Premier modèle KNN

2 mars 2022

Auteurs :

Fabio Cassiano

Table des matières

Table des figures

1 Questionnaire de personnalité

Un test de personnalité a été mis au point sous python, afin de déterminer la personnalité des participants. Chaque personne doit répondre à 10 questions afin d'obtenir un score, sur lequel sera basé la mise en relation avec une personnalité.

Notre objectif ici est d'essayer de prédire la personnalité d'une personne en fonction de ses réponses aux différentes questions, en utilisant un modèle *K-nearest neighboors* (KNN) sans estimer de score. Pour ce faire la première étape est de se constituer un jeu de données. Plusieurs personnes ont ainsi réalisé les questionnaires 10 fois consécutives générant ainsi un fichier *csv* par participant.

2 Création du jeu de données (DataSet)

La création du DataSet se fait en réalisant la concaténation des différents fichier *csv*, afin d'avoir un DataSet suffisament conséquent. Cette étape est réalisé à l'aide de la fonction *create_DataSet()*. Cette fonction parcours le dossier contenant tout les fichiers *csv*, chacun d'entre eux est ensuite lu avec la fonction *read_csv()* du module **pandas**, le contenu est ajouté à la liste *data_list* qui est ensuite concaténé afin de renvoyer le DataSet (*data*).

2.1 Code Python - *create_DataSet()*

```
1  def create_DataSet():
2      file_list = os.listdir("./DataSet/")
3      data_list = []
4      for i in file_list:
5          data_list.append(pd.read_csv("./DataSet/{}".format(i)))
6
7      data = pd.concat(data_list)
8
9      # #
       ------------------------------------------------------------
10     # # ou pour enregistrer un nouveau csv :
11     # data.to_csv( "data.csv", index=False, encoding='utf-8-sig')
12     # #
       ------------------------------------------------------------
13
14     return data
```

3 Préparation du jeu de données

La préparation du jeu de données est l'étape la plus importante, elle conditionne l'apprentissage du modèle ce qui impactera par conséquent les performances de ce dernier. Afin de préparer le DataSet au mieux on réalise 3 étapes essentiel :

1. Suppression des valeurs erronés.

2. Remplacement des valeurs manquantes.
3. Harmonisation des données.

3.1 Suppression des valeurs erronés

Après avoir analysé le code du questionnaire de personnalité, il a été constaté que les utilisateurs ne sont pas limité dans leur réponse, c'est à dire qu'il peuvent répondre avec des valeurs non pris en compte dans le calcul du score. Il est donc essentiel, pour l'entraînement du modèle, de supprimer ces valeurs.

Pour ce faire, les valeurs valides ont été identifiées. Dans notre projet les valeurs sont les suivantes : *"a", "b", "c", 1, 2, 3*. Toutes les autres valeurs sont donc identifier et remplacer pars des *NaN*, afin d'être traiter dans l'étape suivante.

3.2 Remplacement des valeurs manquantes

Il a également été remarqué dans le code du questionnaire, que l'utilisateur peut laisser les champs vide, ce qui engendre des valeurs manquantes (*NaN*) dans le jeu de données. Ces données doivent être traiter afin que le modèle puisse être entraîner. Pour ce faire deux solutions sont possibles :

— Supprimer les observations contenant des données manquante.
— Remplacer les données par une autre valeur (moyenne, min, max, mode, etc..)

Dans notre cas, la suppression des données n'est pas envisageable au vu du nombre d'observation qui constitue notre jeu de données. Cela aurait pour impact de réduire fortement notre DataSet, ce qui empêcherait notre modèle d'être entraîner correctement.

Les données qui composent nos DataSet, sont des données de type qualitative. Il a donc été décidé de remplacer les données manquantes par le mode de chaque *features*. Il aurait également été possible de remplacer les *NaN*, par la valeur *"b" ou 2*, qui sont les entrées qui n'impacte pas le calcul du score dans le code du questionnaire.

3.3 Encodage des données

La dernière étape a effectuer pour finir la préparation du DataSet est l'encodage des données, qui consiste à harmoniser les différentes *features*. Pour ce faire plusieurs méthodes sont possible, il est possible d'encoder les données avec la fonction *OneHotEncoder()*, de la librairie **Sklearn**, ce qui encodera les valeurs de manière binaires. Une autre possibilité étant de remplacer les valeurs *"a", "b", "c"* par leurs correspondance numérique *1, 2, 3*. Cette dernière solution est celle pour laquelle nous avons opté.

Enfin pour finir, le DataSet est divisé en un jeu d'entraînement (*X_ train*) et un jeu de test (*X_ train*), cela à l'aide de la fonction *train_ test_ split()* de la librairie **sklearn**. Le DateSet est maintenant près pour entraîner notre modèle.

4 Développement et entraînement d'un modèle KNN

4.1 Principe du KNN

Le modèle que est développé ici, est celui des K plus proche voisin (KNN). Ce modèle a pour principe de calculer les distances entre une valeurs à tester avec les valeurs du jeu d'entraînement. Les k-distances les plus faible sont ensuite sélectionné, k étant le nombre de valeurs choisis lors de la création du modèle. La valeur testé est ensuite associé au label majoritaire présent dans les k plus faibles distances (voir Fig. 1.

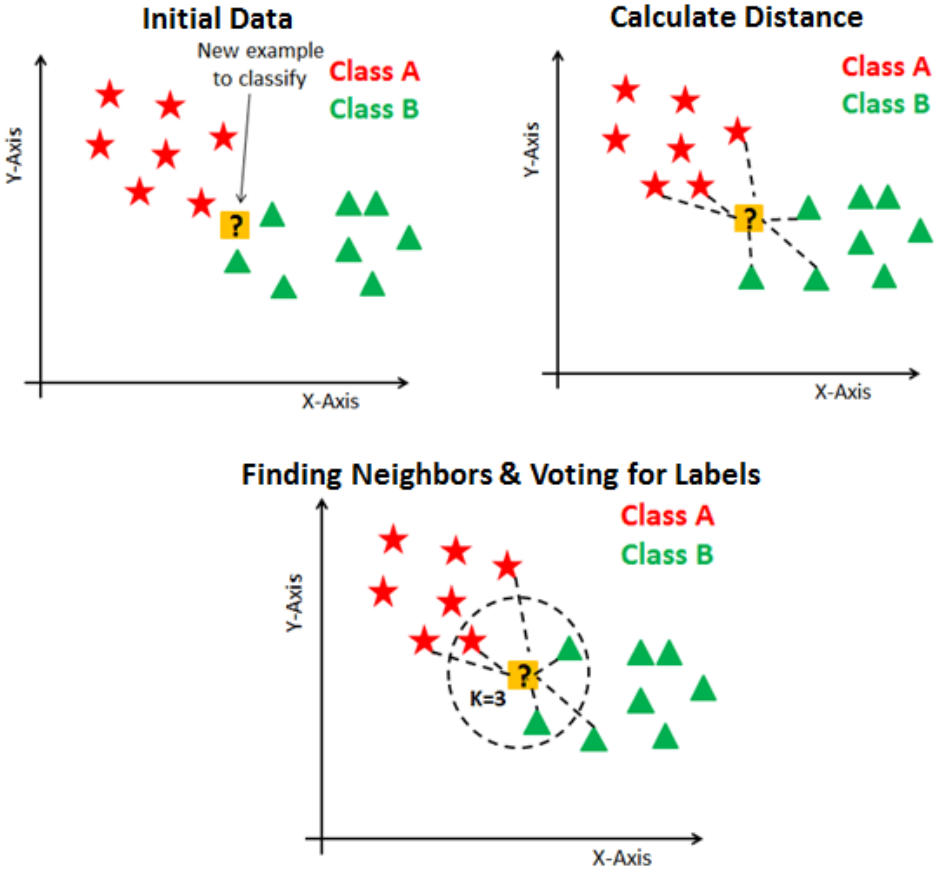

FIGURE 1 – Représentation du principe du modèle de KNN - *source : https ://www.datacamp.com.*

Les distances entre deux valeurs peuvent se calculer de différentes façon (voir Fig. 2). Dans notre étude nous nous intéresserons au méthode suivante : la distance euclidienne (eq. 1), la distance de manhattan (eq. 2), ou encore la distance de minkowski (eq. 3).

$$distance_{euclidienne} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{1}$$

$$distance_{manhattan} = \sum_{i=1}^{n}|x_i - y_i| \tag{2}$$

$$distance_{minkowski} = \sqrt[p]{\sum_{i=1}^{n}(x_i - y_i)^p} \tag{3}$$

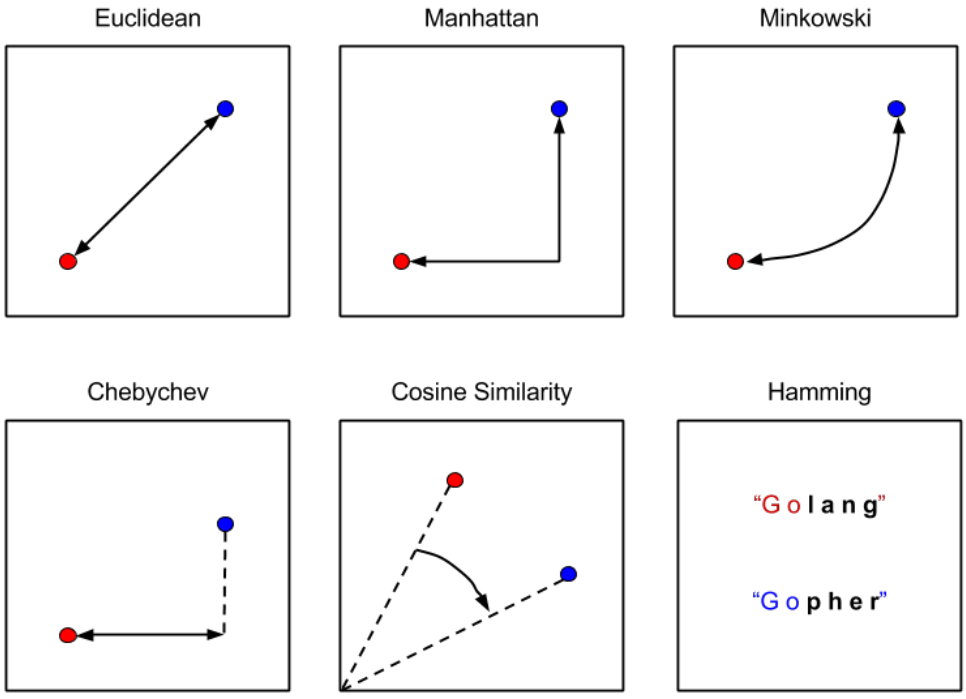

FIGURE 2 – Modélisation du calcul des différentes distances - *source : tuhinmukherjee74 sur medium.*

Dans cette partie nous entraînons deux modèles KNN, le premier étant celui que nous avons développé manuellement (from scratch), et le second est celui présent dans la librairie **sklearn**, le modèle *KNeighborsClassifier()*.

4.2 KNN from scratch

Le modèle KNN from scratch a été développé sous la forme d'une *class*. Cette *class* ***KNN()*** est composé de différentes méthodes permettant d'entraîner le modèle, mais égalment de calculer les distances, afin d'effectuer une prédiction, et même de calculer l'*accuracy* de la prédiction. Ce modèle s'utilise avec des données de type *array*, contenant uniquement des *integer*.

Ce modèle s'utilise de la manière suivante :

1. Initialisation d'une instance (création d'un modèle) : cela permet d'initialiser le modèle avec différentes propriétés.
2. Entraîner le modèle avec le jeu d'entraînement (X_train), les labels correspondant (y_label) et la méthode de calcul de distance désirée ($metric$).
3. Prédiction des labels du jeu de test (X_test).
4. Estimation de l'*accuracy* du modèle en comparant les vrai labels du jeu test (y_test) à celles prédites.

4.2.1 Code Python - *class* KNN()

```
1  class KNN:
2
3      def __init__(self):
4          self.__label_train = []
5          self.__X_train = []
6          self.__metric = 'euclidean'
7          self.__p = []
8          self.__label = []
9          self.__confusion_matrix = []
10         self.__format_model = {}
11
12
13     def distance(self, **kwargs):
14         '''
15         distance permet de calculer la distance d'un échantillon par
       rapport aux autres.
16
17         Paramètres
18         ----------
19         metric: {'Euclidean', 'Manhattan', 'Minkowski'}
20             methode à utiliser pour calculer la distance
21         '''
22
23         X_test = kwargs.get("X_test", None)
24
25         if self.__metric.lower() == 'euclidean':
26             if len(X_test) == 1:
27                 d = np.sqrt(np.sum((self.__X_train-X_test)**2, axis
       =1))
28                 return np.array(d)
```

```
            else:
                d = []
                for x in X_test:
                    d.append(np.sqrt(np.sum((self.__X_train-x)**2,
axis=1)))
                return np.array(d)
        elif self.__metric.lower() == 'manhattan':
            if len(X_test) == 1:
                d = np.sqrt(np.sum(abs(self.__X_train-X_test), axis
=1))
                return np.array(d)
            else:
                d = []
                for x in X_test:
                    d.append(np.sqrt(np.sum(abs(self.__X_train-x),
axis=1)))
                return np.array(d)
        elif self.__metric.lower() == 'minkowski':
            if len(X_test) == 1:
                d = pow(np.sum(abs(self.__X_train-X_test)**self.__p,
 axis=1), 1/self.__p)
                return np.array(d)
            else:
                d = []
                for x in X_test:
                    d.append(pow(np.sum(abs(self.__X_train-x)**self.
__p, axis=1), 1/self.__p))
                return np.array(d)
        else:
            raise ValueError(f"metric prend en uniquement comme
valeur 'euclidean', 'manhattan', ou 'minkowski' (saisie {self.
__metric})")


    def target_format(self, Y_train):
        self.__label = Y_train.sort_values().unique()

        cpt = 1
        for k in self.__label:
            self.__format_model[k] = cpt
            cpt +=1

        target_formated = Y_train.replace(self.__format_model).
values

        return target_formated


    def train(self, X_train, label_train, **kwargs):
        self.__p = kwargs.get('p', 2)
        self.__metric = kwargs.get("metric", "euclidiean")
```

```
72          self.__label_train = self.target_format(label_train) #
    Formatage des labels
73          self.__X_train = X_train
74
75
76      def prediction(self, X_test, k=5):
77          d = self.distance(X_test = X_test)
78
79          if d.ndim == 1:
80              d = d.reshape(1,-1)
81
82          # ind = np.argsort(d,axis=0)[:k,:] # k : nombre de voisin
83          ind = np.argsort(d,axis=1)[:,:k] # k : nombre de voisin
84
85          ppv = []
86          for i in ind:
87              ppv.append(self.__label_train[i]) # .mode()
88
89          # ppv = list(map(list, zip(*ppv))) # transposé liste
90          ppv = np.array(ppv)
91
92          proba = []
93          for i in range(1,3+1):
94              proba.append(np.count_nonzero(ppv == i, axis=1)/k)
95
96          proba = np.array(proba).T
97
98          y_pred = np.argmax(proba, axis=1)+1
99
100         return y_pred
101
102     def accuracy(self, y_test, y_pred, **kwargs):
103
104         y_test = y_test.replace(self.__format_model).values
105         self.__confusion_matrix = confusion_matrix(y_test, y_pred)
106         error_rate = (1 - np.trace(self.__confusion_matrix)/np.sum(
    self.__confusion_matrix))*100
107         accuracy = 100-error_rate
108
109         if kwargs.get('resume', False):
110             self.resume(accuracy, error_rate)
111
112
113         return accuracy, error_rate
114
115
116     def resume(self, accuracy, error_rate):
117         if not(accuracy):
118             print("Le calcul de l'accuracy est nécessaire, utiliser
    la fonction d'instance pour la calculer")
119         else:
```

```
120
121             print("\n ======  Résumé des métrique  =====\n")
122             print("-----------------------")
123             print("Matrice de confusion")
124             print(self.__confusion_matrix)
125 
126             print("\n=======================")
127             print(f"\nAccuracy : {accuracy}")
128             print(f"\nTaux d'erreur : {error_rate}")
129             print("-----------------------")
130 
131             cpt = 0
132             for l in self.__label:
133                 P = self.__confusion_matrix[cpt,cpt]/self.
    __confusion_matrix.sum(axis=0)[cpt]
134                 R = self.__confusion_matrix[cpt,cpt]/self.
    __confusion_matrix.sum(axis=1)[cpt]
135 
136                 cpt += 1
137                 print(f"\nPrécision classe {l} : {P}")
138                 print(f"\nSensibilité classe {l} : {R}")
139                 print("-----------------------")
140 
141             print("\n=======================\n")
```

4.2.2 Résultats obtenus

Afin d'estimer la valeur de k et la métrique les plus optimals pour notre application, l'accuracy a été calculer pour chaque métrique en faisant varier la valeur k de 1 à 3O (voir Fig. 3). D'après les résultats obtenus les paramètres les plus adaptés à notre application sont, la métrique de *manhattan*, et un k de 3 ou 4.

Le paramètre p pour la métrique *minkowski*, a quant-à lui était étudié séparément afin de ne pas surcharger la représentation graphique. Cette étude a été réalisé avec un p variant de 3 à 5. D'après les résultats obtenus la valeur p=*3*, le modèle obtient de meilleur *accuracy* (86%) pour de k valant 5, 6 et 8.

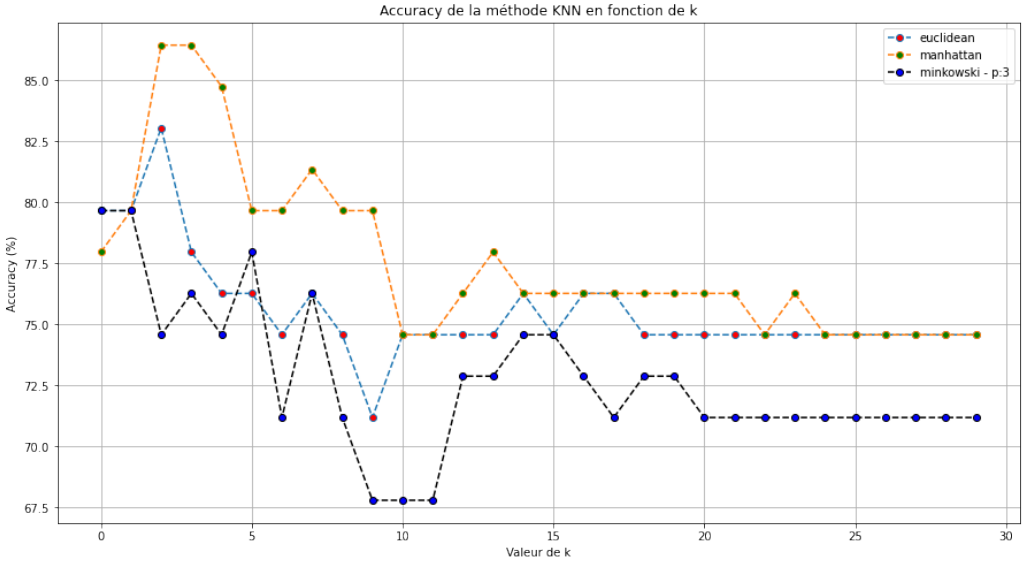

FIGURE 3 – Évaluation de l'*accuracy* du modèle from scratch en fonction du nombre de k, pour les trois méthodes de calcul de distance.

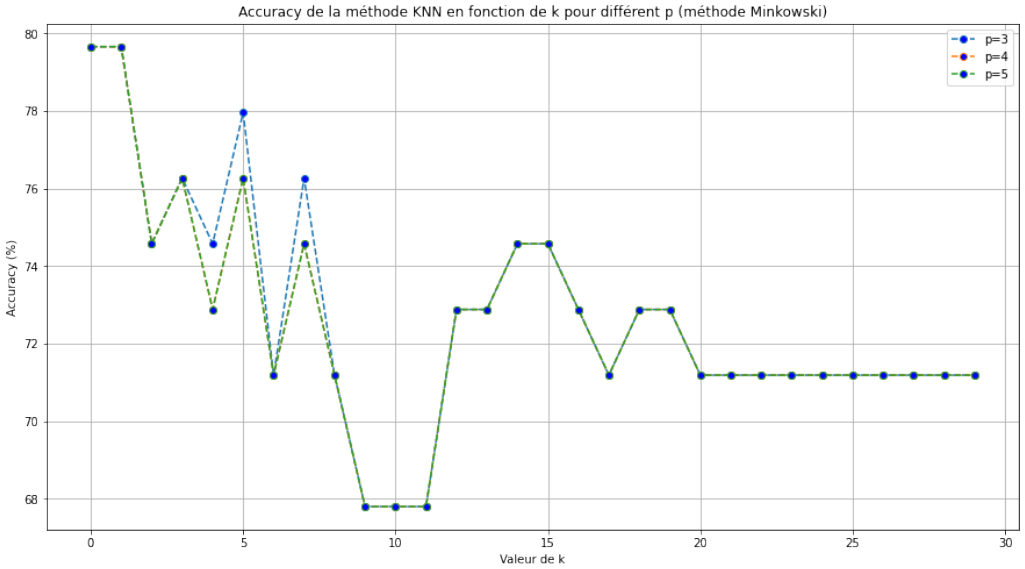

FIGURE 4 – Évaluation de l'*accuracy* du modèle from scratch en fonction du nombre de k, pour différentes valeurs de p

4.3 KNN avec sklearn

Les mêmes opérations, ainsi que les même jeu de données, ont été utiliser pour entraîner le modèle KNN de **sklearn**. Dans un premier temps la détermination des meilleurs hyperparamètres a été réalisé visuellement comme pour le modèle from skratch. D'après les résultats obtenus paramètres permettant d'obtenir la meilleur *accuracy* (83%) sont, la métrique *manhattan* et une valeur de k de 10.

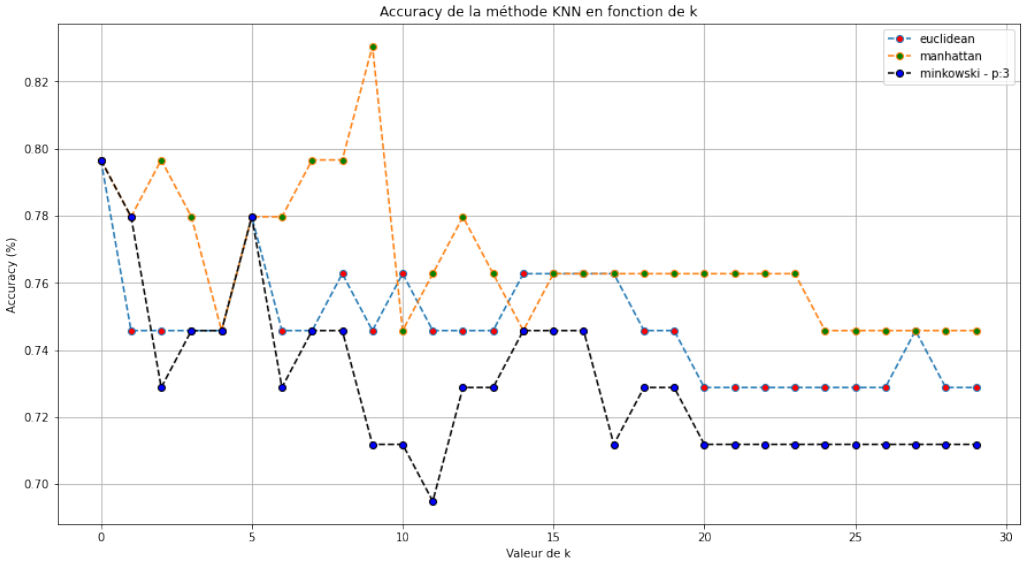

FIGURE 5 – Évaluation de l'*accuracy* du modèle from scratch en fonction du nombre de *k*, pour les trois méthodes de calcul de distance.

Une autre façon d'estimer les meilleurs paramètres est d'utilisé la méthode de *GridSearch*, combiné avec la méthode *K-fold*, grâce à la fonction *GridSearchCV()*. Cela permet de tester les différentes combinaison possible et de réalisé les test sur K sous ensembles de notre jeu d'entraînement, afin d'éviter le sur-apprentissage. Le résultat obtenus par cette méthode est le même que le précèdent, la métrique *manhattan* et un *k=10*.

5 Conclusion

Les résultats obtenu par les modèles entraînés assez bon, avec une *accuracy* dépassant les 80% pour les deux modèles.

D'après notre étude il semblerait que l'on obtient les meilleurs performances de prédiction à partir du modèle KNN développé manuellement. Cela pourrait être expliqué par le fait que plusieurs paramètre du modèle KNN de **sklearn**, on été laissé par défaut.

Cependant pour ce cas d'étude il serait préférable de conservé la classification par le

biais du score, pour ce faire il faudrait intervenir directement de le code du questionnaire afin de restreindre les réponses des utilisateurs.