_rate):
    block = tf.keras.Sequential([
        tf.keras.layers.Dense(units, activation='relu'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dropout(dropout_rate)
    ])
    return block
```

```
def build_model():
    model = tf.keras.Sequential([
        tf.keras.Input(shape=(*IMAGE_SIZE, 3)),

        tf.keras.layers.Conv2D(16, 3, activation='relu', padding='same'),
        tf.keras.layers.Conv2D(16, 3, activation='relu', padding='same'),
        tf.keras.layers.MaxPool2D(),
        conv_block(32),
        conv_block(64),
        conv_block(128),
        tf.keras.layers.Dropout(0.2),
        conv_block(256),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Flatten(),
        dense_block(512, 0.7),
        dense_block(128, 0.5),
        dense_block(64, 0.3),
        tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
    ])
    return model
```

Grâce à ces trois fonctions, nous allons construire le modèle.

```
with strategy.scope():
    model = build_model()

    METRICS = [tf.keras.metrics.AUC(name='auc')]

    model.compile(
        optimizer='adam'
```

```
optimizer = Adam(),
loss=tf.losses.CategoricalCrossentropy(),
metrics=METRICS
)
```

## Entraînement du modèle :

Après la création du modèle, nous allons l'entraîner à l'aide de la fonction "fit". Nous entraînons le modèle sur le jeu de données d'entraînement, ici train\_ds. Puis, nous validons ce modèle à l'aide du jeu de données de validation, ici val\_ds.

```
history = model.fit(
    train_ds,
    validation_data=val_ds,
    callbacks=[checkpoint_cb, early_stopping_cb, lr_scheduler],
    epochs=EPOCHS
)
```

Nous utilisons des callbacks pour stocker les meilleurs entraînements.

Pour chaque étape de l'entraînement, nous obtenons des informations sur celui-ci.

Epoch 1/100

257/257 [=====] - 146s 553ms/step - loss: 1.0989 - auc: 0.7828 - val\_loss: 25.1692 - val\_auc: 0.6634 - lr: 0.0100

Epoch 10/100

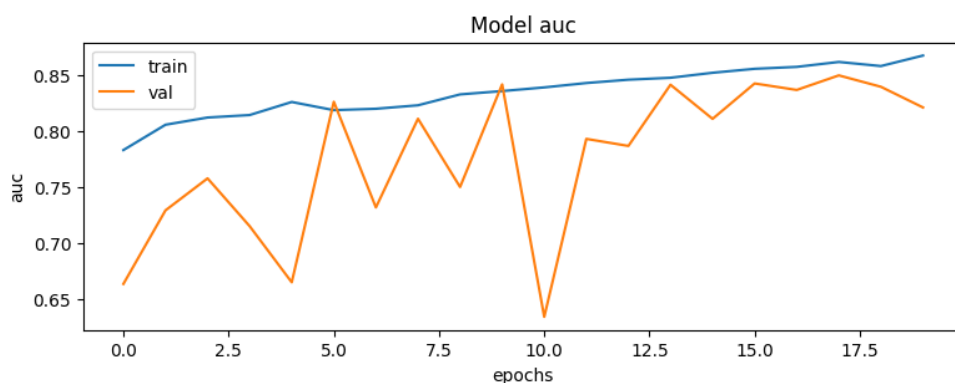
257/257 [=====] - 141s 549ms/step - loss: 0.9126 - auc: 0.8354 - val\_loss: 0.8878 - val\_auc: 0.8414 - lr: 0.0035

Epoch 20/100

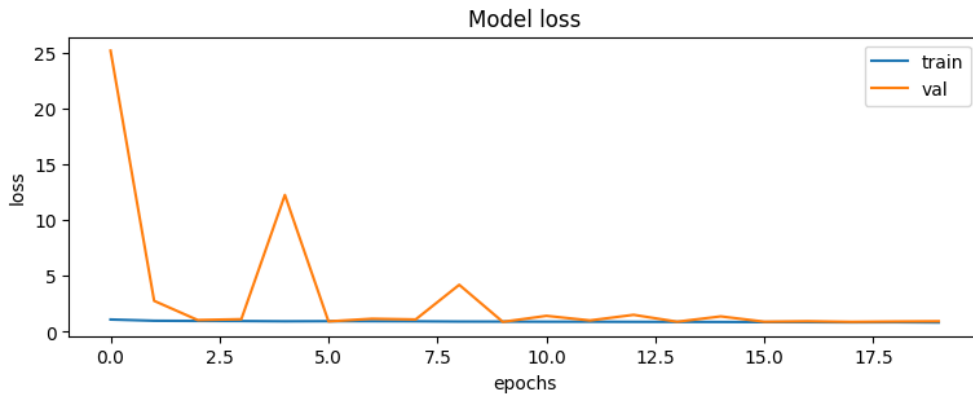
257/257 [=====] - 140s 546ms/step - loss: 0.8363 - auc: 0.8671 - val\_loss: 0.9475 - val\_auc: 0.8208 - lr: 0.0011

Nous pouvons remarquer une diminution des pertes.

Suite à l'entraînement du modèle, nous obtenons les courbes de précision et de pertes, en fonction de l'entraînement, les courbes sont les suivantes :



Courbe de précision



Courbe de pertes

## Evaluation du modèle :

Dans un premier temps, comme pour les jeux de données précédents, nous allons charger les images. Puis, appliquer les fonctions qui permettent d'identifier le "problème" comme étant non-continu.

```
test_ds = tf.keras.preprocessing.image_dataset_from_directory(
    "../MachineLearningProject/AlzheimersDataset/test",
    image_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
)

test_ds = test_ds.map(one_hot_label, num_parallel_calls=AUTOTUNE)
test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Pour finir, nous allons utiliser la fonction "evaluate" pour tester notre modèle.

```
_ = model.evaluate(test_ds)
```

Suite à cette évaluation, nous obtenons le résultat du modèle sur le jeu de test. Les résultats sont les suivants :

```
80/80 [=====] - 11s 131ms/step - loss: 0.8958 - auc: 0.8436
```

Nous obtenons au final un modèle qui a une précision de 84.36%.

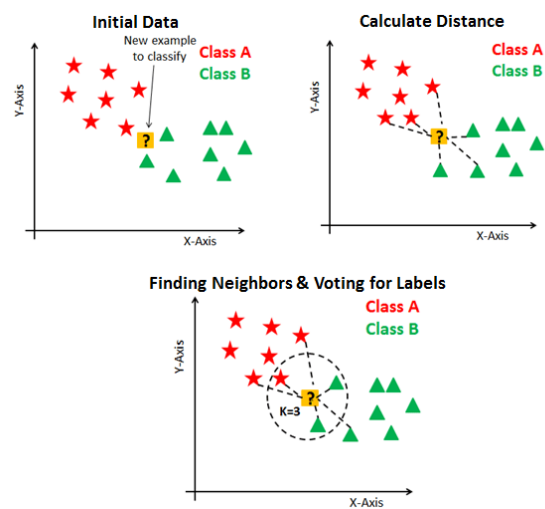
## Méthode KNN .6

---

La méthode du k-plus proche voisins (KNN) est un algorithme de classification supervisée.

Le modèle est simple, il détermine la distance entre la donnée à classer et les données d'entraînements. Ensuite, il classe la donnée en identifiant la nature de la majorité des

données les plus proches.



Voici l'implémentation du modèle :

Les données d'entrée ont été obtenues de la même manière que précédemment. Ensuite, nous utilisons la bibliothèque sklearn neighbors.

Le nombre de voisins a été défini à 5.

```
# # KNN

knn = KNeighborsClassifier(n_neighbors=5)

# # Entraîner le modèle sur les données d'entraînement
knn.fit(X_train.reshape(len(X_train), -1), Y_train)

# # Prédire les classes des données de test
Y_pred = knn.predict(X_test.reshape(len(X_test), -1))

# # Calculer la précision (accuracy) du modèle
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy KNN: {:.2%}".format(accuracy))

knn_2 = KNeighborsClassifier(n_neighbors=5)
scores_knn = cross_val_score(knn_2, X_train.reshape(len(X_train), -1), Y_train, cv=10)

print("Accuracy Knn crossvalidation: {:.2%}".format(scores_knn.mean()))
```

## Comparaison des résultats .7

Modèle	Précisions ( jeu de test )	Cross Validation

KNN	72 %	90 %
Random Forest	62 %	43 %
Decision Trees	47.9 %	36 %
Réseaux de neurones	84.4 %	82%

## Conclusion

Après avoir comparé plusieurs modèles de classification d’images utilisés dans le cadre de la classification de la démence de la pathologie d’Alzheimer, les résultats obtenus sur ces données ont montré que le modèle le plus performant pour cette tâche de classification est le modèle de réseaux de neurones, avec une précision de 84.4% et une Cross Validation de 82%. Ce modèle est reconnu de part son efficacité à extraire les caractéristiques des images et à prédire les classes cibles.

Pour conclure, les résultats de cette étude montrent que les modèles de réseaux de neurones sont capables de prédire avec précision la classe de démence d’Alzheimer à partir d’images IRM de cerveaux et pourraient aider les professionnels de santé à diagnostiquer la maladie plus rapidement.

