Department of Computer Science
Technical University of Cluj-Napoca

Sisteme Inteligente

Proiect laborator 2019-2020

Adrian Groza, Anca Marginean and Radu Razvan Slavescu
Tool: Colaboraty, PyTorch, Pandas, Scikit-Learn

Name: Roxana Ciorea - Cristescu
Group: 30235
Email: roxana.ciorea@gmail.com

Assoc. Prof. dr. eng. Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Chapter 1

Prezentare cerință

1.1 Descriere narativă

Implementați următoarele rețele neuronale de la laborator și realizați modificări.

1. Perceptron - exemplu diferit
2. Multi Layer Perceptron - dataset nou
   (a) Preprocesare date
   (b) Împarțire training set și validation set
   (c) Vizualizare rezultate
   (d) Implementare mini-batch
3. Convolutional Neural Network - dataset nou + modificare rețea
4. Recurrent Neural Network - dataset nou + modificare rețea
5. Decision Tree - dataset nou + modificare parametri
6. K-Means Clustering - modificare parametri
7. K-Nearest Neighbors - dataset nou + modificare parametri
8. Support Vector Machine - dataset nou + modificare parametri

Deoarece nota pentru realizarea tuturor algoritmilor este 12 am decis să nu implementez punctele 3. (CNN) și 4. (RNN).

1.2 Factori

Pentru a realiza algoritmi am ales următoarele detaset-uri:

1. winequality-red - are 9 coloane de date și un rezultat care ne spune dacă avem o calitate bună sau nu
2. winequality-white - este identic cu winequality-red
3. winequality-red-6cls - este winequality-red original, având note de la 3 la 8 pentru calitatea vinului

1.3 Specificați

Pentru realizarea acestui proiect voi folosi mediile de proiectare Colaboratory și Visual Studio, iar codul va fi realizat în Python, folosind bibliotecile:

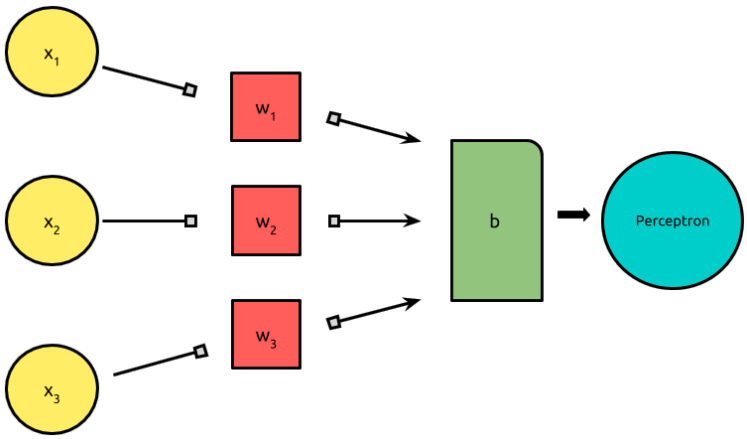
1. numpy - array-uri multi-dimensionale și funți matematice
2. torch - lucru cu seturi de date și algorithmi pt deap learning
3. pandas - manipulare și analiza de date și fișiere
4. sklearn - conține algoritmi de clasificare, regresie și clasterizare

Chapter 2

Detali de implementare

2.1 Perceptron

Perceptronul este un tip de neuron care lucrează strit cu input-uri și output-uri binare. Un perceptron poate primi orice număr de input-uri (eu am 2), pentru a obtine un singur output fiecare input are un weight care determina relevanța lui. Suma input-urilor înmulțite cu weight-urile este apoi comparata cu un threshold/bias, dacă suma este mai mare output-ul este 1, dacă nu 0.



Deoarece bias-ul poate fi definit ca și weight-ul unul output mereu activ putem trece bias-ul in lista de weight-uri pentru a fi mai ușoară modificare la antrenare astfel devenind parte din sumă. Formula pentru activarea devine:

```
f(x) = 1 if w · x > 0
       0 otherwise
w · x = (w1 * x1) + (w2 * x2) + ... + (wn * xn) + (b * 1)
```

Eu am ales să folosesc un perceptron pentru a deduce dacă merg sau nu la curs. Ca intrări am pus cele mai relevante aspecte pentru mine, acestea fiind:

1. A - Dacă se face prezența la curs
2. B - Dacă cursul este de la 8:00 dimineața
3. C - Dacă Mariel (colegul cu care stau mereu) vine la curs

2.1.1 Cod relevant

Codul are 2 parți, definirea clasei perceptron (despre care voi vorbi în continuare) și utilizarea acesteia (despre care voi vorbi la secțiunea Demo Proiect)

Primul pas constă în inițializarea parametrilor: no_of_inputs (se înțelege de la sine), threshold (numarul de epoci, treceri print toate datele), learning_rate (cât de mult se învață per iterație)

```
def __init__(self, no_of_inputs, threshold=100, learning_rate=0.01):
    self.threshold = threshold
    self.learning_rate = learning_rate
    self.weights = np.zeros(no_of_inputs + 1)
```

Mai apoi se realizează formula menționată anterior, adică se face înmulțirea dot între vectorul de input-uri și weight-uri. Dacă suma este mai mare decât 0 atunci activation adică output-ul este 1, altfel 0.

```
def predict(self, inputs):
    summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
    if summation > 0:
      activation = 1
    else:
      activation = 0
    return activation
```

Nu în ultimul rând, trebuie să antrenăm perceptronul, adică să modificăm weight-urile și bias-ul în funcție de diferența dintre răspunsul așteptat și cel prezis. Asta se face de threshold ori (care este numărul de epoci ales de noi), pentru fiecare epoca se vor lua pe rând fiecare input, se va face o predicție, și apoi se vor realiza modificările.

```
def train(self, training_inputs, labels):
    for _ in range(self.threshold):
        for inputs, label in zip(training_inputs, labels):
            prediction = self.predict(inputs)
            self.weights[1:] += self.learning_rate * (label - prediction) * inputs
            self.weights[0] += self.learning_rate * (label - prediction)
```

2.2 Prelucrarea datelor

Un pas foarte importan înainte de a implementa oricare din algorithmi care vor fi menționați mai jos este introducerea și prelucrarea datelor.

2.2.1 Cod relevant

Primul lucru pe care trebuie să îl facem este să deschidem dataset-ul, asta se poate face atât prin uploadarea acestuia și deschiderea sa din contents, sau direct prin URL

```
#Load dataset
wine = pd.read_csv("/content/winequality-red.csv")
wine.head()
```

```
# Load the train and test datasets to create two DataFrames
train_url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/train.csv"
train = pd.read_csv(train_url)
test_url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/test.csv"
test = pd.read_csv(test_url)
```

Mai departe trebuie să separăm featere-urile (datele de intrare) de target (output), acestea se lavează în 2 variabile dupa cum urmeaza

```
#Selectăm datele de intrare in retea eliminand ultima coloană din csv
X = df.drop("quality", axis=1)
#obținem etichetele pentru date salvand ultima coloana
y = df['quality']
```

Acum trebuie să împărțim datele într-un subset pentru training și unul pentru test/validare

```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=True)
```

În cazul în care avem feature-uri care nu sunt numerice putem să le codificăm astfel:

```
labelEncoder = LabelEncoder()
labelEncoder.fit(train['Sex'])
labelEncoder.fit(test['Sex'])
train['Sex'] = labelEncoder.transform(train['Sex'])
test['Sex'] = labelEncoder.transform(test['Sex'])
```

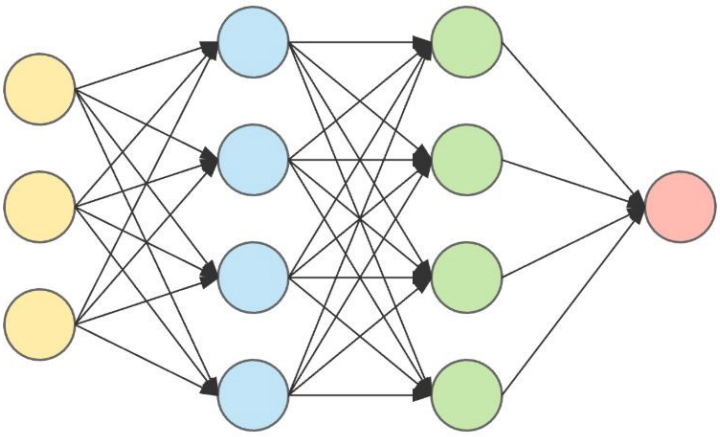
Pentru o mai bună funcționare a algoritmilor putem și să normalizăm datele

```
# X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
# X_scaled = X_std * (max - min) + min

sc = MinMaxScaler((-1, 1))
X_train_sc = sc.fit_transform(X_train)
X_test_sc = sc.transform(X_test)
```

2.3 Multi Layer Perceptron

Puterea rețelelor neuronale provine din capacitatea lor de a învăța reprezentarea datele și cum să le relaționeze cel mai bine cu output-ul pe care dorim să îl preziciem, în acest sens, rețelele neuronale învață o mapare. Din punct de vedere matematic, sunt capabili să învețe orice funcție de mapare și s-a dovedit că sunt un algoritm universal de aproximare.

Capacitatea de predicție provine din structura ierarhică/multi-layer a rețelei. La bază stau neuroni, structuri simple definite în secțiunea anterioară. Aceștia sunt aranjați în layer-e și fiecare învață câte ceva despre datele de input.



O rețea are mai multe tipuri de layer-e:

1. Layer de input/vizibil - primul layer, care primește datele
2. Layer-e ascunse - toate layer-ele dintre cel de input și cel de output, acunse deoarece nu sunt expuse la factori din lumea exterioară
3. Layer de output - care ne va oferi predicția

IMPORTANT: Înainte de a începe antrenarea rețelei dataset-ul trebuie pregătit, asta însemnă codarea oricăror feature-uri care nu sunt în format numeric și normalizarea tuturor datelor.

2.3.1 Cod relevant

După prelucrarea datelor primul lucru pe care îl facem este definirea unei clase care va încapsula dataset-ul nostru:

1. torch.tensor este folosit pentru a crea o matrice multidimensională având toate elementele de același tip

```
#Dataset - o clasă din PyTorch foarte utilă gestionării seturilor de date
class Dataset(Dataset):
    """ Wine dataset."""
    # Initialize your data, download, etc.
    def __init__(self, x, y):
        #Citim setul de date
        self.len = len(x)

        self.x=torch.tensor(x).float()
        self.y=torch.tensor(y.values).long()

    def __getitem__(self, index):
        return self.x[index], self.y[index]

    def __len__(self):
        return self.len
```

Mai apoi trebuie să ne creem aceste obiecte Dataset:

1. DataLader este o funcție care iterează peste un Dataset și suportă despărțire automată în batch-uri

```
trainDataset=Dataset(X_train, y_train)
trainLoader=DataLoader(dataset=trainDataset,
                       batch_size=16,
                       shuffle=True,
                       num_workers=1)

validationDataset=Dataset(X_test, y_test)
validationLoader=DataLoader(dataset=validationDataset,
                       batch_size=16,
                       shuffle=True,
                       num_workers=1)
```

Următorul pas este crearea rețelei noastre neuronale, în cazul meu aceasta are 5 layer-e și folosește funcția ReLu:

```
class WineNN(nn.Module):
    def __init__(self):
        super(WineNN, self).__init__()

        #Sequential oferă o alternativă mai estetică a codului
        #Rețeaua noastră are 2 neuroni pentru output.
        #Unul va prezice probabilitatea pentru cazul afirmativ
        self.sequential= nn.Sequential(
            nn.Linear(11,50),
            nn.ReLU(),
            nn.Linear(50, 100),
            nn.ReLU(),
            nn.Linear(100, 30),
            nn.ReLU(),
            nn.Linear(30, 60),
            nn.ReLU(),
            nn.Linear(60, 2)
        )

    def forward(self, x):
        return self.sequential(x)
```

În continuare definim funcția de training, aceasta va fi apelată în fiecare epocă și va trece prin toate datele, luând fiecare batch pe rând:

1. Datele din batch sunt separate in input-uri și labels (output-uri dorite)
2. Aflăm predicția rețelei
3. Loss-ul este calculat folosind nn.CrossEntropyLoss() de predinți si labels
4. Folosind loss-ul si learning rate-ul din optimizer se face un backward pass și se updatează weight-urile

```
# Training loop
def train(epoch):
    # Setează câteva flaguri în rețeaua neurală. Specific activează Dropout-ul și BatchNormalization
    # În exemplul nostru are un rol pur demonstrativ, nefiind necesar.
    net.train()
    losses=[]
    for batch_idx, data in enumerate(trainLoader, 0):
      inputs, labels =data
      #Obținem predictii
      outputs = net(inputs)
      # Compute and print loss
      loss = criterion(outputs, labels)

      losses.append(loss.item())
      # Zero gradients, perform a backward pass, and update the weights.
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
      print(f"[Train Epoch: {epoch}, Batch: {batch_idx+1}, Loss: {loss.item()}")
    mean_loss=sum(losses)/len(losses)
    scheduler.step(mean_loss)
    train_losses.append(mean_loss)
    print(f"[TRAIN] Epoch: {epoch} Loss:{mean_loss}")
```

Ultimul pas este definirea funcției de validare:

1. Oprim flag-ul de învățare (weight-urile nu se vor mai modifica)
2. Datele din batch sunt separate in input-uri și labels (output-uri dorite) și aflăm predicția rețelei
3. Loss-ul este calculat folosind nn.CrossEntropyLoss() de predinți si labels
4. Calculăm acurateția în funcție de câte predicți corecte a făcut rețeaua

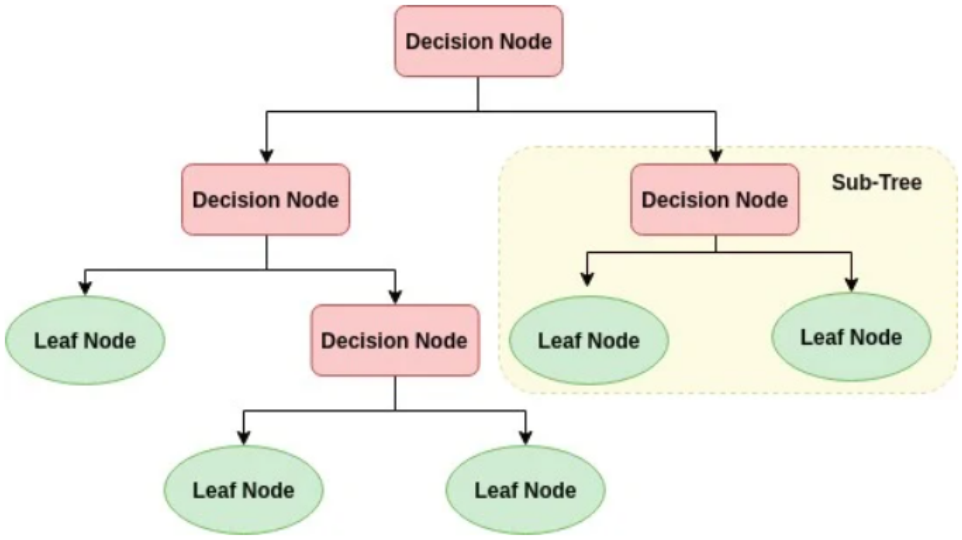
```
def validation():
    #Pune pe off flagurile setate in model.train()
    #Din nou, în exemplul nostru e pur demonstrativ.
    net.eval()

    test_loss=[]
    correct = 0

    with torch.no_grad():
        for batch_idx, data in enumerate(validationLoader, 0):
          inputs, labels = data

          output=net(inputs)

          loss= criterion(output, labels)
          test_loss.append(loss.item())

          #Obținem predictiile pentru fiecare linie din setul de validare.
          #Practic ne returnează rezultatul cu cea mai mare probabilitate pentru fiecare intrare din setul de validare
          pred = output.data.max(1, keepdim=True)[1]

          #Verificăm câte predicții sunt corecte și le însumăm numărul pentru a afla totalul de predicții corecte
          correct += pred.eq(labels.data.view_as(pred)).sum()
          current_correct=pred.eq(labels.data.view_as(pred)).sum()
          print("============")
          print(f"[Validation set] Batch index: {batch_idx+1} Batch loss: {loss.item()}, Accuracy: {100. * current_correct/len(inputs)}%")
          print("============")
        mean_loss=sum(test_loss)/len(test_loss)
        test_losses.append(mean_loss)
        accuracy = 100. * correct/len(validationLoader.dataset)
        print(f"[Validation set] Loss: {mean_loss}, Accuracy: {accuracy}%")

        accuracies.append(accuracy)
```

Tot ce mai rămâne acum de făcut este să apelăm funcțiile de un număr ales de noi de epoci:

```
#după fiecare epocă de train() verificăm rezultatele pe setul de validare
for epoch in range(100):
  train(epoch)
  validation()
```

2.4 Decision Tree

Un arbore de decizi este o structură tip flux de date în care un nod intern reprezintă un feature (sau un atribut), ramură reprezintă o regulă de decizie și fiecare nod frunză reprezintă rezultatul aplicări reguli. Învață să partiționeze pe baza valorii atributelor și face acest lucru recursiv. Arborele de decizii este un algoritm ML de tip cutie albă, acest lucru însemnând ca programatorul poate vedea logica internă.



Un arbore de decizi fucționeaza conform următorilor pasi:

1. Selectează cel mai bun atribut folosind ASM pentru a împărți înregistrările.
2. Face din acel atribut un nod de decizie și sparge setul de date în subseturi mai mici.
3. Începe construirea arborilor prin repetarea acestui proces recursiv pentru fiecare copil până când:
   (a) Toate tuplurile aparțin aceleiași valori de atribut.
   (b) Nu mai rămân atribute.
   (c) Nu mai există cazuri.

ASM sau Masutătorile de selecție a atributelor sunt:

1. Information Gain sau Entropy - măsoară impuritatea setului de intrare
2. Gain Ratio - o îmbunătățire a Entropiei prin normalizarea câștigului de informații folosind Split Info
3. Gini index - consideră o împărțire binară pentru fiecare atribut

2.4.1 Cod relevant

Deoarece algoritmul Decision Tree există deja implementat în scikit-learn codul relevant pentru această secțiune constă în 3 lini de cod pentru implementare.

Crearea clasificaorului arborelui de decizii, aici putem să modificăm atributele pentru a încerca să ajungem la un rezultat optim.

```
# Create Decision Tree classifer object
clf = DecisionTreeClassifier(criterion="entropy",splitter="random", max_depth=3)
```

Câțiva dintre parametri acestei funcți sunt:

1. criterion - ASM-ul folosit (”entropy” sau ”gini”)
2. splittet - alege cum se face împărțirea (”best” sau ”random”)
3. max_depth - adâncimea maximă a arborerul (defaul = ”none”, putem pune int-uri)
4. min_sample_split - numărum minim de date pt un split (defaul = 2)

Echivalentul funției de training de până acum este funcția .fit la care se trimite dataset-ul de training și rezultatele acestuia.

```
# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)
```

Ultimul pas este realizarea predicțiilor pentru un alt dataset de test, se poate observa și calcularea acurateți rezultatelor prezise.

```
#Predict the response for test dataset
y_pred = clf.predict(X_test)

# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

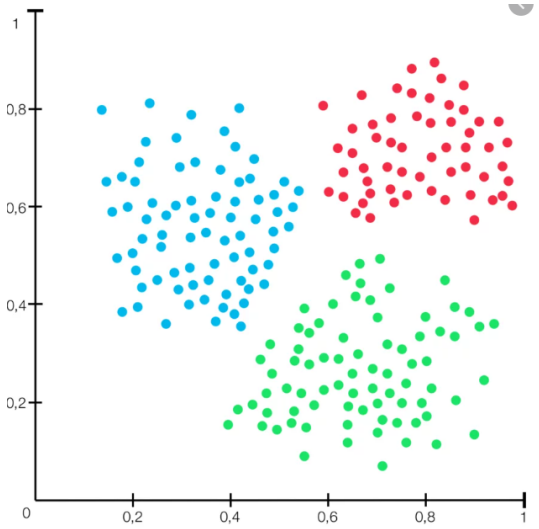
Flosind bibliotecile graphviz și pydotplus avem posibilitatea de a crea o imagine conținând aroborele creat.

```
dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True,feature_names = feature_cols,class_names=['0','1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('WhiteWine.png')
Image(graph.create_png())
```

2.5 K-Means Clustering

Clustering este acțiunea de a grupa un set de obiecte astfel încât obiectele din același cluster să fie mai similare între ele decât cu obiecte din alte clustere. Similaritatea este o metrică care reflectă puterea relației dintre două obiecte.Clustering face parte din ramura de Unsupervized learning, cea ce înseamă că dataset-ul nu are inclus și output-ul deci acesta nu poate fi folosit la antrenare.

K-Means se încadrează în categoria grupărilor bazate pe centroid. Un centroid este un punct de date care se găsește în centrul unui cluster, acest nu trebune neapărat să aparțină dataset-ului.



Conceptul de bază al algoritm-ului este:

1. Luăm oricare K centroizi sau puncte de date
2. După alegerea centroizi toate datele sunt alocate unui cluster în funcție de distanța până la cetroid
3. Algoritmul va continua actualizarea centroizilor cu centrul cluster-ului până când actualizarea nu modifică conținutul cluster-ul

2.5.1 Cod relevant

K-Means este un alt algoritm care se găsește în biblioteca scikit-learn, de aceea codul constă doar din funcțiile apelate, fară implementarea acestora

Primul pas este crearea obiectului K-means, pentru asta se folosește următoarea funcție:

```
# You want cluster the passenger records into 2: Survived or Not survived
kmeans = KMeans(n_clusters=2, max_iter=600, algorithm = 'auto')
```

Pentru a obține rezultate mai bune parametri se pot schimba, câțiva dintre acești parametri sunt:

1. n_clusters - numărul de clustere (int)
2. init - metoda de inițializare (”k-means++”, ”random” sau putem introduce noi un array cu centroizi)
3. n_init - de câte ori va fi rulat algoritmul cu centroizi diferiți (int)
4. max_iter - număr maxim de iterați pt o rulare (int)

Echivalentul funcției de training, de data asta având ca și parametru doar dataset-ul cu input-uri:

```
kmeans.fit(X)
```

Pentru a putea vedea acurateția algoritmului luăm fiecare obiect din input, îl convertim în float și facem o predicție, apoi acea predicție este comparată cu rezultatul pe care noi trebuia să îl primim:

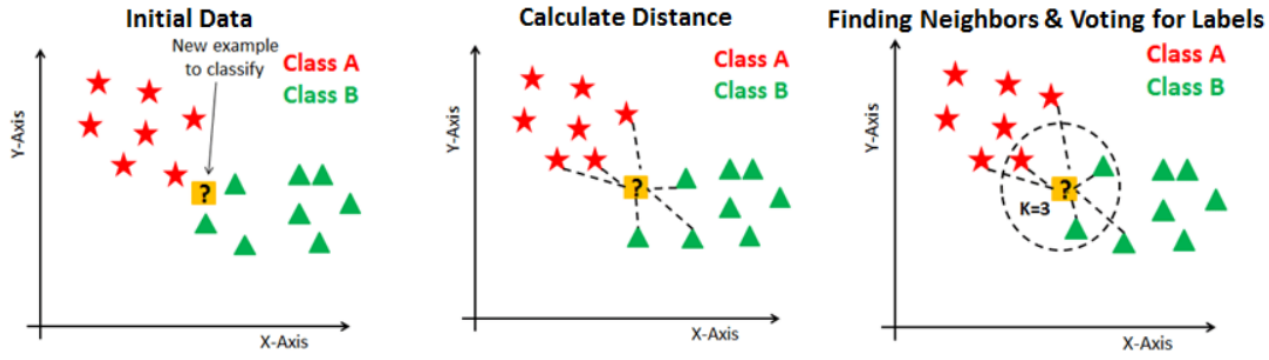
```
correct = 0
for i in range(len(X)):
    predict_me = np.array(X[i].astype(float))
    predict_me = predict_me.reshape(-1, len(predict_me))
    prediction = kmeans.predict(predict_me)
    if prediction[0] == y[i]:
        correct += 1

print(correct/len(X))
```

2.6 K-Nearest Neighbors

KNN este un algoritm non-parametric și lazy de învățare. Non-parametric înseamnă că nu există nicio presupunere pentru distribuirea datelor, cu alte cuvinte, structura modelului este determinată din setul de date. Acest lucru este foarte util în practică, deoarece majoritatea dataset-urilor din lumea reală nu respectă ipotezele teoretice matematice. Algoritm Lazy înseamnă că nu are nevoie de puncte de antrenare pentru generarea modelului. Toate datele sunt utilizate în faza de testare. Acest lucru face învățarea mai rapidă și testarea mai lentă și mai costisitoare.

În KNN, K este numărul vecinilor care este factorul decisiv principal. K este în general un număr impar dacă numărul de clase este 2. Cum funcționează K-nearest neighbor?

1. Calculăm dinstanța dintre punctele din model și punctul pentru care facem predicția
2. Găsim cei mai apropiați k vecini de punctul nostru
3. Fiecare vecin votează pentru clasa lui.
4. Clasa cu cele mai multe voturi este luată ca predicție



2.6.1 Cod relevant

Precum cele două algoritm-uri precedente, KNN deasemenea se poate găsi în biblioteca scikit-learn de aceea codul va consta doar din 3 lini de cod (4 cu calcularea acurateți) Crearea retelei (putem să modifiăm parametri pentru a încerca să ajungem la un rezultat optim)

```
#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=3)
```

Exemple de parametri ai funcției:

1. n_neighbors - numărul de vecini luați în considerare (int)
2. weights - importanța vecinilor (”unifor” sau ”distance” vecini mai apropiați au importanță mai mare)
3. algorithm - algoritmul folosit pentru găsirea vecinilor mai apropiați (”auto”, ”ball_tree”, ”kd_tree” sau ”brute”)

Funcția de training, care primește dataset-ul de training și rezultatele corecte ale acestuia:

```
#Train the model using the training sets
knn.fit(X_train, y_train)
```

Realizarea predicților pentru un dataset de test:

```
#Predict the response for test dataset
y_pred = knn.predict(X_test)
```

Și nu în ultimul rând calculare acurateți:

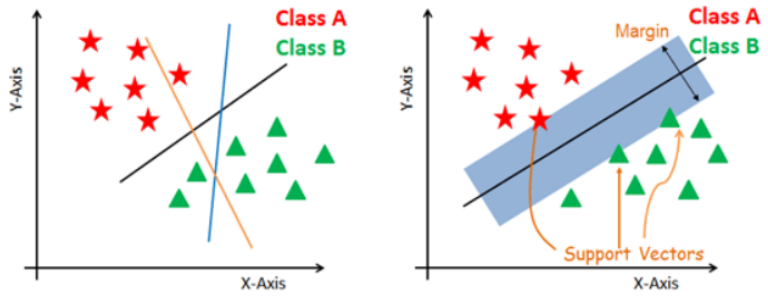
```
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

2.7 Support Vector Machine

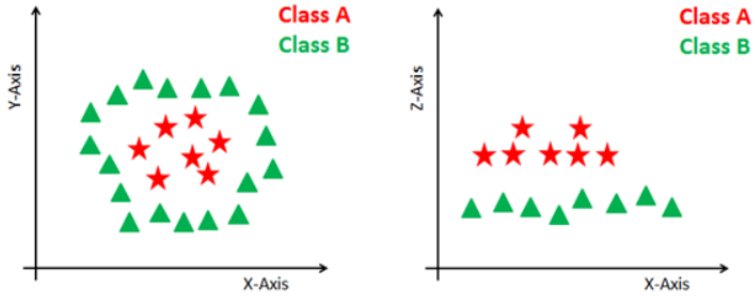
Clasificatorul SVM separă punctele dintr-un spațiu multi-dimensional folosind un hiperplan cu cea mai mare marjă. De aceea, un clasificator SVM este cunoscut și ca clasificator discriminator. SVM găsește un hiperplan optim într-un mod iterativ, care este utilizat pentru a minimiza eroarea și ajută la clasificarea datelor noi.

Algoritmul funcționează pe următorul principiu:

1. Generează hiperplane care segregă clasele.
2. Selectează hiperplanul cu magine maximă dintre punctele cele mai apropiate de clase diferite



În cazul în care clasele sunt non-lineare și inseparabile SVM folosește cu kernel pentru a transforma spațiul datelor într-o nouă dimensiune:



Există mai multe tipuri de kernel-uri

1. Linear - K(x, xi) = sum(x * xi)
2. Polynomial - K(x,xi) = 1 + sum(x * xi)$\hat{}$d
3. Radial Basis Function - K(x,xi) = exp(-gamma * sum((x – xi$\hat{}$2))

2.7.1 Cod relevant

La fel ca și algoritmi anteriori, SVM este definit în biblioteca scikit-learn așadar crearea, antrenarea și testarea algoritmului va consta din 3 lini de cod:

```
#Create a svm Classifier
clf = svm.SVC(kernel='sigmoid', gamma='auto')
```

Desigur parametri acestei funcți pot fi modificați pentru a ajunge la rezultate mai bune, auci se pot găsi explicați câțiva din acești parametri

1. kernel - specifică tipul de kernel care va fi folosit (”linear”, ”poly”, ”rbf”, ”sigmoid”, ”precomputed”)
2. degree - gradul kernel-ului polynomial
3. gamma - coeficient pentru kernel poly, rbf sau sigmoid
4. shrinking - dacă se folosește euristica de micșorare
5. probability - dacă se folosesc estimări ale probabilități (utilizează 5-fold cross-validatio)

Mai jos sunt funcțiile pentru antrenare și testare

```
#Train the model using the training sets
clf.fit(X_train_sc, y_train)
```

```
#Predict the response for test dataset
y_pred = clf.predict(X_test_sc)
```

Ultimul pas este afișarea rexultatelor, în acest caz 3:

```
# Model Accuracy: how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

# Model Precision: what percentage of positive tuples are labeled as such?
print("Precision:",metrics.precision_score(y_test, y_pred))

# Model Recall: what percentage of positive tuples are labelled as such?
print("Recall:",metrics.recall_score(y_test, y_pred))
```

Chapter 3

Aspecte ale proiectului

3.1 Perceptron

3.1.1 Avantaje și limitari ale soluției

Perceptronul, fiind echivalentul unui singur element dintr-o rețea neurală este ușor de folosit și de înțeles, însă cea mai mare limitare este faptul că nu poate învața foarte mult, este util pentru probleme decizionale dar doar până la o oarecare dificultate

3.1.2 Posibilități de extindere

Un perceptron este o structură bine definită, astfel el propriuzis nu poate fi înbunătățit, extinderea apare odată cu algoriti care folosesc mai mulți perceptroni, fiecare folosit pentru a învața o parte din informația furnizată.

3.2 Multi Layer Perceptron

3.2.1 Avantaje și limitari ale soluției

Rețelele neuronale sunt avantajoase deoarece sunt flexibile și pot fi utilizate atât pentru probleme de regresie, cât și pentru clasificare, orice date care pot fi făcute numeric pot fi utilizate în model, sunt bune pentru modelarea pe date non-liniare, cu un număr mare de intrări. Odată antrenate, previziunile sunt destul de rapide și pot fi antrenate cu orice număr de input-uri și layer-e.

Există totuși dezavantaje, cel mai notabil fiind faptul că o rețea neuronală este o cutie neagră, ceea ce înseamnă că nu putem ști cât de mult sunt influențate output-urile de fiecare input. Deasemenea antrenarea unui MLP este foarte costisitoare din punct de vedere al timpului. Și nu în ultimul rând rețeaua depinde foarte mult de datele de intrare iar dacă nu suntem atenți putem ajunge la over-fitting sau under-fitting.

3.2.2 Posibilități de extindere

O posibilitate o reprezintă schimbarea parametrilor (learning-rate, număr de epoci, batch-size) și a modelului (mai multe layer-e, mai mulți neuroni per layer).

3.3 Decision Tree

3.3.1 Avantaje și limitari ale soluției

Un mare avantajele al arborilor de decizie este faptul că este ușor de inerpretat și vizualizat. Deasemenea acesta poate să se ocupe cu ușurință de paramentri care nu sunt liniari și are nevoie de mai puțină preprocesare a datelor din partea user-ului, și nu în ultimul rând poate predice valor lipsă.

Desigur există și limitări pentru acest algorit printre care se numără sensibilitatea la zgomote și la mici modificări de date (rezultând un arbore complet nou) și faptul că este influențabil când are date dezechilibrate.

3.3.2 Posibilități de extindere

Cel mai simplu mod de îmbunătățire a acestui algoritm este prin modificarea parameptrilor funcției DecisionTreeClassifier(). Petru extindere o bună opțiune este crearea unei Păduri Random (Random Decision Forest) care, precum o rețea neurală conține mai mulți neuroni, conține mai mulți arbori de decizii.

3.4 K-Means Clustering

3.4.1 Avantaje și limitari ale soluției

Avantajele la K-Means sunt faptul ca este simplu, extrem de flexibil și eficient. Simplitatea face ușor de explicat rezultatele în contrast cu rețelele neuronale artificiale iar flexibilitatea permite ajustarea ușoară în cazul în care există probleme.

Un dezavantaj al algoritmului este faptul că nu permite dezvoltarea celui mai optim număr de clustere (acesta fiind decis de user). Deasemena k-means selectează aleatoriu mai multi centroizicest lucru poate fi bun sau rău, în funcție de locul în care algoritmul alege să înceapă.

3.4.2 Posibilități de extindere

Cel mai simplu mod de îmbunătățire a acestui algoritm este prin modificarea parameptrilor funcției Kmeans(). Deoarece K-Means funcționează doar pe date numerice, acesta nu poate fi folosit pe date din lumea reală, 2 algoritmi care aduc posibilitatea extinderi tipurilor de date sunt algoritmi K-Modes și K-Krototypes.

3.5 K-Nearest Neighbors

3.5.1 Avantaje și limitari ale soluției

Faza de antrenare a clasificării KNN este mult mai rapidă comparativ cu alți algoritmi de clasificare. Nu este nevoie să instruiți un model pentru generalizare, de aceea KNN este cunoscut ca un algoritm de învățare simplu și bazat pe instanțe. Deasemenea KNN poate fi util în cazul datelor neliniare.

Însă există și dezavantaje, faza de testare a clasificării folosind acest algoritm este mai lentă și mai costisitoare din punct de vedere al timpului și al memoriei. Deasemenea este necesară scalarea datelor, deoarece KNN folosește distanța euclidiană între două puncte iar aceasta este sensibilă la magnitudini.

3.5.2 Posibilități de extindere

Pe lângă modificarea parametrilor pentru a obține o acuratețe mai bună există clasificări care folosesc KNN, printre acestea se numără Density Based kNN Classifier (care ia în considerare densitatea de puncte în jurul unui vecin), Variable kNN Classifier (care testează pt fiecare set mai multe valori pentru k), Weighted kNN Classifier (fiecare feature primește un weight în funcție de cât de relevant este).

3.6 Support Vector Machine

3.6.1 Avantaje și limitari ale soluției

Marile avantaje ale acestui algoritm sunt: faptul că este foarte eficient în spații cu număr foarte mare de dimensiuni, faptul că este bun atunci când numărul de feature-uri este mai mare decât numărul de puncte/obiecte, și este cel mai bun algoritm atunci când clasele sunt separate.

Printre limitări avem faptul că are nevoie de mult timp de procesare pentru dataset-uri mare și nu are o bună performanță pe clase care se suprapun.

3.6.2 Posibilități de extindere

Câteva extensi posibile pentru acest algoritm se refera la clasificarea mai multor clase (cea mai simpla metoda fiind unul împotriva celorlalți), estimarea robabilităților (output-ul de la SVM nu poate fi interpretat direct ca și o probabilitate), SVM regression (pentru care exista mai multe extensi).

Chapter 4

Demo proiect

4.1 Perceptron

După cum am menționat în capitolul Detali de implementare, datele mele sunt construite pentru a deduce dacă voi merge sau nu la curs în funcție de câțiva parametri. Perceptronul făcând parte din ramura de supervized learning din ML am avut nevoie să construiesc niște output-uri corecte.

```
# A: Face prezenta
# B: E de la 8:00
# C: Vine Mariel
# Out: Merg la curs
# A|B|C|Out
# 0|0|0| 0
# 0|0|1| 1
# 0|1|0| 0
# 0|1|1| 0
# 1|0|0| 1
# 1|0|1| 1
# 1|1|0| 0
# 1|1|1| 1
```

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

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

În continuare am creat un obiect de tip perceptron cu 3 input-uri și am apelat funcția de învățare

```
perceptron = Perceptron(3)
perceptron.train(training_inputs, labels)
```

Iar ca pas final am facut câteva predicți, inclusiv una care nu se afla in setul de training

```
inputs = np.array([1, 0, 1])
print("Nu face prezenta, nu e de la 8:00 si vine Mariel => %d" % perceptron.predict(inputs))
#=> Merg
inputs = np.array([0, 1, 1])
print("Nu face prezenta, e de la 8:00 si vine Mariel => %d" % perceptron.predict(inputs))
#=> Nu merg
inputs = np.array([1, 0, 0])
print("Face prezenta, nu e de la 8:00 si nu vine Mariel => %d" % perceptron.predict(inputs))
#(Ex nou) => Merg
```

4.2 Multi Layer Perceptron

Pentru a realiza un demo la această rețea neurală am folosit dataset-ul winequality-red, pe care l-am modificat astfel încât output-ul să aibă o valoare binară (înainte output-ul era o notă care reprezenta calitatea vinului, acum output-ul este dacă vinul este unul de calitate (¿ nota 5) sau nu (¡= nota 5)

După un shuffle așa arată datele:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
776	6.9	0.765	0.18	2.4	0.243	5.5	48.0	0.99612	3.40	0.60	10.3
1102	6.1	0.480	0.09	1.7	0.078	18.0	30.0	0.99402	3.45	0.54	11.2
755	7.8	0.910	0.07	1.9	0.058	22.0	47.0	0.99525	3.51	0.43	10.7
781	6.5	0.460	0.14	2.4	0.114	9.0	37.0	0.99732	3.66	0.65	9.8
1502	7.3	0.585	0.18	2.4	0.078	15.0	60.0	0.99638	3.31	0.54	9.8
...	...	...	...	...	...	...	...	...	...	...	...
698	9.4	0.615	0.28	3.2	0.087	18.0	72.0	1.00010	3.31	0.53	9.7
1011	8.9	0.320	0.31	2.0	0.088	12.0	19.0	0.99570	3.17	0.55	10.4
1133	7.2	0.480	0.07	5.5	0.089	10.0	18.0	0.99684	3.37	0.68	11.2
722	7.6	0.420	0.08	2.7	0.084	15.0	48.0	0.99680	3.21	0.59	10.0
173	7.4	0.620	0.05	1.9	0.068	24.0	42.0	0.99610	3.42	0.57	11.5

1279 rows × 11 columns

Am decis să creez mini-batch-uri de 16 iar rețeaua are nivelele după cum urmează

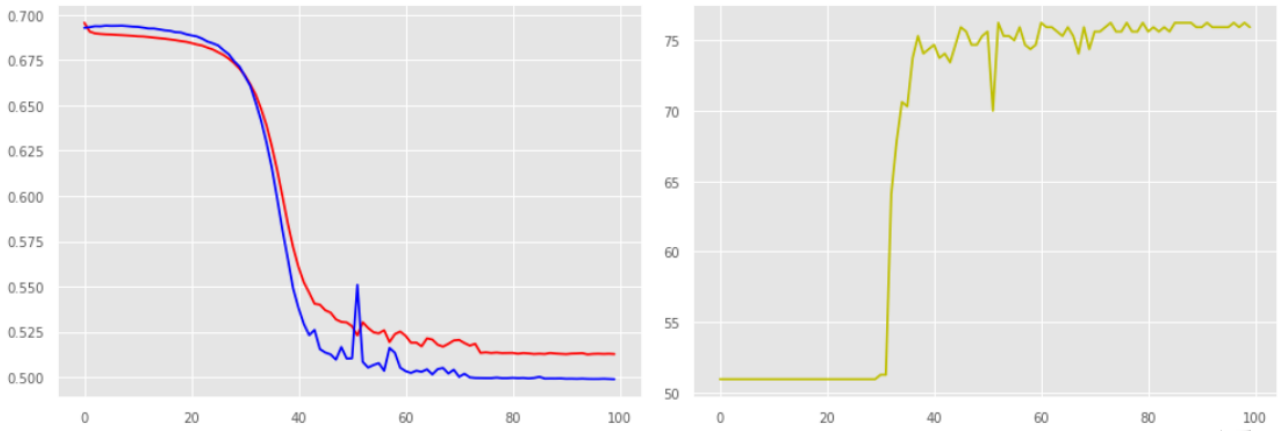
```
self.sequential= nn.Sequential( nn.Linear(11,50), nn.ReLU(),
    nn.Linear(50, 100), nn.ReLU(), nn.Linear(100, 30), nn.ReLU(),
    nn.Linear(30, 60), nn.ReLU(), nn.Linear(60, 2))
```

Mai jos se pot observa rezultatele la cea d-a 99-a epocă pentru cele 2 seturi de date:

```
[Train Epoch: 99, Batch: 70, Loss: 0.4717840850353241
[Train Epoch: 99, Batch: 71, Loss: 0.4064854681491852
[Train Epoch: 99, Batch: 72, Loss: 0.8047884106636047
[Train Epoch: 99, Batch: 73, Loss: 0.2937597632408142
[Train Epoch: 99, Batch: 74, Loss: 0.37233468890190125
[Train Epoch: 99, Batch: 75, Loss: 0.4384734034538269
[Train Epoch: 99, Batch: 76, Loss: 0.49716609716415405
[Train Epoch: 99, Batch: 77, Loss: 0.6041545867919922
[Train Epoch: 99, Batch: 78, Loss: 0.4950072765350342
[Train Epoch: 99, Batch: 79, Loss: 0.5798892378807068
[Train Epoch: 99, Batch: 80, Loss: 0.37721315026283264
[TRAIN] Epoch: 99 Loss:0.5126686947420239
```

```
[Validation set] Batch index: 15 Batch loss: 0.3621167838573456, Accuracy: 93.75%
============
============
[Validation set] Batch index: 16 Batch loss: 0.5524299144744873, Accuracy: 75.0%
============
============
[Validation set] Batch index: 17 Batch loss: 0.3792543113231659, Accuracy: 87.5%
============
============
[Validation set] Batch index: 18 Batch loss: 0.418857216835022, Accuracy: 75.0%
============
============
[Validation set] Batch index: 19 Batch loss: 0.48816347122192383, Accuracy: 87.5%
============
============
[Validation set] Batch index: 20 Batch loss: 0.4265097975730896, Accuracy: 68.75%
============
[Validation set] Loss: 0.4987938076257706, Accuracy: 75.9375%
```

În continuare am afișat comparativ loss-urile pentru training set și pentru validation set și acuratețea pe set-ul de test/validation



Ca ultim pas am făcut o verificare pe un element oarecare din dataset-ul de test:

```
index_of_test=30
print(f"Neural network response is: {try_a_single_example_with_the_network(index_of_test).item()}")
print(f"Actual response is: {validationDataset[index_of_test][1].view(-1).item()}")
```

```
Neural network response is: 1
Actual response is: 1
```

4.3 Decision Tree

Pentru a face un demo acestui algorim am ales un dateset online winequality-white, coloanele acestuia se pot observa mai jos:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	1
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	1
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	1
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	1
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	1

Rezultatele obținute sunt:

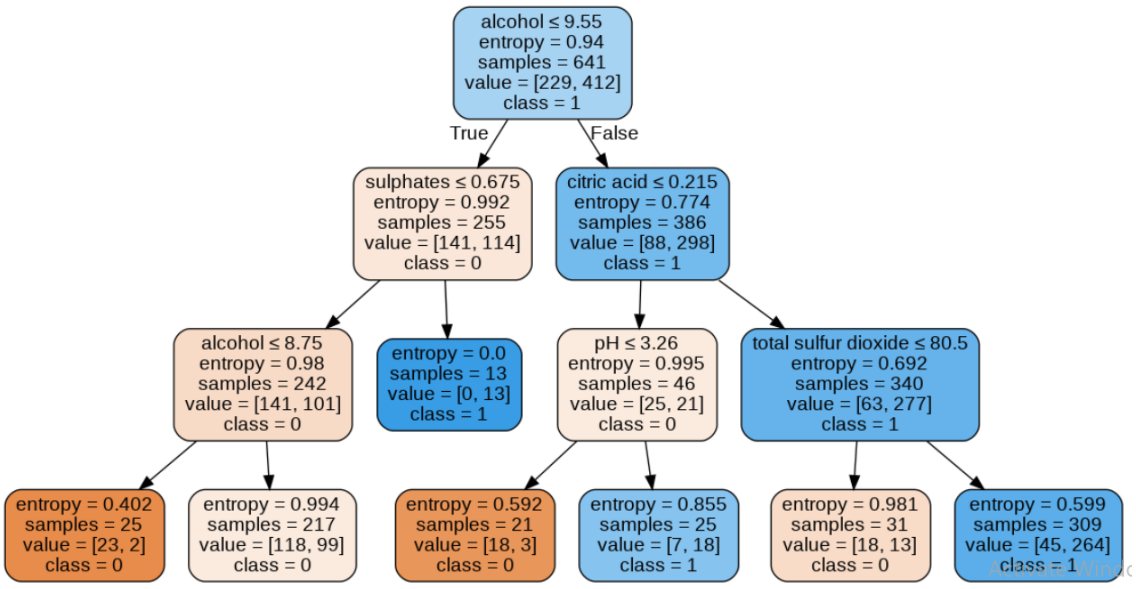
DecisionTreeClassifier(criterion="entropy", max_depth=3)

```
Accuracy: 0.7018181818181818
```

DecisionTreeClassifier(criterion="gini",splitter="random", max_depth=8)

```
Accuracy: 0.6727272727272727
```

Arborele obținut în urma primei variante de DecisionTreeClasifier() este:



4.4 K-Means Clustering

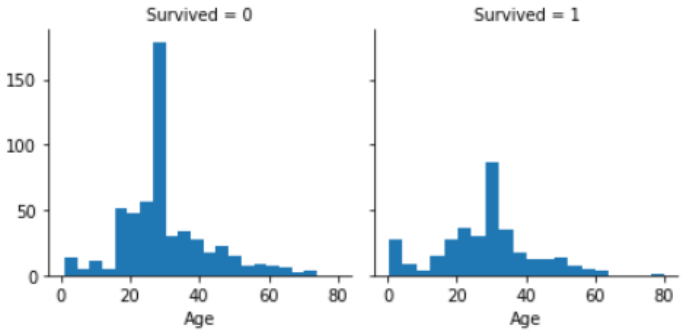
Pentru a verifica funcționalitatea algoritmului am folosit dataset-ul Titanic:

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Fare
0	1	0	3	1	22.0	1	0	7.2500
1	2	1	1	0	38.0	1	0	71.2833
2	3	1	3	0	26.0	0	0	7.9250
3	4	1	1	0	35.0	1	0	53.1000
4	5	0	3	1	35.0	0	0	8.0500

În imaginea de mai jos se pot observa câteva dependente, cum ar fi procentajul de supraviețuitori in funcție de sex și clasă socială, și numărul de supraviețuitori și morți în funcție de vărstă:

	Pclass	Survived
0	1	0.629630
1	2	0.472826
2	3	0.242363

	Sex	Survived
0	0	0.742038
1	1	0.188908



Rezultatele obținute sunt:

KMeans(n_clusters=2, max_iter=600, algorithm = 'auto')

```
Accuracy: 0.5084175084175084
```

KMeans(algorithm = 'full', max_iter=400, n_clusters=2, precomput_distances=True, random_state=1234)

```
Accuracy: 0.6262626262626263
```

4.5 K-Nearest Neighbors

Pentru a testa acest algoritm am folosit din nou dataset-ul winequality-red, de această dată în varianta sa originală, calitatea nu este binară (vinul este bun sau nu) cu reprezintă o notă (de la 3 la 8) care reprezintă cât de bun este acest vin.

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

Rezultatele obținute sunt:

KNeighborsClassifier(n_neighbors=3)

```
Accuracy: 0.4625
```

KNeighborsClassifier(n_neighbors=5, weights="distance")

```
Accuracy: 0.6375
```

4.6 Support Vector Machine

Demo-ul acestui algoritm l-am realizat pe dataset-ul winequality-red (varianta cu 2 clase)

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	0
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	0
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	0
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	1
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	0

Rezultatele obținute sunt după cum urmează:

svm.SVC(kernel='linear')

```
Accuracy: 0.7166666666666667
Precision: 0.7759336099585062
Recall: 0.6951672862453532
```

svm.SVC(kernel='poly', degree=5, gamma='scale', coef0=0.1)

```
Accuracy: 0.74375
Precision: 0.7967479674796748
Recall: 0.7286245353159851
```

Appendix A

Your original code

This section should contain only code developed by you, without any line re-used from other sources.

A.1 Perceptron

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

labels = np.array([0, 1, 0, 0, 1, 0, 1])

perceptron = Perceptron(3)
perceptron.train(training_inputs, labels)

inputs = np.array([1, 0, 1])
print("Nu face prezenta, nu e de la 8:00 si vine Mariel => %d"
    % perceptron.predict(inputs))
#=> Merg
inputs = np.array([0, 1, 1])
print("Nu face prezenta, e de la 8:00 si vine Mariel => %d"
    % perceptron.predict(inputs))
#=> Nu merg
inputs = np.array([1, 0, 0])
print("Face prezenta, nu e de la 8:00 si nu vine Mariel => %d"
    % perceptron.predict(inputs))
#(Ex nou) => Merg
```

A.2 MLP

```
df=pd.read_csv("/content/winequality-red.csv")
df.head()

#Statisticile pot fi salvate În format html
prof=ProfileReport(df,minimal=True)
prof.to_file(output_file='/content/output-min.html')
prof

#Selectăm datele de intrare in retea eliminand ultima coloană din csv
X = df.drop("quality", axis=1)
#obținem etichetele pentru date salvand ultima coloana
y = df['quality']

trainDataset=Dataset(X_train, y_train)
trainLoader=DataLoader(dataset=trainDataset,
                       batch_size=16,
                       shuffle=True,
                       num_workers=1)

validationDataset=Dataset(X_test, y_test)
validationLoader=DataLoader(dataset=validationDataset,
                       batch_size=16,
                       shuffle=True,
                       num_workers=1)

class WineNN(nn.Module):
    def __init__(self):
        super(WineNN, self).__init__()

        #Sequential oferă o alternativă mai estetică a codului
        #Rețeaua noastră are 2 neuroni pentru output.
        #Unul va prezice probabilitatea pentru cazul afirmativ al bolii, iar celălalt va
        self.sequential= nn.Sequential(
            nn.Linear(11,50),
            nn.ReLU(),
            nn.Linear(50, 100),
            nn.ReLU(),
            nn.Linear(100, 30),
            nn.ReLU(),
            nn.Linear(30, 60),
            nn.ReLU(),
            nn.Linear(60, 2)
        )

    def forward(self, x):
        return self.sequential(x)
```

A.3 Decision Tree

```
col_names = ['fixed acidity', 'volatile acidity', 'citric acid',
    'residual sugar', 'chlorides', 'free sulfur dioxide',
    'total sulfur dioxide', 'density', 'pH', 'sulphates', 'alcohol',
    'quality']
# load dataset
wine = pd.read_csv("/content/winequality-white.csv")
wine.columns=col_names

sc = MinMaxScaler((-1, 1))
X_train_sc = sc.fit_transform(X_train)
X_test_sc = sc.transform(X_test)

# Create Decision Tree classifer object
clf = DecisionTreeClassifier(criterion="gini",splitter="random", max_depth=8)
```

A.4 K-Means Clustering

```
centroids = np.asarray([X_scaled[0],X_scaled[1]])
kmeans = KMeans(algorithm = 'full', max_iter=400, n_clusters=2,
    precompute_distances=True, random_state=1234)

scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
kmeans.fit(X_scaled)
```

A.5 KNN

```
sc = MinMaxScaler((-1, 1))
X_train_sc = sc.fit_transform(X_train)
X_test_sc = sc.transform(X_test)

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5, weights="distance")

#Train the model using the training sets
knn.fit(X_train_sc, y_train)

#Predict the response for test dataset
y_pred = knn.predict(X_test_sc)

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

A.6 SVM

```
sc = MinMaxScaler((-1, 1))
X_train_sc = sc.fit_transform(X_train)
X_test_sc = sc.transform(X_test)

#Create a svm Classifier
clf = svm.SVC(kernel='poly', degree=5, gamma='scale', coef0=0.1)

#Train the model using the training sets
clf.fit(X_train_sc, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test_sc)
```

Bibliography

[1] Perceptron in 19 lini de cod.
[2] Rețele neurale de la laborator.
[3] Analizare date cu Pandas Profiler.
[4] Decision Trees.
[5] Random Forest.
[6] K-Means Clustering.
[7] Extensi K-Means.
[8] K-Nearest Neighbors.
[9] Extensi KNN.
[10] Support Vector Machine.