Modélisation et prediction spatio-temporelle de l'expansion urbaine

Nathan Robert

Numéro SCEI : 30004

Contexte et motivations

- Phénomène d'urbanisation croissante, besoin d’outils de simulation pour la planification urbaine.
- Simuler l’évolution de la couverture terrestre entre deux périodes grâce à un modèle de type automate cellulaire.
- Utilisation d'algortithmes et de structures d'apprentissage non supervisé afin d'affiner la prédiction.

Exemple d'une simulation (ville de New-Delhi)

Plan de l'exposé

MODÉLISATION DE LA CROISSANCE URBAINE PAR AUTOMATES CELLULAIRES

ANALYSE SPATIALE ET CALIBRAGE DU MODÈLE

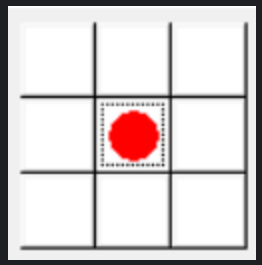
VERS UN MODÈLE HYBRIDE AVEC INTELLIGENCE ARTIFICIELLE

PERSPECTIVE D'AMÉLIORATION ET DE DÉVELOPPEMENT FUTUR

Utilisation et manipulation de fichiers Rasters

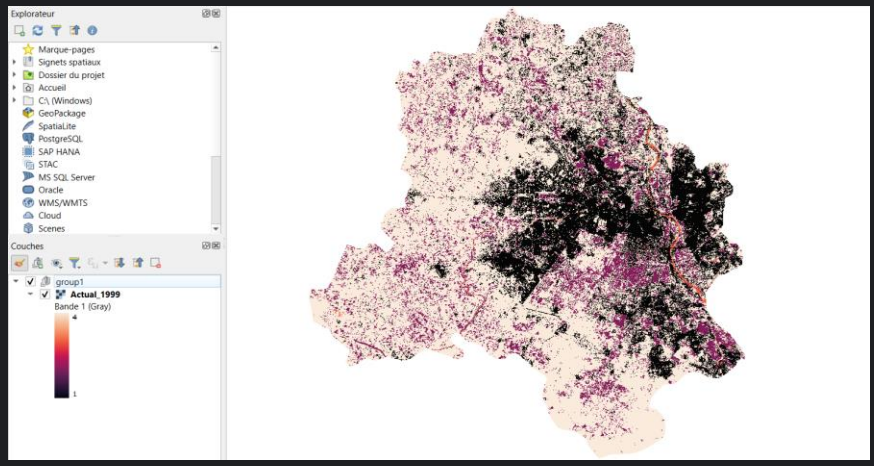
- La Geospatial Data Abstraction Library (GDAL) : gestion de formats de données géospatiales matricielles et vectorielles.
- Lecture des images de couverture terrestre (période finale VS période initiale).
- Lecture des facteurs de croissance (CBD, routes, pente, population, zones restreintes).

Règles de voisinage et automates cellulaires : le "Jeu de la vie"



Une cellule et ses 8 plus proches voisins :

- 3 voisins → naissance d'une case vide.
- 0,1,4,5,6,7,8 voisins → mort.
- 2 ou 3 voisins → survie.



- Chaque cellule représente une zone de terrain.

- Les voisins influencent l'évolution : si on a assez de voisins bâtis → la cellule devient bâtie.

- Les règles sont modifiées selon des facteurs géographiques (distance à la route, pente, etc.).

Une première implémentation : facteurs de croissance et couverture de terrain

```python
class FacteursCroissance:  # facteurs de croissance
    def __init__(self, *args):
        self.gf = dict()  # dictionnaire de matrices, chaque pixel
        # contenu correspond à une valeur propre au facteur considéré
        self.gf_ds = dict()  # dictionnaire des fichiers gdal des facteurs
        self.nFacteurs = len(args)  # nombre de facteurs
        self.n = 1
        for fichier in args:  # remplir les dictionnaires
            self.gf_ds[self.n], self.gf[self.n] = readraster(fichier)
            self.n += 1
        self.taille_correspondance()
```

```python
class Fitmodel:
    def __init__(self, landcoverClass: Landcover, facteursClass: FacteursCroissance):
        self.seuils = []
        self.seuil_construction = 0
        self.prediction = []  # tableau de cellules bâties
        self.landcovers = landcoverClass
        self.facteurs = facteursClass  # facteurs de croissance
        self.taille_correspondance()
        self.noyau = 3
```

- On implémente un objet growthfactors afin de manipuler et de comparer les données du terrain en entrée.

- La méthode performchecks() permet de comparer la taille des rasters en entrée : il est primordial qu'ils aient les mêmes dimensions.

- Elle contient la logique de prédiction (via predire()) en analysant les voisins d'un pixel (noyau de cellules).

- Les images de couverture terrestre (landcover) pour connaître les classes passées et présentes.

- La matrice de transition de l'état 1 à l'état 2 du territoire.

Une première implémentation : évolution d'un état à un autre

```
def matrice_transition(self):
    n_classes = max(self.arr_lc1.max(), self.arr_lc2.max())  # nombre total de classes
    self.matrice = np.zeros( shape: (n_classes, n_classes), dtype=int)
    for x in range(self.row):
        for y in range(self.col):
            t1_pixel = self.arr_lc1[x, y]
            t2_pixel = self.arr_lc2[x, y]
            if t1_pixel > 0 and t2_pixel > 0:  # Ignore les NoData (valeurs 0)
                self.matrice[t1_pixel - 1, t2_pixel - 1] += 1
    print("\nMatrice de transition calculée. Normalisation...")
    self.matriceNorm = np.zeros_like(self.matrice, dtype=float)
    for i in range(self.matrice.shape[0]):
        rangee_somme = self.matrice[i, :].sum()
        if rangee_somme > 0:
            self.matriceNorm[i, :] = self.matrice[i, :] / rangee_somme
        else:
            self.matriceNorm[i, :] = 0
```

- Matrice de transition de l'état 1 à l'état 2 : la cellule à l'indice [i,j] correspond à la probabilité que sa classe i évolue vers l'état j.

```
transition_probs =
[[0.90 0.05 0.03 0.02]  ← classe 1 (bâti)
 [0.20 0.60 0.10 0.10]  ← classe 2 (végétation)
 [0.10 0.10 0.70 0.10]  ← classe 3 (eau)
 [0.15 0.15 0.05 0.65]] ← classe 4 (autres)
```

Ces données sont issues de fichiers raster où chaque pixel a une classe :

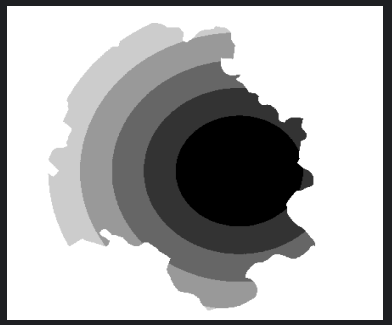
- **1** : zone bâtie
- **2** : végétation
- **3** : eau
- **4** : autres

L'objectif est d'étudier l'évolution entre 2 dates précises pour pouvoir prédire l'évolution future de la dernière date sur la même période.

Une première implémentation : facteurs de croissance

- **ROADDIST** (Distance au réseau routier) : Les cellules proches des routes sont plus susceptibles d'être urbanisées.
- **CBDDIST** (Distance au centre-ville) : Plus une cellule est proche du centre-ville, plus elle a de chances de devenir bâtie.
- **SLOPE** (Pente du terrain) : Les terrains en pente forte sont moins susceptibles d'être bâtis.
- **DEN19XX** (Densité de population en 19XX) : Une densité de population élevée peut favoriser l'urbanisation.
- **RESTRICTED ZONES** (Zones interdites à la construction) : Empêche toute croissance urbaine dans les zones protégées.

DISTANCE AUX ROUTES
Bande : [500; 20000]



DISTANCE AU CENTRE-VILLE
Bande : [10000, 35000]

Une première implémentation : prédiction du résultat

```python
def predire(self):
    self.prediction = deepcopy(self.landcovers.arr_lc2)
    marge_lat = math.ceil(self.kernelSize / 2)
    transition_probs = self.landcovers.matriceNorm  # Matrice de transition
    for x in range(marge_lat, self.rangee - (marge_lat - 1)):  # lignes
        for y in range(marge_lat, self.col - (marge_lat - 1)):  # colonnes
            kernel = self.landcovers.arr_lc1[x - (marge_lat - 1):x + marge_lat,
                     y - (marge_lat - 1):y + marge_lat]
            compteur_construction = np.sum(kernel == 1)
            classe_act = self.landcovers.arr_lc2[x, y]
            if classe_act < 1 or classe_act > transition_probs.shape[0]:
                continue
            proba_construire = transition_probs[classe_act - 1, 0]
```

- Extraction de la sous matrice Kernel de taille 3x3 qui entoure le pixel [x,y].
- Compteur_construction compte le nombre de cellules entourant le pixel qui sont de classe bâtie.

```python
if (compteur_construction >= self.seuil_construction) and (self.facteurs.gf[5][x, y] != 1):
    score = 0
    for facteur in range(1, self.facteurs.nfacteurs + 1):
        val = self.facteurs.gf[facteur][x, y]
        seuil = self.seuil[facteur - 1]
        if seuil < 0 and val <= abs(seuil):
            score += 1
        elif 0 < seuil <= val:
            score += 1
    if score >= 3 and proba_construire >= 0.25:
        self.prediction[x, y] = 1
```

- Un seuil positif (resp. négatif) signifie une valeur à (resp. 'ne pas') dépasser pour (resp. 'ne pas') construire une habitation.
- Le seuil de construction est renseigné en même temps que les autres facteurs, stockés dans une liste self.facteurs.

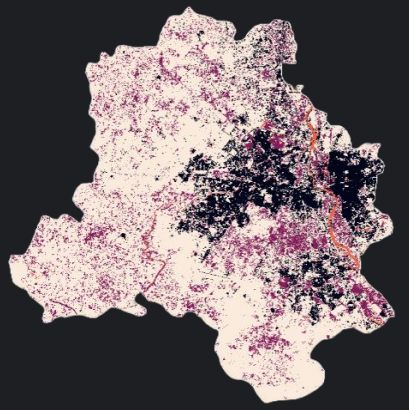
```python
couvertureTerrain = Landcover(file1, file2)
facteurs = FacteursCroissance( *args: cbd, road, pop01, slope, restricted)
prediction = Fitmodel(couvertureTerrain, facteurs)
prediction.setseuils( seuil_construction: 3,  *OtherseuilssInSequence: -15000, -10000, 8000, -3, -1)
```

Résultats de la première implémentation

- La correspondance se calcule en comparant chaque pixels dans la carte réelle et la carte prédite.

- Elle peut être trompeuse si un trop grand nombre de cases demeurent inchangées.

```
Croissance actuelle: 81, Croissance prédite : 112
Correspondance spatiale: 58.746040 %
```



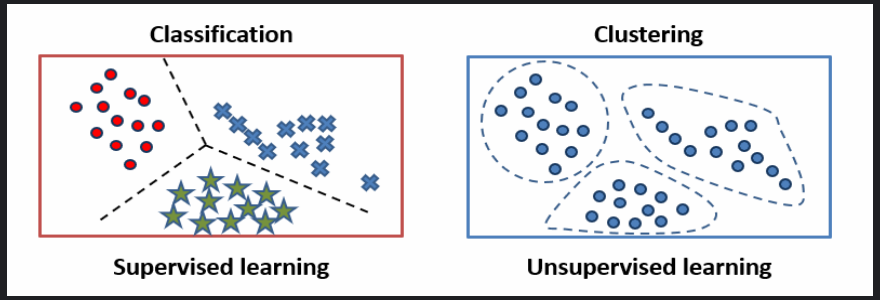
Ville de New-Delhi en 1999 prédite

Ville de New-Delhi en 1999

Amélioration : apprentissage non supervisé

Problématique actuelle :

- Difficulté à **capturer la complexité** des facteurs de croissance.
- Nécessité d'une méthode **plus adaptative** aux données.
- Absence d'apprentissage automatique dans le modèle initial.



- Objectif : identifier automatiquement les zones favorables à l'urbanisation.
- Approche : Regrouper les pixels similaires selon les facteurs de croissance.

L'algorithme des k-moyennes

Algorithme 15 : k moyennes

Données : Un ensemble E d'éléments $x \in \mathbb{R}^d$, un entier $k \in \mathbb{N}^\star$ indiquant le nombre de catégories recherchées

Résultat : Une partition $C_0, \ldots, C_{k-1}$ de E

Étape 1 : Tirer k points $\mu_0, \ldots, \mu_{k-1}$ de E au hasard ;

Étape 2 : $C_i \leftarrow \varnothing$ pour $i \in [\![0, k-1]\!]$;

pour $x \in E$ **faire**

- $j_{\min} \leftarrow 0$;
- $d_{\min} \leftarrow dist(x, \mu_0)$;
- **pour** $j \in [\![1, k-1]\!]$ **faire**
  - **si** $d_{\min} > dist(x, \mu_j)$ **alors**
    - $j_{\min} \leftarrow j$;
    - $d_{\min} \leftarrow dist(x, \mu_j)$;
- $C_{j_{\min}} \leftarrow C_{j_{\min}} \cup \{x\}$;

pour $j \in [\![0, k-1]\!]$ **faire**

- $\mu_j \leftarrow$ barycentre(C_j) ;

si *les barycentres* μ_j *ont changé* **alors**

- recommencer à partir de **l'étape 2** ;

renvoyer $C_0, \ldots, C_{k-1}$

- On sélectionne les pixels des clusters propices (où le taux de bâti est élevé).
- Si un pixel est dans un cluster propice et qu'il est entouré de suffisamment de pixels bâtis, alors il devient bâti.

- Données d'entrée : raster des facteurs (pente, accessibilité, etc.).
- But : créer des **clusters homogènes** en termes de caractéristiques.
- Résultat : chaque pixel appartient à un cluster → certains clusters favorisent plus l'urbanisation.

Utilisation de l'algorithme sur notre modèle

```python
def predire_kmoyenne(self, n_clusters, ratio_max_pixels_per_cluster=0.2, voisinage_seuil=3):
    print("Clustering des données avec K-means...")

    # Créer un masque pour le territoire (exclure le fond)
    territoire_mask = (self.landcovers.arr_lc2 != 0)
    # un pixel de fond a pour valeur 0
    proprietes = []
    coords = []  # pour garder la position des pixels valides

    for x in range(self.rangee):
        for y in range(self.col):
            if not territoire_mask[x, y]:
                continue  # Ignorer les pixels hors territoire
            vecteur = [self.facteurs.gf[i][x, y] for i in range(1, self.facteurs.nfacteurs + 1)]
            proprietes.append(vecteur)  # matrice de taille Npixels * Nfacteurs
            coords.append((x, y))

    proprietes = np.array(proprietes)
```

- Les pixels qui composent le fond du raster sont ignorés.
- Chaque pixel est associé à son vecteur de facteurs de croissance.

```python
# Standardisation
scaler = StandardScaler()
proprietes_scaled = scaler.fit_transform(proprietes)

kmeans = KMeans(n_clusters=n_clusters, n_init=10, random_state=0)
labels = kmeans.fit_predict(proprietes_scaled)
```

- Si les facteurs n'ont pas les mêmes unités ou échelles, les plus grands domineront les plus petits.
- La standardisation les met tous sur un pied d'égalité.

```python
# ratio de bâtis enregistré dans un dictionnaire cluster_stats
seuil_auto = 0.7 * max(cluster_stats.values())
propices = [c for (c, v) in cluster_stats.items() if v >= seuil_auto]
print(f"Clusters propices identifiés (seuil {seuil_auto:.2f}) : {propices}")
```

```python
cluster_stats = {0: 0.12, 1: 0.45, 2: 0.91}
```

- 12% des pixels du cluster 0 sont bâtis.
- 91% des pixels du cluster 2 sont bâtis → probablement une zone urbaine.

Algorithme des k-moyennes : quelques valeurs de k

K = 1

K = 2

K = 4

K = 6

Utilisation de l'algorithme sur notre modèle

```
self.prediction = deepcopy(self.landcovers.arr_lc2)
for c in propices:
    mask = (self.clustered == c) & (self.landcovers.arr_lc2 != 1)
    indices = list(zip(*np.where(mask)))
    random.shuffle(indices)
    limit = int(ratio_max_pixels_per_cluster * len(indices))
```

- np.where(mask) retourne les coordonnées [x, y] de tous les pixels candidats.
- On mélange les indices aléatoirement (pour ne pas avoir un biais spatial lors de la croissance).
- On ne veut modifier **qu'un certain pourcentage** des pixels candidats, défini par ratio_max_pixels_per_cluster.

```
changed = 0
for x, y in indices:
    if 1 <= x < self.rangee - 1 and 1 <= y < self.col - 1:
        kernel = self.landcovers.arr_lc2[x - 1:x + 2, y - 1:y + 2]
        voisins_constr = np.sum(kernel == 1)
        if voisins_constr >= voisinage_seuil:
            self.prediction[x, y] = 1
            changed += 1
    if changed >= limit:
        break
```

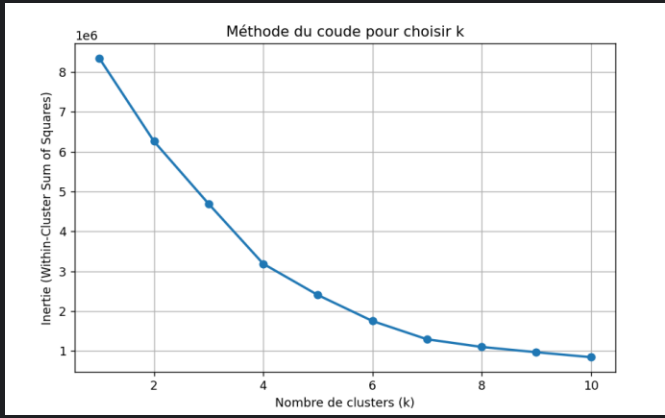
- On regarde un voisinage 3×3 autour du pixel [x, y].
- Si suffisamment de voisins sont déjà bâtis (≥ `voisinage_seuil`), on considère ce pixel comme propice à l'urbanisation, comme dans l'approche précédente.

Optimiser le nombre de clusters

```
for k in range(k_min, k_max + 1):
    self.prediction = deepcopy(self.landcovers.arr_lc2)
    print(f"\n🔁 Test avec k = {k}")
    self.predictWithKMeans(n_clusters=k)
    self.checkAccuracy()

    if self.accuracy > best_accuracy:
        best_accuracy = self.accuracy
        best_k = k
        best_prediction = deepcopy(self.prediction)
# Réappliquer la meilleure config
print(f"\n✅ Meilleur k trouvé : {best_k} avec une exactitude de {best_accuracy:.2f}%")
```

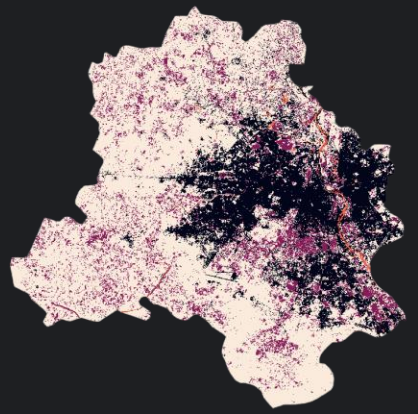
- On garde en mémoire le k et la prédiction si la précision est meilleure que la précédente.
- Problème d'attente évident, des appels peuvent par ailleurs être inutiles.



- On applique k-moyennes pour plusieurs valeurs de k (nombre de clusters), et on calcule l'inertie (somme des distances entre chaque point et son centre de cluster).
- Le "coude" est le point où la baisse de l'inertie ralentit nettement : ce k est considéré comme optimal, car il équilibre bien précision et complexité.

Résultats de l'implémentation k-moyennes

```
✅ Meilleur k trouvé : 5 avec une exactitude de 67.53%
Clustering des données avec K-means...
Cluster 0 : ratio de bâtis = 0.40
Cluster 1 : ratio de bâtis = 0.01
Cluster 2 : ratio de bâtis = 0.05
Cluster 3 : ratio de bâtis = 0.09
Cluster 4 : ratio de bâtis = 0.07
Clusters propices identifiés (seuil 0.05) : [0, 3, 4]
Croissance prédite terminée.
Croissance réelle : 81, Croissance prédite : 70
Exactitude spatiale : 67.529593
```



Ville de New-Delhi en 1999 prédite

Ville de New-Delhi en 1999

Perspective d'amélioration et de développement futur

- **Amélioration du traitement des bordures et pixels manquants** : Gérer plus finement les zones avec données manquantes ou hors territoire pour éviter des biais en prédiction.
- **Utilisation de données temporelles continues** : Passer d'une approche bitemporelle à une modélisation multi-temporelle pour mieux capter les dynamiques progressives.
- **Ajout de mécanismes d'apprentissage supervisé** : Utiliser des algorithmes comme les forêts aléatoires ou les réseaux de neurones pour affiner la prédiction du bâti.
- **Évaluation multi-critères de la qualité de prédiction** : Intégrer d'autres métriques spatiales (ex : F1-score spatial) pour une évaluation plus complète.

Annexe :

```
import os
import numpy as np
from osgeo import gdal
from copy import deepcopy
import matplotlib.pyplot as plt
import random
import math
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
```

```
# Définition de la fonction pour lire un fichier raster et retourner un tableau et la source de données
4 usages
def readraster(file):
    dataSource = gdal.Open(file)
    print(dataSource.GetRasterBand)
    band = dataSource.GetRasterBand(1)
    band = band.ReadAsArray()
    return (dataSource, band)


2 usages
def identicalList(inList):
    global logical
    inList = np.array(inList)
    logical = inList == inList[0]
    return sum(logical) == len(inList)
```

Annexe :

```
2 usages
class Landcover:
    def __init__(self, file1, file2, file3):
        self.matriceNorm = None
        self.matrice = None
        self.ds_lc1, self.arr_lc1 = readraster(file1)  # fichier gdal, tableau
        # des cellules bâties
        self.ds_lc2, self.arr_lc2 = readraster(file2)
        self.ds_lc3, self.arr_lc3 = readraster(file3)
        self.nClasses = 4  # nombre de classes de cellules possibles
        self.taille_correspondance()  # méthode de verification de la correspondance de
        # la taille des entrées

    1 usage
    def taille_correspondance(self):
        # Vérification des dimensions des rasters en entrée
        print("Vérification de la taille des rasters en entrée...")
        if (self.ds_lc1.RasterXSize == self.ds_lc2.RasterXSize) and (
                self.ds_lc1.RasterYSize == self.ds_lc2.RasterYSize):
            print("Les tailles des données de couverture terrestre correspondent.")
            self.ligne, self.col = (self.ds_lc1.RasterYSize, self.ds_lc1.RasterXSize)
        else:
            print("Les fichiers de couverture terrestre en entrée ont des hauteurs et largeurs différentes.")
        # Vérification du nombre de classes dans les images de couverture terrestre
        print("\nVérification des classes d'occupation du sol...")
        if (self.arr_lc1.max() == self.arr_lc2.max()) and (self.arr_lc1.min() == self.arr_lc2.min()):
            print("Les classes des fichiers de couverture terrestre en entrée correspondent.")
            self.nClasses = len(np.unique(self.arr_lc1))  # nb classes distinctes
        else:
            print("Les données de couverture terrestre en entrée ont des valeurs de classe différentes.")
```

Annexe :

```python
class FacteursCroissance:  # facteurs de croissance
    def __init__(self, *args):
        self.gf = dict()  # dictionnaire de matrices, chaque pixel
        # contenu correspond à une valeur propre au facteur considéré
        self.gf_ds = dict()  # dictionnaire des fichiers gdal des facteurs
        self.nfacteurs = len(args)  # nombre de facteurs
        self.n = 1
        for fichier in args:  # remplir les dictionnaires
            self.gf_ds[self.n], self.gf[self.n] = readraster(fichier)
            self.n += 1
        self.taille_correspondance()

    1 usage
    def taille_correspondance(self):
        print("\nVérification de la taille des facteurs de croissance en entrée...")
        lignes = []
        cols = []
        for n in range(1, self.nfacteurs + 1):
            lignes.append(self.gf_ds[n].RasterYSize)
            cols.append(self.gf_ds[n].RasterXSize)
        if identicalList(lignes) and identicalList(cols):
            print("Les facteurs en entrée ont le même nombre de lignes et de colonnes.")
            self.ligne = self.gf_ds[self.nfacteurs].RasterYSize
            self.col = self.gf_ds[self.nfacteurs].RasterXSize
        else:
            print("Les facteurs en entrée ont des dimensions différentes.")
```

Annexe :

```python
class Fitmodel:
    def __init__(self, landcoverClass: Landcover, facteursClass: FacteursCroissance):
        self.spatialAccuracy = 0
        self.construction_fictive = 0
        self.construction_reelle = 0
        self.seuils = []
        self.seuil_construction = 0
        self.prediction = []  # tableau de cellules bâties
        self.landcovers = landcoverClass
        self.facteurs = facteursClass  # facteurs de croissance
        self.taille_correspondance()
        self.noyau = 3

    1 usage
    def taille_correspondance(self):
        print("\nCorrespondance de la taille de la couverture terrestre et des facteurs de croissance...")
        if (self.landcovers.ligne == self.facteurs.ligne) and (self.landcovers.col == self.facteurs.col):
            print("Taille des rasters correspondante.")
            self.rangee = self.facteurs.ligne
            self.col = self.facteurs.col
        else:
            print("ERREUR ! La taille des rasters ne correspond pas, veuillez vérifier.")

    1 usage
    def setseuils(self, seuil_construction, *OtherseuilssInSequence):
        self.seuils = list(OtherseuilssInSequence)
        self.seuil_construction = seuil_construction
        if len(self.seuils) == (len(self.facteurs.gf)):
            print("\nSeuil défini pour les facteurs")
        else:
            print("ERREUR ! Veuillez vérifier le nombre de facteurs.")
```

Annexe :

```python
def matrice_transition(self):
    n_classes = max(self.arr_lc1.max(), self.arr_lc2.max())  # nombre total de classes
    self.matrice = np.zeros( shape: (n_classes, n_classes), dtype=int)
    for x in range(self.ligne):
        for y in range(self.col):
            t1_pixel = self.arr_lc1[x, y]
            t2_pixel = self.arr_lc2[x, y]
            if t1_pixel > 0 and t2_pixel > 0:  # Ignore les NoData (valeurs 0)
                self.matrice[t1_pixel - 1, t2_pixel - 1] += 1
    print("\nMatrice de transition calculée. Normalisation...")
    self.matriceNorm = np.zeros_like(self.matrice, dtype=float)
    for i in range(self.matrice.shape[0]):
        rangee_somme = self.matrice[i, :].sum()
        if rangee_somme > 0:
            self.matriceNorm[i, :] = self.matrice[i, :] / rangee_somme
        else:
            self.matriceNorm[i, :] = 0
```

Annexe :

```
def predire(self):
    self.prediction = deepcopy(self.landcovers.arr_lc2)
    marge_lat = math.ceil(self.noyau / 2)
    transition_probs = self.landcovers.matriceNorm  # Matrice de transition
    for y in range(marge_lat, self.rangee - (marge_lat - 1)):
        for x in range(marge_lat, self.col - (marge_lat - 1)):
            kernel = self.landcovers.arr_lc1[y - (marge_lat - 1):y + marge_lat,
                     x - (marge_lat - 1):x + marge_lat]
            compteur_construction = np.sum(kernel == 1)  # nombre de bâties
            classe_act = self.landcovers.arr_lc2[y, x]
            if classe_act < 1 or classe_act > transition_probs.shape[0]:
                continue  # ignorer des valeurs innatendues
            proba_construire = transition_probs[classe_act - 1, 0]  # [classe][se_construire]
            if (compteur_construction >= self.seuil_construction) and (self.facteurs.gf[5][y, x] != 1):
                score = 0
                for facteur in range(1, self.facteurs.nfacteurs + 1):
                    val = self.facteurs.gf[facteur][y, x]
                    seuil = self.seuils[facteur - 1]
                    if seuil < 0 and val <= abs(seuil):
                        score += 1
                    elif 0 < seuil <= val:
                        score += 1
                # on combine le score et la probabilité de construction
                if score >= 3 and proba_construire >= 0.25:  # le seuil de référence est arbitraire.
                    self.prediction[y, x] = 1
            if (y % 500 == 0) and (x % 500 == 0):
                print("rangee: %d, Col: %d, Builtup cells count: %d\n" % (y, x, compteur_construction), end="\r",
                      flush=True)
```

Annexe :

```python
def exactitude(self):
    # Exactitude statistique
    self.construction_reelle = difference_periode(self.landcovers.arr_lc2, self.landcovers.arr_lc3)
    self.construction_fictive = difference_periode(self.landcovers.arr_lc2, self.prediction)

    # Calcul de l'exactitude spatiale
    self.spatialAccuracy = 100 - (
            sum(sum(((self.prediction == 1).astype(float) - (self.landcovers.arr_lc2 == 1).astype(float)) != 0)) /
            sum(sum(self.landcovers.arr_lc2 == 1))
    ) * 100

    print("Croissance réelle : %d, Croissance prédite : %d" % (self.construction_reelle, self.construction_fictive))

    # Affichage de l'exactitude spatiale
    print("Exactitude spatiale : %f" % self.spatialAccuracy)

1 usage
def exportprediction(self, outFileName):
    driver = gdal.GetDriverByName("GTiff")
    outdata = driver.Create(outFileName, self.col, self.rangee, 1, gdal.GDT_UInt16)  # option : GDT_UInt16, GDT_Float32
    outdata.SetGeoTransform(self.landcovers.ds_lc1.GetGeoTransform())
    outdata.SetProjection(self.landcovers.ds_lc1.GetProjection())
    outdata.GetRasterBand(1).WriteArray(self.prediction)
    outdata.GetRasterBand(1).SetNoDataValue(0)
    outdata.FlushCache()
    outdata = None
```

Annexe :

```python
def distance(a, b):
    """Calcule la distance euclidienne entre deux points a et b."""
    return math.sqrt(sum((ai - bi) ** 2 for ai, bi in zip(a, b)))
```

1 usage

```python
def mean(points):
    """Calcule le barycentre (moyenne) d'une liste de points."""
    if not points:
        return []
    d = len(points[0])
    result = [0.0] * d
    for point in points:
        for i in range(d):
            result[i] += point[i]
    return [x / len(points) for x in result]
```

```python
def k_means(E, k, max_iter=100):
    """Algorithme k-moyennes sans numpy, version lisible."""
    # Étape 1 : Initialisation aléatoire des k centres
    mu = random.sample(E, k)
    clusters = []
    for _ in range(max_iter):
        # Étape 2 : Initialiser les k clusters vides
        clusters = [[] for _ in range(k)]
        # Assigner chaque point au cluster le plus proche
        for x in E:
            index_min = 0
            dist_min = distance(x, mu[0])
            for j in range(1, k):
                d = distance(x, mu[j])
                if d < dist_min:
                    dist_min = d
                    index_min = j
            clusters[index_min].append(x)
        # Recalculer les barycentres
        new_mu = []
        for cluster in clusters:
            new_mu.append(mean(cluster))
        # Vérifier si les barycentres ont changé
        if all(distance(mu[i], new_mu[i]) < 1e-6 for i in range(k)):
            break  # Convergence atteinte
        mu = new_mu  # Mettre à jour les centres
    return clusters
```

Annexe :

```python
def predire_kmoyenne(self, n_clusters, ratio_max_pixels_per_cluster=0.2, voisinage_seuil=3):
    print("Clustering des données avec K-means...")

    # Créer un masque pour le territoire (exclure le fond)
    territoire_mask = (self.landcovers.arr_lc2 != 0)
    # un pixel de fond a pour valeur 0
    proprietes = []
    coords = []  # pour garder la position des pixels valides

    for y in range(self.rangee):
        for x in range(self.col):
            if not territoire_mask[y, x]:
                continue  # Ignorer les pixels hors territoire
            vecteur = [self.facteurs.gf[i][y, x] for i in range(1, self.facteurs.nfacteurs + 1)]
            proprietes.append(vecteur)  # matrice de taille Npixels * Nfacteurs
            coords.append((y, x))

    proprietes = np.array(proprietes)

    # Standardisation
    scaler = StandardScaler()
    proprietes_scaled = scaler.fit_transform(proprietes)

    kmeans = KMeans(n_clusters=n_clusters, n_init=10, random_state=0)
    labels = kmeans.fit_predict(proprietes_scaled)

    # Créer une matrice clusterisée initialisée à -1 (hors territoire)
    self.clustered = -1 * np.ones( shape: (self.rangee, self.col), dtype=int)

    # Remplir la matrice clusterisée seulement aux indices valides
    for (y, x), label in zip(coords, labels):
        self.clustered[y, x] = label
```

Annexe :

```
cluster_stats = dict()
for c in range(n_clusters):
    mask = (self.clustered == c)
    if mask.sum() == 0:
        cluster_stats[c] = 0
        continue
    builtup_ratio = np.sum((self.landcovers.arr_lc2 == 1) & mask) / np.sum(mask)
    cluster_stats[c] = builtup_ratio

for c, ratio in cluster_stats.items():
    print(f"Cluster {c} : ratio de bâtis = {ratio:.2f}")

# ratio de bâtis enregistré dans un dictionnaire cluster_stats
seuil_auto = 0.125 * max(cluster_stats.values())
propices = [c for (c, v) in cluster_stats.items() if v >= seuil_auto]
print(f"Clusters propices identifiés (seuil {seuil_auto:.2f}) : {propices}")

# Prédiction avec règles de voisinage et limitation
self.prediction = deepcopy(self.landcovers.arr_lc2)
for c in propices:
    mask = (self.clustered == c) & (self.landcovers.arr_lc2 != 1)
    indices = list(zip(*np.where(mask)))
    random.shuffle(indices)
    limit = int(ratio_max_pixels_per_cluster * len(indices))
    changed = 0
    for y, x in indices:
        if 1 <= y < self.rangee - 1 and 1 <= x < self.col - 1:
            kernel = self.landcovers.arr_lc2[y - 1:y + 2, x - 1:x + 2]
            voisins_constr = np.sum(kernel == 1)
            if voisins_constr >= voisinage_seuil:
                self.prediction[y, x] = 1
                changed += 1
        if changed >= limit:
            break

print("Croissance prédite terminée.")
```

Annexe :

```python
def find_best_k(self, k_min=5, k_max=5):
    best_k = None
    best_accuracy = -1
    best_prediction = None

    for k in range(k_min, k_max + 1):
        self.prediction = deepcopy(self.landcovers.arr_lc2)
        print(f"\n🔄 Test avec k = {k}")
        self.predire_kmoyenne(n_clusters=k)
        self.checkAccuracy()

        if self.accuracy > best_accuracy:
            best_accuracy = self.accuracy
            best_k = k
            best_prediction = deepcopy(self.prediction)
    # Réappliquer la meilleure config
    print(f"\n✅ Meilleur k trouvé : {best_k} avec une exactitude de {best_accuracy:.2f}%")

    self.predire_kmoyenne(n_clusters=best_k)
    self.prediction = best_prediction
    self.checkAccuracy()
    self.exportprediction("meilleur_v2.tif")
```

Annexe :

```python
def methode_coude(self, k_min=1, k_max=10):
    print("Calcul des inerties pour la méthode du coude...")
    territoire_mask = (self.landcovers.arr_lc2 != 0)
    proprietes = []

    for y in range(self.rangee):
        for x in range(self.col):
            if not territoire_mask[y, x]:
                continue
            vecteur = [self.facteurs.gf[i][y, x] for i in range(1, self.facteurs.nfacteurs + 1)]
            proprietes.append(vecteur)

    proprietes = np.array(proprietes)
    scaler = StandardScaler()
    proprietes_scaled = scaler.fit_transform(proprietes)

    inertias = []
    K = range(k_min, k_max + 1)
    for k in K:
        kmeans = KMeans(n_clusters=k, n_init=10, random_state=0)
        kmeans.fit(proprietes_scaled)
        inertias.append(kmeans.inertia_)

    plt.figure(figsize=(8, 5))
    plt.plot( *args: K, inertias, 'o-', linewidth=2)
    plt.xlabel("Nombre de clusters (k)")
    plt.ylabel("Inertie (Within-Cluster Sum of Squares)")
    plt.title("Méthode du coude pour choisir k")
    plt.grid(True)
    plt.show()
```

```python
def afficher_clusters(caModel):
    plt.figure(figsize=(10, 8))
    cluster_map = np.copy(caModel.clustered).astype(float)
    cluster_map[cluster_map == -1] = np.nan  # Pixels hors territoire transparents
    plt.imshow(cluster_map, cmap='tab10')
    plt.title("proprietete des clusters K-means")
    plt.colorbar(label="Cluster ID")
    plt.axis("off")
    plt.show()
```

Annexe :

```
os.chdir(r"C:\Users\natha\OneDrive\Documents\TIPE")

# Entrée des fichiers GeoTIFF pour deux périodes de temps
file1 = "Actual_1989.tif"
file2 = "Actual_1994.tif"
file3 = "Actual_1999.tif"

# Entrée de tous les paramètres
cbd = "cbddist.tif"
road = "roaddist.tif"
restricted = "dda_2021_government_restricted.tif"
pop01 = "den1991.tif"
pop11 = "den2011.tif"
pop19 = "den2019.tif"
pop24 = "den2024.tif"
slope = "slope.tif"
ds = gdal.Open("slope.tif")

# Créer une classe de couverture terrestre qui prend les données de couverture terrestre pour deux périodes
myLandcover = landcover(file1, file2, file3)

# Créer une classe de facteurs qui configure tous les facteurs pour le modèle
myfacteurs = grangeethfacteurs( *args: cbd, road, pop01, slope, restricted)

# Initialiser le modèle avec les classes créées ci-dessus
caModel = fitmodel(myLandcover, myfacteurs)
```

```
# Selon l'exactitude statistique et spatiale affichée, les seuils doivent être ajustés
caModel.setThreshold( builtupThreshold: 3,  *OtherThresholdsInSequence: -15000, -10000, 8000, -3, -1)

# Exécuter le modèle
caModel.find_best_k()
afficher_clusters(caModel)

'''
caModel.checkAccuracy()
caModel.exportprediction("prediction_bestK.tif")'''

# Exporter la couche prédite
caModel.exportprediction('Nouveau_rapport.tif')
```