Rapport Classification

[**KNN**](#_bpydj381lj7) **1**

[**Changements**](#_knznc9c2gdid) **3**

[Notion de distance](#_h7qjh8rqdzhi) 3

[Régression logistique](#_5l5o37m1osm) 4

[**Conclusion**](#_4v7bsmweg4k) **5**

# KNN

Ci dessous les méthodes que nous avons utilisé pour notre algorithme des K plus proches voisins (KNN), dans le cas classique de l’algorithme.

**def** euclidean\_distance(vector1, vector2):

*"""*

*Return the euclidean distance between two vectors*

**:param** *vector1: first vector*

**:param** *vector2: second vector*

**:return***: euclidean distance between the vectors*

*"""*

**def** get\_neighbours(X\_train, X\_test\_instance, k):

*"""*

*Get the k neighbors of an instance*

**:param** *X\_train: X for which the label is known*

**:param** *X\_test\_instance: X instance*

**:param** *k: give the number of neighbors to find*

**:return***: k nearest neighbors*

*"""*

**def** predictkNNClass(x, output,X\_train, y\_train):

*"""*

*Predict the class of an instance*

**:param** *x: instance to predict*

**:param** *output: nearest neighbors indexes*

**:param** *X\_train: X for which we know the label*

**:param** *y\_train: Respective label for X\_train*

**:return***: the predicted class*

*"""*

**def** kNN\_test(X\_train, X\_test, Y\_train, k):

*"""*

**:param** *X\_train: X for which we know the label*

**:param** *X\_test: X for which we want to know the label*

**:param** *Y\_train: Respective label for X\_train*

**:param** *k: give the number of neighbors*

**:return***: array of predicted classes*

*"""*

**def** prediction\_accuracy(predicted\_labels, original\_labels):

*"""*

*Calculate the prediction accuracy*

**:param** *predicted\_labels: predicted labels*

**:param** *original\_labels: real labels*

**:return***: accuracy between 0 and 1*

*"""*

# 

# Changements

Pour créer nos prédictions nous avons fait quelques changements sur l’algorithme KNN ou les données en elle-mêmes

## Notion de distance

Premièrement nous avons ajouté une notion de distance pour la prédiction des classes. En effet l’algorithme KNN infère la classe d’une instance en comptabilisant le mode des classes parmis ses k plus proches voisins. Nous avons ajouté donc une notion de distance. Ainsi, le poids accordé à chaque voisin est inversement proportionnel à sa distance par rapport à l’instance à prédire. D’où la méthode suivante :

**def** predictkNNClass(x, output,X\_train, y\_train):

*"""*

*Predict the class of an instance*

**:param** *x: instance to predict*

**:param** *output: nearest neighbors indexes*

**:param** *X\_train: X for which we know the label*

**:param** *y\_train: Respective label for X\_train*

**:return***: the predicted class*

*"""*

classVotes = {}

**for** i **in** range(len(output)):

*# inversely proportional to the distance*

**if** y\_train[output[i]] **in** classVotes:

classVotes[y\_train[output[i]]] += 1 / euclidean\_distance(x, X\_train[i])

**else**:

classVotes[y\_train[output[i]]] = 1 / euclidean\_distance(x, X\_train[i])

sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1), reverse=**True**)

**return** sortedVotes[0][0]

## 

## Régression logistique

Enfin nous avons réalisé une régression logistique sur nos données d'entraînement pour concevoir une dimension supplémentaire à nos instances. A savoir, la probabilité donnée par cette régression logistique d’appartenir à la classe Y. nous avons pu faire ceci aussi simplement car il n’y avait que deux classes possible à prédire mais nous aurions pu créer un 1vsAll aussi.

**class** LogisticRegression:

**def** \_\_init\_\_(self, X, y):

self.X = X

self.y = y

self.cost = **None**

self.weights, self.bias = **None**, **None**

self.learning\_rate = 0.2

self.epoch = 0

self.X\_bias = np.hstack((X, (np.ones((X.shape[0], 1)))))

**def** step\_logistic\_gradient(self):

*"""*

*One step of the gradient descent in logistic regression case*

*"""*

**def** logistic\_gradient(self, MAX\_EPOCH, learning\_rate=0.1):

*"""*

*Run the logistic regression*

*"""*

**def** get\_prediction(self, X=**None**):

*"""*

*Get predictions with current weights and bias*

**:param** *X: Array of parameters*

**:return***: predictions*

*"""*

**def** sigmoid(z):

*"""*

*Sigmoid function*

**:param** *z: input of the sigmoid function*

**:return***: sigmoid calculation*

*"""*

# Conclusion

Nous pouvons donc traiter nos données

*# get the data for training*

data = np.genfromtxt(**'training.csv'**, delimiter=**','**, dtype=str)

*# extract the coordinates X and the label for each point*

X, y = np.array(data[:, :-1]).astype(float), data[:, -1]

*# convert the label into 0 or 1 for the logistic regression as there are only 2 possible labels*

y\_binary = np.array([0 **if** label == **'N' else** 1

**for** \_, label **in** enumerate(y)])

*# create a logistic regression on data*

log\_reg = LogisticRegression(X, y\_binary)

log\_reg.logistic\_gradient(100000, learning\_rate=0.2)

*# get the probability for each sample to be a Y or a N*

prediction = log\_reg.get\_prediction()

*# add this probability as a new parameter and convert it into -10 and 10 to have a better weight on its parameter*

X = np.array([np.concatenate((np.array(X[i, :]), 20\*(np.array([pred - 0.5])))) **for** i, pred **in** enumerate(prediction)])

Nous avons testé notre algorithme pour k allant de 1 à 100, 50 fois pour avoir une moyenne de la précision pour chaque k.

*# test each k from k\_min to k\_max included*

k\_min = 1

k\_max = 100

predicted\_classes = [0] \* k\_max

final\_accuracies = [0] \* k\_max

**for** k **in** range(k\_min, k\_max + 1):

n = 50 *# number of test for each k*

print(k)

**for** step **in** range(n):

*# split the known samples into training and test sets (randomly and at each iteration)*

training\_idx = np.random.randint(X.shape[0], size=100)

test\_idx = []

[test\_idx.append(i) **for** i **in** range(X.shape[0]) **if** i **not in** training\_idx]

X\_train = X[training\_idx]

X\_test = X[test\_idx]

y\_train = y[training\_idx]

y\_test = y[test\_idx]

*# calculate the accuracy for k in this n-th test*

predicted\_classes[k - 1] = kNN\_test(X\_train, X\_test, y\_train, k)

final\_accuracies[k - 1] += prediction\_accuracy(predicted\_classes[k - 1], y\_test) /n

print(final\_accuracies[k - 1])

plt.figure(figsize=(15, 6))

plt.plot(range(k\_min, k\_max + 1), final\_accuracies[k\_min -1:])

plt.title(**"Plot of the prediction accuracy of KNN Classifier as a function of k (Number of Neighbours)"**)

plt.show()

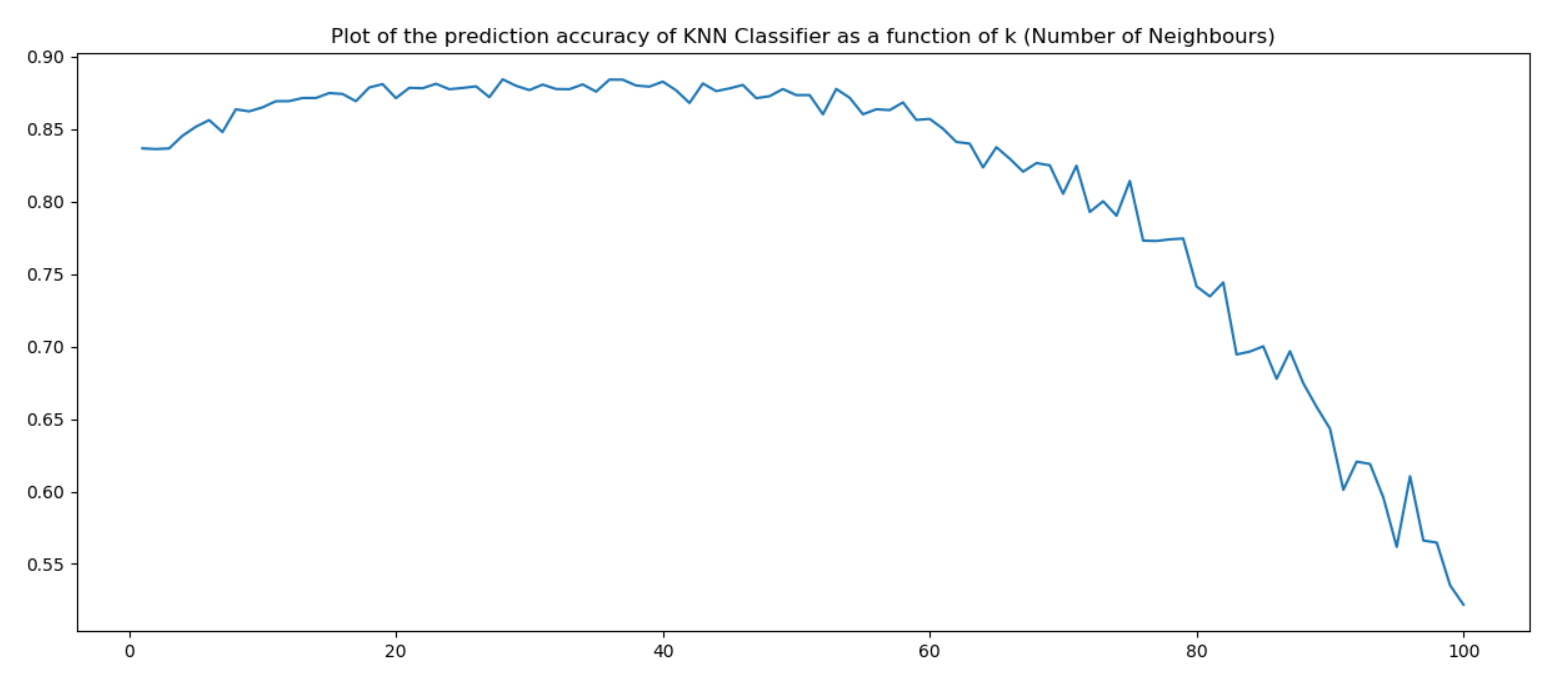
print(np.amax(final\_accuracies), np.argmax(final\_accuracies))

*# endregion*

*# get the best k*

k = np.argmax(final\_accuracies) + 1

Il en est ressorti que k = 27 était la meilleure solutions à notre problème avec une précision de 88.2% en moyenne.



Ainsi nous avons pu exporter nos données :

*# get the data to predict*

data = np.genfromtxt(**'predict.csv'**, delimiter=**','**, dtype=str)

XX = np.array(data[:, :]).astype(float)

*# create the parameter with the logistic regression*

prediction = log\_reg.get\_prediction(XX)

XX = np.array([np.concatenate((np.array(XX[i, :]), 20\*(np.array([pred - 0.5])))) **for** i, pred **in** enumerate(prediction)])

*# get the predicted classes*

predicted\_classes = kNN\_test(X, XX, y, k)

*# then, write them*

fname = **"THEVENOT\_PYTEL.txt"**

file = open(fname, **"w"**)

**try**:

**for** \_, pred **in** enumerate(predicted\_classes):

file.write(pred + **"\n"**)

**finally**:

file.close()