

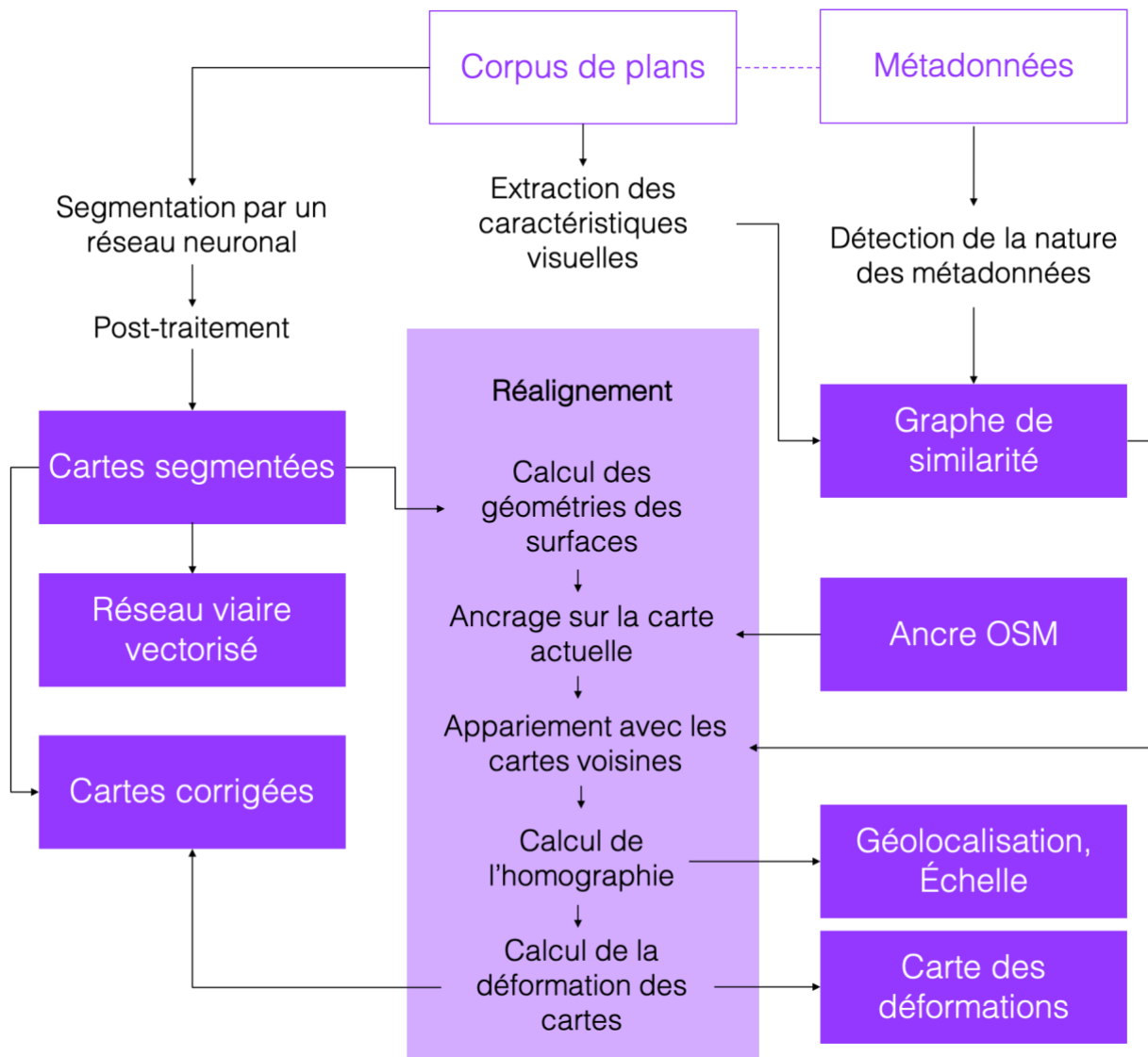
# JADIS

Documentation du programme  
et manuel d'utilisation

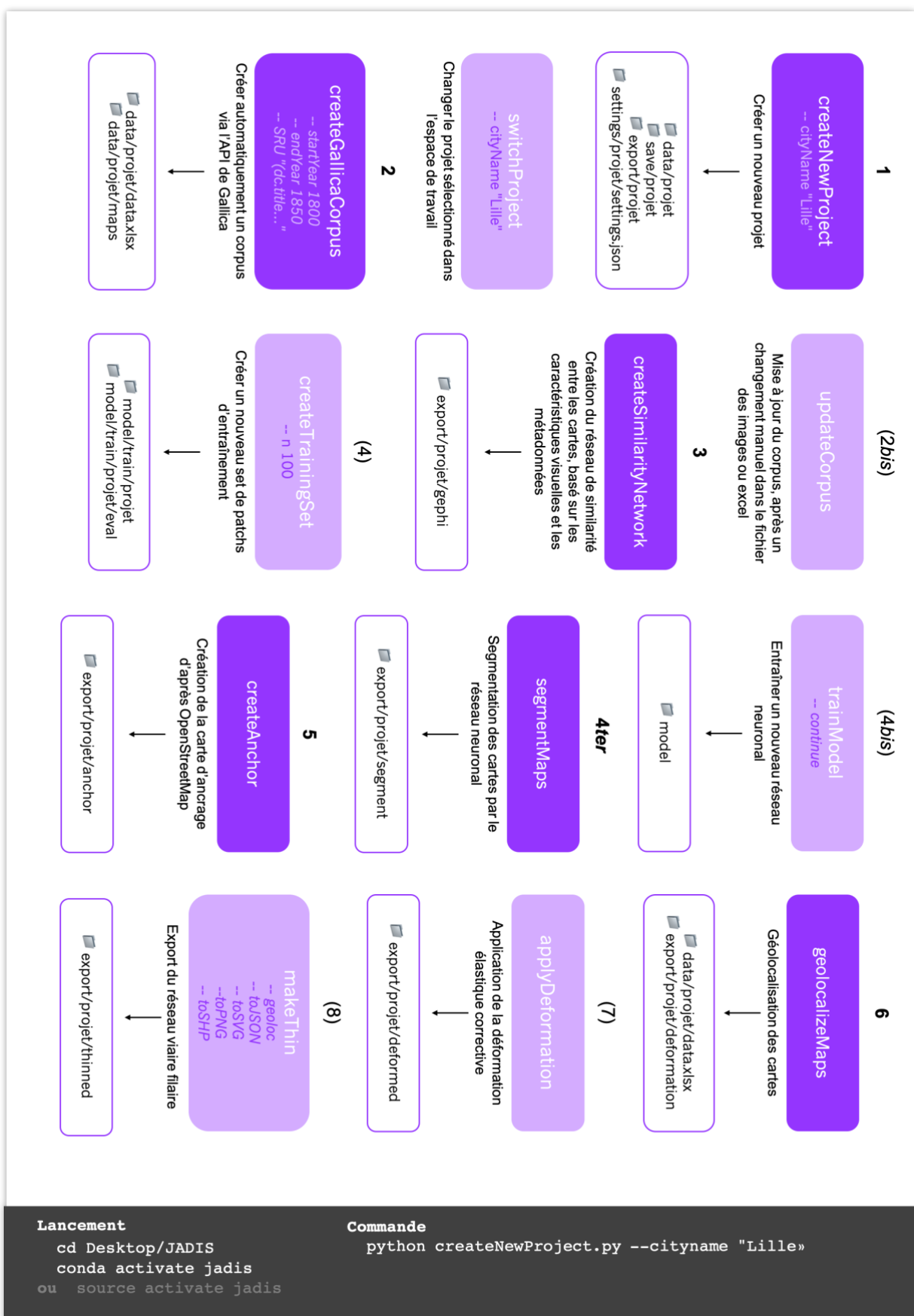
# Table des matières

<b>TABLE DES MATIERES.....</b>	<b>1</b>
<b>VUE GLOBALE DE L'ALGORITHME .....</b>	<b>2</b>
<b>FICHE-RÉSUMÉ DU PROGRAMME.....</b>	<b>3</b>
<b>NOTE SUR LES NIVEAUX D'UTILISATEURS-TRICES .....</b>	<b>4</b>
<b>EXIGENCES MATÉRIELLES .....</b>	<b>4</b>
<b>INSTALLER LE PROGRAMME (NIVEAU UTILISATEUR 🙋) .....</b>	<b>5</b>
<b>LANCER LE PROGRAMME (NIVEAU UTILISATEUR 🙋) .....</b>	<b>6</b>
<b>DOSSIERS ET FICHIERS.....</b>	<b>7</b>
<b>UTILISER LES FONCTIONS (NIVEAU UTILISATEUR 🙋).....</b>	<b>9</b>
INFORMATIONS GÉNÉRALES SUR LE FONCTIONNEMENT DU PROGRAMME .....	9
1. CRÉER UN NOUVEAU PROJET – CREATENewPROJECT .....	9
REPRENDRE UN PROJET EXISTANT – SWITCHPROJECT .....	10
2. CRÉER UN CORPUS DE CARTES – CREATEGALLICACORPUS.....	10
2BIS. AFFINER LE CORPUS – UPDATECORPUS .....	11
3. INTERCONNECTER LES CARTES – CREATESIMILARITYNETWORK .....	11
4. CRÉER UN NOUVEAU SET D'ENTRAÎNEMENT – CREATETRAININGSET .....	11
4BIS. ENTRAÎNER UN NOUVEAU RÉSEAU NEURONAL DE SEGMENTATION – TRAINMODEL.....	12
4TER. SEGMENTER ET SÉMANTISER LES CARTES – SEGMENTMAPS .....	13
5. CRÉER UNE ANCRE MODERNE DE RÉFÉRENCE – CREATEANCHOR .....	13
6. GÉOLOCALISER LES CARTES – GEOLOCALIZEMAPS .....	14
7. DÉFORMER LES CARTES – APPLYDEFORMATION.....	15
8. VECTORISER LES RUES – MAKETHIN .....	15
<b>CONCEVOIR UNE CHAÎNE DE TRAITEMENT (NIVEAU UTILISATEUR 🙋).....</b>	<b>16</b>
AUTOMATISER LA CHAÎNE DE TRAITEMENT (NIVEAU PROGRAMMEUR 🙋).....	17
<b>PARAMÈTRES (NIVEAU BIDOUILLEUR 🛠️) .....</b>	<b>18</b>
CORPUS.....	18
NETWORKING > METADATA.....	18
NETWORKING > VISUAL_FEATURES.....	19
NETWORKING > GEPHI.....	20
CNN > TRAINING_PARAMS.....	20
CNN > GPU, PATCH_SIZE.....	22
SEGMENT .....	22
ANCHOR .....	23
MATCHING.....	23
<b>CONTACT .....</b>	<b>25</b>

## Vue globale de l'algorithme



# Fiche-résumé du programme



## Note sur les niveaux d'utilisateurs·trices

### Niveau utilisateur 🧑

Niveau par défaut. Ce type d'utilisation ne nécessite aucune compétence en programmation et ne nécessite pas de savoir ouvrir et modifier des bases de données autres qu'un fichier excel. Les instructions en ligne de commande sont basiques et peuvent être copiées-collées telles quelles du manuel.

### Niveau bidouilleur 🛠️

Ce type d'utilisation intermédiaire ne nécessite pas de compétences en programmation. Il est accessible à un·e simple utilisateur·trice qui connaît déjà le fonctionnement de JADIS ou à un·e programmeur·euse qui n'en connaît pas bien le fonctionnement. Il est nécessaire de comprendre le fonctionnement de base des fichiers JSON et de posséder un logiciel pour les ouvrir (Visual Studio, Sublime Text) et y apporter des modifications simples.

### Niveau programmeur 🧑‍💻

Utilisation avancée du programme. Nécessite des compétences en python et une connaissance d'opencv, ainsi que des librairies de base : numpy, pandas. Des compétences en *computer vision* et en *deep learning* sont aussi nécessaires pour comprendre certaines parties de code.

## Exigences matérielles

	Minimum	Recommandé
Stockage (fonctionnement)	20 Go / 500 cartes	25 Go / 500 cartes
Stockage (archivage)	10 Go	
Mémoire vive RAM	8 Go	16 Go ou plus
Processeur	4x Intel Core i5 3.1 GHz	12x Intel Xeon E2136 3.3 GHz
Carte graphique	* NVIDIA 4Go	** NVIDIA 6Go ou plus

\* En théorie, le programme tel quel peut fonctionner sans carte graphique, à condition que la mémoire vive RAM soit suffisante et que les cartes ne soient pas trop grandes. Les temps de calcul peuvent cependant être multipliés par 5.

\*\* Pour le projet JADIS Paris, la carte utilisée était NVIDIA Quadro P2000 4 Go. Par conséquent, environ 25% des calculs de la segmentation devaient être effectués par le processeur. À noter que seules les cartes de la marque NVIDIA sont capables de faire fonctionner CUDA, et donc les bibliothèques actuelles de *deep learning*.

## Installer le programme (Niveau utilisateur 🙋)

1. Installez Anaconda (<https://docs.anaconda.com/anaconda/install/>)
2. Téléchargez le programme JADIS (<https://github.com/BnF-jadis/jadis>), décompressez-le et placez-le dans le dossier de votre choix, par exemple sur le Bureau
3. Téléchargez le réseau de neurones entraîné ([https://drive.google.com/file/d/13iRsEwFv9tTe68v5d\\_dXlEAJ9sn0qsb/view?usp=sharing](https://drive.google.com/file/d/13iRsEwFv9tTe68v5d_dXlEAJ9sn0qsb/view?usp=sharing)), décompressez-le et placez-le dans le dossier JADIS
4. Ouvrez une invite de commande. Sur Windows 7, 8 ou 10, vous pourrez utiliser l'application *Invite de commandes*, installée par défaut et accessible via la recherche. À défaut, vous pouvez utiliser par exemple *Anaconda prompt*. Sur Linux/Fedora ou sur Mac, utilisez *Terminal*.
5. Vous vous trouvez dans l'un des dossiers de votre ordinateur, en général le dossier source de votre compte. Le nom du dossier est indiqué à gauche de votre curseur, par exemple « *Users\Remi* » ou « *~Remi \$* ». Sur Unix (Mac, Linux), vous pouvez taper :

```
(base) MacBook: ~Remi$ ls
```

Pour lister les fichiers situés dans le répertoire où vous vous trouvez.

6. Naviguez jusqu'au dossier JADIS dans votre ordinateur. Par exemple, s'il se trouve sur votre Bureau, le chemin sera probablement *Desktop/JADIS* :

```
(base) MacBook: ~Remi$ cd Desktop/JADIS
```

7. Utilisez la fonction `setup.py` pour installer le programme :

```
(base) MacBook: ~Remi$ python setup.py
```

8. Lorsque l'invite de commande vous demande si vous souhaitez continuer, tapez *y* puis *retour*.
9. Si l'installation échoue, vérifiez votre connexion internet et ré-essayez une deuxième fois.

## Lancer le programme (Niveau utilisateur 🙌)

10. Si vous ne l'avez pas encore fait, installez le programme comme décrit dans le chapitre précédent.
11. Ouvrez une invite de commande. Sur Windows 7, 8 ou 10, vous pourrez utiliser l'application *Invite de commandes*, installée par défaut et accessible via la recherche. À défaut, vous pouvez utiliser par exemple *Anaconda prompt*. Sur Linux/Fedora ou sur Mac, utilisez *Terminal*.
12. Vous vous trouvez dans l'un des dossiers de votre ordinateur, en général le dossier source de votre compte. Le nom du dossier est indiqué à gauche de votre curseur, par exemple « *Users\Remi* » ou « *~Remi* \$ ». Sur Unix (Mac, Linux), vous pouvez taper :

```
(base) MacBook: ~Remi$ ls
```

Pour lister les fichiers situés dans le répertoire où vous vous trouvez.

13. Naviguez jusqu'au dossier JADIS dans votre ordinateur. Par exemple, s'il se trouve sur votre Bureau, le chemin sera probablement *Desktop/JADIS* :

```
(base) MacBook: ~Remi$ cd Desktop/JADIS
```

14. Activez l'environnement de JADIS. Cet environnement a normalement été créé lors de l'installation.

```
(base) MacBook: ~Remi$ conda activate jadis
```

15. Si cela a fonctionné, le mot (base), indiquant l'environnement en début de ligne, doit être remplacé par (jadis).

## Dossiers et fichiers



data

### Niveau utilisateur

Dossier de stockage du **corpus** (données et métadonnées). Les images de cartes (*maps*) ainsi que le tableau excel des métadonnées (*data.xlsx*) s'y trouvent. La géolocalisation et l'échelle sont aussi stockés dans ce tableau excel, en fin de calcul.



export

### Niveau utilisateur

Dossier vers lequel sont exportés tous les **résultats**, à l'exception de la géolocalisation et de l'échelle, qui sont stockés dans le dossier *data*.

*anchor* : ancre

*deformation* : déformation

*gephi* : réseau de similarité

*thinned* : réseau viaire filaire

*segmented* : cartes segmentées



model

### Niveau bidouilleur

Dossier dans lequel sont stockés les modèles (**réseaux de neurones** entraînés). Par défaut, le modèle utilisé est *jadis\_standard*. Le dossier *train* permet de stocker de nouvelles *images* d'entraînement, leurs *labels* (annotations) et le set de validation associé (*eval*).



save

### Niveau bidouilleur

Dossier dans lequel sont stockés les résultats intermédiaires du calcul, en particulier les **données mathématiques** et géométriques.

*HHOG* : histogrammes des HOG des images brutes du corpus, utilisés comme indicateurs de la similarité visuelle des images

IF : *invariant features*, ou caractéristiques (géométriques) invariantes des îlots des cartes segmentées

*match* : résultat brut (non-géolocalisé) de l'appariement de chaque carte, lorsque celui-ci est réussi.

*network\_matrix* : matrices des relations entre les images brutes de cartes, sur laquelle se base le réseau de similarité

*projection* : informations par rapport à la reprojection de l'ancre sur la surface quasi-sphérique de la Terre



settings

### Niveau bidouilleur

Dossier contenant les **paramètres**



utils

### Niveau programmeur

Dossier contenant les **fonctions python** utilisées par les scripts principaux

*match.py* : fonctions utilisées principalement par *geolocalizeMaps.py*

*networking.py* : fonctions utilisées par *makeSimilarityNetwork.py*

*segment.py* : fonctions utilisées principalement par *segmentMaps.py*



*thin.py* : fonctions utilisées par *makeThin.py*  
*utils.py* : fonctions utilitaires globales



workshop

### Niveau programmeur

Dossier de travail et de stockage **transitoire** des données intermédiaires pendant les calculs

## Utiliser les fonctions (Niveau utilisateur 🙋)

### Informations générales sur le fonctionnement du programme

Le programme JADIS est composé de plusieurs fonctions indépendantes ou pas les unes des autres. Il est ainsi possible de construire un processus de traitement des cartes sur mesure, en fonction des résultats que vous souhaitez obtenir. Le programme est conçu de sorte que tout ce que vous calculez soit sauvegardé. Ainsi, si pour une raison ou une autre, vous stoppez le programme, il reprendra automatiquement là où il s'était arrêté lorsque vous le relancerez. Par conséquent, il n'est pas possible de se tromper en répétant deux fois une étape, car le programme constatera rapidement qu'il n'y a plus rien à calculer. Si vous souhaitez néanmoins volontairement répéter une étape, il suffit de vous rendre dans le dossier où les résultats produits par la fonction sont sauvegardés, et de supprimer les éléments que vous souhaitez recalculer. Pour chaque fonction, les dossiers modifiés sont indiqués dans la description.

Pour des questions d'organisation, l'arborescence des fichiers est fixe et prédéterminée. Ne modifiez pas le nom d'un dossier à l'intérieur d'un projet, et ne déplacez pas les dossiers, sauf si vous les avez vous-même manuellement créés ou si vous n'avez plus l'intention d'utiliser JADIS pour le projet en cours.

#### 1. Créer un nouveau projet – createNewProject

Quelle que soit l'utilisation que vous souhaitez faire du programme, il vous faudra créer un nouveau projet JADIS. Cette fonction permet simplement de créer l'arborescence de fichiers.

##### Paramètre de la commande :

-- cityName : Le nom exact de la ville sur laquelle le projet porte, éventuellement entre guillemets si le nom est composé

```
python createNewProject.py --cityName "Paris"
```

##### Fichiers créés :

- 📁 data/projet
- 📁 save/projet
- 📁 export/projet
- 📁 settings/projet/settings.json

##### Fichier modifié :

- 📁 utils/projet.json

## Reprendre un projet existant – switchProject

JADIS travaille par défaut sur le dernier projet créé ou sur le dernier projet que vous avez chargé à l'aide de cette commande. Si vous souhaitez changer de projet ou revenir à un projet plus ancien, utilisez cette fonction.

### Paramètre de la commande :

-- cityName : Le nom de la ville utilisé pour créer le projet ou le nom des dossiers du projet dans l'arborescence. Si vous ne vous en souvenez plus exactement, lancez la commande et le programme vous fournira la liste des projets existants

```
python switchProject.py --cityName "Paris"
```

### Fichier modifié :

 utils/projet.json

## 2. Créer un corpus de cartes – createGallicaCorpus

Vous pouvez rassembler vous-même un corpus d'images de cartes et ses métadonnées, ou extraire automatiquement un corpus grâce à l'API Gallica. Ce programme crée automatiquement une requête SRU basée sur le nom de la ville du projet en cours. Il est possible de spécifier des bornes chronologiques dans les paramètres de la commande. Cette fonction crée automatiquement un fichier excel contenant les métadonnées du corpus et télécharge également toutes les images de cartes correspondantes.

Par défaut, les cartes sont aussi triées par rapport à l'échelle, lorsque cette donnée existe. Ce critère peut être modifié dans les paramètres du projet (lire le chapitre Paramètres, à la fin du document), tout comme le nom de ville utilisé pour la requête, lorsqu'il y a une polysémie (e.g. "Paris (Texas)").

```
python createGallicaCorpus.py --startYear 1800 --endYear1850
```

```
python createGallicaCorpus.py --SRU
"(dc.title%20all%20%22Paris%22%20%20and%20(dc.type%20all%20%22carte%22)%20and%20(provenance%20adj%20%22bnf.fr%22)&suggest=10"
```

### Paramètres de la commande :

-- startYear : (facultatif) Année de début du corpus  
 -- endYear : (facultatif) Année de fin du corpus  
 -- SRU : (facultatif) Ce champ permet, si vous le souhaitez, d'utiliser une requête SRU que vous avez créé vous-même. Dans ce cas, les bornes chronologiques ci-dessus ne sont pas prises en compte

### Fichiers créés ou modifiés :

 data/projet/data.xlsx  
 data/projet/maps

**Fichiers utilisés :**


 settings/projet/settings.json

## 2bis. Affiner le corpus – updateCorpus

Il est judicieux de contrôler manuellement le corpus constitué et de sélectionner les cartes qui vous intéressent, voire d'éliminer les cartes hors-sujet. Vous pouvez ensuite supprimer les entrées indésirables dans le fichier excel, ou dans le dossier contenant les images de cartes si vous préférez. Cette fonction permet de synchroniser ensuite l'un et l'autre.

```
python updateCorpus.py
```

**Fichiers modifiés :**

 data/projet/data.xlsx

 data/projet/maps


## 3. Interconnecter les cartes – createSimilarityNetwork


Si vous souhaitez géolocaliser les cartes, ou simplement si vous souhaitez analyser votre corpus, cette fonction permet de créer un graphe de similarité basé sur les caractéristiques visuelles et/ou sur les métadonnées. Il est possible de spécifier plus précisément les critères de construction du réseau dans les paramètres (voir chapitre à la fin du document).

Le graphe est basé sur une matrice des relations entre toutes les cartes du corpus, qui est sauvegardée pour les étapes ultérieures du projet. Un fichier graphe est également exporté et peut-être manipulé et visualisé grâce au logiciel libre Gephi. Dans ce dernier, chaque point correspond à une ligne dans le fichier excel des métadonnées.


```
python createSimilarityNetwork.py
```


**Fichiers modifiés :**

 export/projet/gephi

 save/projet/network\_matrix

**Fichiers utilisés :**

 data/projet/data.xlsx

 data/projet/maps

 settings/projet/settings.json

## 4. Créer un nouveau set d'entraînement – createTrainingSet

Si vous souhaitez entraîner vous-même un nouveau réseau neuronal, il est possible de constituer un nouveau set d'entraînement à partir de votre corpus. Ce programme va aléatoirement créer des patches d'images à partir du corpus.

Il vous faudra ensuite annoter manuellement les données (lire la documentation spécifique à la préparation des données d'entraînement dans la documentation technique vulgarisée). Les données annotées prennent place dans les dossiers labels, tandis que les données de référence prennent place dans les dossiers images. Ces dernières ne doivent pas être modifiées.

JADIS prend en charge au maximum 8 classes sémantiques différentes, qui doivent être annotées avec les couleurs pures ci-dessous, dans un logiciel de dessin (Photoshop, Gimp).

noir : [0, 0, 0] (HEX #000000)

blanc : [255, 255, 255] (HEX #FFFFFF)

rouge : [255, 0, 0] (HEX #FF0000)

vert : [0, 255, 0] (HEX #00FF00)

bleu : [0, 0, 255] (HEX #0000FF)

cyan : [0, 255, 255] (HEX #00FFFF)

magenta : [255, 0, 255] (HEX #FF00FF)

jaune : [255, 255, 0] (HEX #FFFF00)





Des patches de validation/d'évaluation sont également automatiquement créés. Ces derniers permettront au réseau neuronal d'apprendre correctement et doivent également être annotés.

```
python createTrainingSet.py --n 100
```

#### Paramètres de la commande :

-- n : (facultatif) Nombre de patches d'entraînement à créer (+ les patches de validation)

#### Fichiers modifiés :

-  model/train/projet/images
-  model/train/projet/labels
-  model/train/projet/eval/images
-  model/train/projet/eval/labels

#### Fichiers utilisés :

-  data/projet/maps
-  settings/projet/settings.json

### 4bis. Entraîner un nouveau réseau neuronal de segmentation – trainModel

L'entraînement d'un réseau neuronal nécessite un corpus annoté sur lequel le réseau peut apprendre. Une fois ce set d'entraînement constitué, vous pouvez entraîner votre propre réseau neuronal, afin de le rendre plus spécifique à votre corpus de base, ou simplement si vous souhaitez améliorer le modèle standard de JADIS. Cette étape est fortement conseillée si vous souhaitez calculer une segmentation sémantique de haute qualité. Les premiers résultats peuvent être obtenus avec une centaine d'exemples d'entraînement. Pour des résultats de segmentation qualitatifs, des sets de plus de 1000 exemples peuvent être nécessaires. Le modèle standard de JADIS vise avant tout à géolocaliser les

cartes, et s'appuie donc sur un corpus intermédiaire de 300 patches. Vous pouvez tout-à-fait partir de ce set d'entraînement et l'enrichir de nouvelles données d'apprentissage.

Lors du lancement, la fonction vérifie et nettoie également les labels annotés. Cette fonction est un wrapper de l'outil dhSegment, dont la documentation est également disponible en ligne.

```
python trainModel.py --continue
```






#### Paramètres de la commande :

-- continue : (facultatif) Permet de reprendre l'entraînement à partir d'un modèle pré-existant. Les conventions d'annotations doivent évidemment être identiques.

#### Fichiers modifiés :

 model

#### Fichiers utilisés :

 model/train/projet/images  
 model/train/projet/labels  
 model/train/projet/eval/images  
 model/train/projet/eval/labels  
 settings/projet/settings.json

### 4<sup>ter</sup>. Segmenter et sémantiser les cartes – segmentMaps




Cette fonction permettra d'extraire le réseau viaire ou d'exécuter une tâche de sémantisation spécifique si le réseau neuronal a été ré-entraîné. Le résultat de sortie est identique au format qui a été utilisé pour les labels d'entraînement du réseau.

```
python segmentMaps.py
```

#### Fichiers modifiés :

 export/projet/segmented

#### Fichiers utilisés :

 data/projet/maps  
 model  
 settings/projet/settings.json

### 5. Créer une ancre moderne de référence – createAnchor

Pour géolocaliser les cartes, il faut une carte de référence. Dans le programme JADIS, cette carte de référence est appelée ancre. Cette fonction permet de créer automatiquement une ancre (contemporaine), à partir des données d'OpenStreetMap. L'ancre est reprojétée sur la surface de la

Terre et les données de la projection sont stockées, pour pouvoir récupérer les latitudes et longitudes des objets géolocalisés plus tard.

```
python createAnchor.py
```

**Fichiers modifiés :**

- 📁 export/projet/anchor
- 📁 save/projet/projection

**Fichiers utilisés :**

- 📁 settings/projet/settings.json

## 6. Géolocaliser les cartes – geolocalizeMaps

Cette fonction permet de géolocaliser les cartes. Elle fonctionne en trois étapes. La première calcule les propriétés géométriques des îlots des cartes segmentées. La deuxième étape vise à réaligner un maximum de cartes sur l’ancrage, carte de référence actuelle. Les cartes qui ont pu être ancrées avec une précision maximale sont considérées comme des « ancres secondaires » et sont utilisées, lors de la troisième étape, pour réaligner les cartes qui n’ont pas pu être ancrées directement à l’ancrage actuelle.

Evidemment, vous devez spécifier au moins un format d’export lorsque vous lancez la commande.

```
python geolocalizeMaps.py --toPNG --toJSON --toSVG --toSHP
--geoloc
```

**Paramètre de la commande :**

- toPNG : pour un export image raster .png
- toSVG : pour un export image vectoriel .svg
- toJSON : pour un export web vectoriel .svg
- toSHP : pour un export vectoriel compatible avec QGIS
- geoloc : si ce paramètre est ajouté, l’export JSON pour le web sera géoréférencé

**Fichiers modifiés :**

- 📁 save/projet/IF
- 📁 save/projet/match/primary
- 📁 save/projet/match/secondary
- 📁 export/projet/deformation
- 📁 data/projet/data.xlsx

**Fichiers utilisés :**

- 📁 export/projet/anchor
- 📁 save/projet/projection
- 📁 settings/projet/settings.json

## 7. Déformer les cartes – applyDeformation

Cette fonction peut servir à déformer les cartes pour essayer de compenser les disparités avec la carte actuelle. Cette technique est encore expérimentale.

```
python applyDeformation.py
```

### Fichiers modifiés :

 export/projet/deformed

### Fichiers utilisés :

 data/project/maps

 export/project/deformation

## 8. Vectoriser les rues – makeThin

Cette fonction permet de vectoriser les rues pour récupérer un réseau viaire filaire, c'est-à-dire une représentation en bâtonnets du réseau viaire. L'export est possible dans plusieurs formats, selon l'usage souhaitez. Pour les GIS, il est possible de l'exporter en format shapefile (utilisé par QGIS notamment). Pour la visualisation, l'export se fera en png ou svg (voir ci-dessous). Pour le web, en format json.

Pour l'export au format svg, il est nécessaire d'installer Homebrew, ImageMagick et Potrace, dont l'installation n'est pas incluse dans JADIS. L'installation varie d'un ordinateur à l'autre, mais la suite de commande la plus fréquente est la suivante :

```
(base) MacBook: ~Remi$ /usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"  
(base) MacBook: ~Remi$ brew install imagemagick  
(base) MacBook: ~Remi$ brew install potrace
```

```
python makeThin.py
```

### Fichiers modifiés :

 export/projet/deformed

### Fichiers utilisés :

 data/project/maps

 export/project/deformation



## Concevoir une chaîne de traitement (Niveau utilisateur 🧑)

Comme les fonctions de JADIS sont conçues comme des blocs, il est possible de concevoir une chaîne de traitement sur mesure. Voici quelques exemples, utilisant les blocs principaux numérotés et décrits au chapitre précédent.

### Créer un corpus de cartes

1 => 2 => 2bis

### Etudier la similarité des cartes d'un corpus

1 => 2 => tri manuel => 2bis => 3

Si vous disposez d'un corpus de cartes annexes, vous pouvez aussi passer directement de la fonction 1 à la 3 en déposant vos cartes dans le dossier data/projet/maps. Un fichier excel associé peut être déposé dans data/projet/data.xlsx

### Etudier la similarité visuelle des cartes d'un corpus (Niveau bidouilleur 🧑🔧)

1 => dans settings/projet/settings.json, remplacez les lignes 7 à 57 par :

```
"networking": {
  "metadata": {
    "maps": {
      "filenames_column":
"filenames"
    },
    "columns": []
  },
  "visual_features": {
    "weight": 1,
    "HOG": {
      "angle_bins": 180,
      "reduce_size_factor": 4
    }
  }
}
```

=> 2 => tri manuel => 2bis => 3

### Segmenter le réseau viaire d'un corpus de cartes

1 => 2 => tri manuel => 2bis => 4bis => 4ter

Si vous disposez d'un corpus de cartes annexes, vous pouvez aussi passer directement de la fonction 1 à la 4bis en déposant vos cartes dans le dossier data/projet/maps.

### Entraîner un réseau neuronal à sémantiser un corpus de cartes

1 => 2 => tri manuel => 2bis => 4 => annotation du corpus d'entraînement (model/train/projet/labels et model/train/projet/eval/labels) =>

(optionnel **Niveau bidouilleur** 🛠️) Si vous n'avez pas utilisé les 3 couleurs de base de jadis (blanc, noir, cyan), modifiez les lignes 82-99 du fichier settings/projet/settings.json, comme expliqué dans le chapitre Paramètres.

Si vous disposez d'un corpus de cartes annexes, vous pouvez aussi passer directement de la fonction 1 à la 4 en déposant vos cartes dans le dossier data/projet/maps.

Pour utiliser le réseau neuronal entraîné, utilisez la fonction 4ter.

### Géolocaliser un corpus de cartes

1 => 2 => tri manuel => 2bis => 3 => 4ter => 5 => 6

Si vous disposez d'un corpus de cartes annexes, vous pouvez aussi passer directement de la fonction 1 à la 4 en déposant vos cartes dans le dossier data/projet/maps. Un fichier excel associé peut être déposé dans data/projet/data.xlsx

### Vectoriser un corpus de cartes

1 => 2 => tri manuel => 2bis => 4bis => 4ter => 8

Si vous disposez d'un corpus de cartes annexes, vous pouvez aussi passer directement de la fonction 1 à la 4bis en déposant vos cartes dans le dossier data/projet/maps.

### Etudier la déformation d'un corpus de cartes

1 => 2 => tri manuel => 2bis => 3 => 4ter => 5 => 6 (=> 7)

Si vous disposez d'un corpus de cartes annexes, vous pouvez aussi passer directement de la fonction 1 à la 4 en déposant vos cartes dans le dossier data/projet/maps. Un fichier excel associé peut être déposé dans data/projet/data.xlsx.

### Automatiser la chaîne de traitement (**Niveau programmeur** 🐍)

1. Créez un nouveau fichier python, par exemple en copiant-collant l'un des fichiers existants.
2. Nommez-le. Par exemple *process.py*.
3. Tapez `import os` sur la première ligne, puis complétez avec les fonctions de votre choix :

```
import os

os.system('python createSimilarityNetwork.py')
os.system('python segmentMaps.py')
os.system('python createAnchor.py')
os.system('python geolocalizeMaps.py --toJSON --geoloc')
```

4. Lancez le script (p.ex en tapant `python process.py` dans le terminal)

## Paramètres (Niveau bidouilleur )

Vous trouverez les paramètres sous `settings/projet/settings.json`<sup>1</sup>. Ils permettent de modifier le de l'algorithme pour les adapter à son problème.

### corpus

Cette famille de paramètres permet d'agir sur la constitution automatique du corpus avec *makeGallicaCorpus.py*, et de changer le nom de la ville du projet

**city\_name** : Nom de la Ville du projet en cours. Par exemple *Paris*.

**min\_scale** et **max\_scale** : Fourchette d'échelles acceptées lors de la constitution automatique du corpus. Les échelles sont indiquées en  $1/x$  (i.e.  $1: x$ ). Par conséquent, leur ordre est inversé. Pour sélectionner les cartes entre 1: 1000 et 1: 50 000, régler les paramètre comme ci-dessous.

**resolution\_ppi** : Résolution de la numérisation, en pixels par pouce. Cette donnée est utile, lorsque cette donnée manque, pour estimer l'échelle, après géolocalisation de la carte.

```
"city_name " : "",
"min_scale" : 1000,
"max_scale" : 50000,
"resolution_ppi" : 600
```

### networking > metadata

Cette famille de paramètres permet d'agir sur la constitution d'un graphe de similarité entre les images de cartes avec *makeSimilarityNetwork.py*.

**maps > filenames\_column** : Nom de la colonne correspondant au nom des fichiers images de *data/Projet/maps*, dans le fichier excel de *data/Projet/data.xlsx*.

**columns** : colonnes des métadonnées (de *data.xlsx*) à intégrer dans la création du graphe de similarité. Si vous ne souhaitez inclure aucune métadonnée, supprimez cette partie du tableau comme ceci :

```
"maps" : {
    "filenames_columns" : ...
},
"columns" : []
},
"visual_features" ...
```

<sup>1</sup> Les fichiers JSON sont des tableaux ordonnés qui structurent les informations hiérarchiquement. Ils sont lisibles et éditables avec n'importe quel outil de traitement de texte de base (par exemple TextEdit), ou à l'aide d'une application spécifique à la programmation (Visual Studio Code, Sublime Text). Ils sont aussi lisibles avec pratiquement tous les navigateurs (Firefox, Google Chrome).

columns > **name** : nom des colonnes des métadonnées à intégrer dans la création du graphe de similarité.

columns > **distance** : type de distance à utiliser, selon le type de données. Doit prendre l'une des valeurs suivantes :

- **numerical** : La distance entre les deux chiffres est calculée et divisée par la plus grande distance rencontrée dans cette colonne
- **categorical** : Les valeurs sont considérées comme des classes discontinues. Le score augmente uniquement si les deux cartes prennent exactement la même valeur
- **auto** : choix automatique entre numerical et categorical, selon si la donnée entrée est numérique ou pas.

columns > **weight** : poids, importance, coefficient à attribuer à cette colonne dans le calcul de la similarité

columns > **log** : Pris en compte seulement si la distance est numérique (numerical). Prend les valeurs **0** (= non) ou **1** (= oui). Si oui, la distance est calculée par rapport au logarithme en base 10. Cette colonne peut être utile si les valeurs s'étendent sur plusieurs ordres de grandeur, comme pour une échelle. Par exemple, si l'on veut que la distance entre l'échelle 1: 1000 et 1:10 000 soit identique à la distance entre l'échelle 1: 10 000 et 1: 100 000, on activera cette option.

```
{
  "name": "date",
  "distance": "auto",
  "weight": 2,
  "log": 0
},
```

## networking > visual\_features

**weight** : poids, importance, coefficient à attribuer aux caractéristiques visuelles dans le calcul de la similarité

**HOG** : *Histogram of oriented gradient* (histogramme du gradient orienté). Technologie qui permet de calculer les caractéristiques visuelles d'une image.

HOG > **angle\_bins** : Nombre de division

HOG > **reduce\_size\_factor** : Réduction de la taille de l'image. Il n'est pas utile de conserver l'image en pleine résolution, car les caractéristiques visuelles doivent être extraites de manière générale. Augmenter ce facteur permet aussi de réduire considérablement le temps de calcul.

```
"weight": 10,
"HOG": {
  "angle_bins": 180,
  "reduce_size_factor ": 4
}
```

## networking &gt; gephi

Ce champ permet de modifier les paramètres de l'export du réseau de similarité au format gephi.

**n\_neighbours** : nombre de liens minimal entre les nœuds du graphe (réseau) de similarité. Chaque nœud sera lié à ses n plus proches voisins. Augmenter ce nombre permettra d'obtenir un graphe plus interconnecté, donc potentiellement plus compact. Pour une visualisation optimale, conserver ce facteur entre 3 et 20, selon la taille du corpus.

```
"gephi": {
  "n_neighbours": 10
}
```

## cnn &gt; training\_params

**training\_params** : Paramètres spécifiques à l'entraînement du réseau de neurones. Il n'est utile de les modifier que si vous prévoyez de ré-entraîner le réseau à l'aide de la fonction *trainModel.py*

training\_params > **learning\_rate** : Taux d'apprentissage. Actuellement, l'entrée est une valeur en notation scientifique, vous pouvez cependant aussi utiliser des valeurs à point (e.g. *0.00005*, qui correspond à  $5e-05$  ( $5 \cdot 10^{-5}$ )). Augmenter cette valeur permet d'accélérer l'apprentissage. Cependant, cela augmente fortement le risque de divergence. La divergence représente le risque de « décrochage scolaire » du réseau de neurone, avec des performances catastrophiques à la fin. Si les performances sont catastrophiques, vous pouvez également réduire le taux d'apprentissage et augmenter le nombre d'époques, pour réduire le risque de divergence. Le taux d'apprentissage peut également être augmenté sans risque si la *batch\_size* (voir ci-dessous) est augmentée.

training\_params > **batch\_size** : Taille des lots, des groupes d'exemples sur lesquels le réseau apprend en même temps. Idéalement aussi haut que possible sur la mémoire de votre carte graphique. Si vous constatez que la mémoire de votre carte graphique n'est pas pleinement exploitée, augmentez la *batch\_size*. Le minimum est 1. Une taille élevée permet un apprentissage à la fois plus fiable, plus performant et plus rapide.

training\_params > **make\_patches** : *true* ou *false*. Le réseau neuronal découpe les patches en patches plus petits encore. Activer cette option est déconseillé car elle risque d'empêcher le réseau d'acquérir une vue d'ensemble.

training\_params > **training\_margin** : Si vos exigences sur le réseau ne sont pas maximales, ou si vous disposez d'un grand corpus d'entraînement et que vous augmentez nettement le nombre d'époques (*n\_epochs*), vous pouvez spécifier un seuil de performances à atteindre. Lorsque ce seuil est atteint, l'apprentissage prend fin. Lorsque cette valeur est fixée à 0, il n'y a pas de seuil et l'apprentissage se termine après que le nombre d'époques *n\_epochs* a été atteint.

training\_params > **n\_epochs** : Nombre de passages de l'information dans le réseau de neurones. Un nombre plus important d'époques est souhaitable (jusqu'à 100-200), mais nécessite un corpus

d'entraînement important. Conserver ce nombre plus bas (e.g vers 25-30) permet d'éviter le surapprentissage (*overfitting*). Le surapprentissage apparaît lorsque le réseau apprend par cœur les exemples d'entraînement mais se révèle incapable de généraliser ce qu'il a appris. Ce paramètre est l'un des plus importants. Il faut chercher à l'optimiser lorsque vous entraînez votre propre réseau.

training\_params > **data\_augmentation** : *true* ou *false*. Augmentation des données d'entraînement. Activer cette option permet de créer de multiples variantes des exemples d'entraînement, et donc potentiellement de compenser un corpus d'entraînement petit. Cela permet de rendre également le réseau robuste à plusieurs cas de figure (voir les paragraphes suivants)

```
"training_params": {
  "learning_rate": 5e-05,
  "batch_size": 1,
  "make_patches": false,
  "training_margin": 0,
  "n_epochs": 25,
  "data_augmentation": true,
  "data_augmentation_max_rotation": 6.28,
  "data_augmentation_max_scaling": 0.5,
  "data_augmentation_flip_lr": true,
  "data_augmentation_flip_ud": true,
  "data_augmentation_color": false,
  "evaluate_every_epoch": 5,
  "weights_labels": [1, 1, 1]
}
```

training\_params > **data\_augmentation\_max\_rotation** : Entre 0 et 6.28 ( $= 2\pi$ ). Rend le réseau robuste à diverses rotations de l'image.

training\_params > **data\_augmentation\_max\_scaling** : Entre 0 et 1. Rend le réseau robuste à des différences d'échelle.

training\_params > **data\_augmentation\_flip\_lr** : Fortement conseillé. Application d'un miroir gauche-droite.

training\_params > **data\_augmentation\_flip\_ud** : Fortement conseillé. Application d'un miroir haut-bas

training\_params > **data\_augmentation\_color** : Si votre corpus est composé de cartes utilisant des conventions couleur relativement uniformes (e.g parcs plutôt verts, cours d'eau plutôt bleus), il est conseillé de ne pas activer cette option.

training\_params > **evaluate\_every\_epoch** : Nombre d'époques entre chaque validation intermédiaire de l'apprentissage. Permet d'évaluer l'apprentissage en cours, et de le réorienter automatiquement. Cette étape est relativement longue, mais très utile à l'apprentissage lorsqu'elle est espacée de quelques époques.

training\_params > **weights\_label** : Poids des labels/classes dans l'apprentissage. Liste entre crochets de nombres, séparés par des virgules. Permet d'axer l'apprentissage sur une classe ou l'autre (par exemple sur les routes), si la qualité de la détection de certaines catégories est prioritaire.

cnn > gpu, patch\_size

training\_params > **gpu** : Si vous avez plusieurs cartes graphiques, le numéro de la carte de vous souhaitez utiliser pour cet entraînement. Vous pouvez le trouver dans les spécifications matérielles de votre ordinateur.

training\_params > **patch\_size** : taille des patches d'entraînement. Les patches doivent être carrés. **ATTENTION** : seule la constitution du corpus d'entraînement et l'entraînement proprement-dit est assuré pour une taille différente de 1000. Le reste du programme est entièrement prévu pour cette taille (toutes les tailles et formes de cartes sont en revanche prises en charge).

```
"gpu": "0",
"patch_size": 1000
```

## segment

Paramètres relatifs à la fonction *segmentMaps.py*

**classes** : Liste des couleurs (canaux rouge-vert-bleu (RVB), entre 0 et 255) des classes à segmenter sur la carte. Doit correspondre aux classes sur lesquelles le réseau a été entraîné. Si vous avez ré-entraîné un réseau, ce champ est pris en compte automatiquement lors de l'entraînement. Doit aussi correspondre exactement aux couleurs présentes sur les labels du corpus d'entraînement. Utilisez les couleurs pures suivantes (et les noms HEX correspondants sur votre logiciel d'annotation)

noir : [0, 0, 0] (HEX #000000)

blanc : [255, 255, 255] (HEX #FFFFFF)

rouge : [255, 0, 0] (HEX #FF0000)

vert : [0, 255, 0] (HEX #00FF00)

bleu : [0, 0, 255] (HEX #0000FF)

cyan : [0, 255, 255] (HEX #00FFFF)

magenta : [255, 0, 255] (HEX #FF00FF)

jaune : [255, 255, 0] (HEX #FFFF00)

Vous pouvez évidemment ajouter plus de couleurs (jusqu'à 8). N'oubliez pas les virgules !

**model** : Nom du modèle à utiliser (à placer dans le dossier *model*, ou spécifiez le chemin) pour la segmentation.

```
"segment": {
  "classes": [[0, 0, 0],
              [0, 255, 255],
              [255, 255, 255]],
  "model": "jadis_standard"
}
```

## anchor

Paramètres relatifs à la fonction *createAnchor.py*

**streetwidth\_coef** : Largeur des rues sur l'image de l'ancre moderne, extraite d'OpenStreetMap

**image\_maxsize** : Taille maximale de l'image de l'ancre moderne, extraite d'OpenStreetMap

**admin\_level** : Niveau administratif de l'entité OpenStreetMap à extraire. En France et dans la plupart des pays, le niveau 8 correspond au niveau de la ville. Le niveau 9 correspond à l'arrondissement municipal et le 7 à l'arrondissement (départemental). La liste complète est disponible [en ligne](#).

```
"anchor": {
  "streetwidth_coef": 4,
  "image_maxsize": 15000,
  "admin_level": 8
}
```

## matching

Paramètres relatifs à la fonction *geolocalizeMaps.py*. Deux variables (Lowes et RANSAC) peuvent être optimisées pour permettre un appariement optimal.

**Lowes** : Le ratio de Lowes correspond à un seuil de spécificité de chaque appariement. Le but est de ne conserver que les appariements qui sont suffisamment uniques, et donc fiables, car ils concernent des îlots à la forme atypique et reconnaissable. Un ratio de Lowe à 0.8, signifie que l'appariement doit dépasser tout autre voisin concurrent d'au moins 20%, en termes de spécificité. Différentes valeurs, comprises entre **min** et **max** sont testées pour chaque appariement et le meilleur résultat est conservé. **bins** spécifie le nombre de valeurs à tester dans l'intervalle.

```
"Lowes": {
  "min": 0.75,
  "max": 1,
  "bins": 20
},
```



**RANSAC** : *Random Sample Consensus*, est un algorithme qui ne permet de conserver, dans la liste de transformations possibles pour projeter la carte A sur la carte B, que le plus grand ensemble cohérent d'appariements, avec une certaine marge de tolérance, qui correspond à la déformation maximale potentielle de la carte (en pixels). C'est cette marge de tolérance de l'algorithme vise à optimiser, et qu'il est possible de modifier dans ce champ. La liste entre crochet des valeurs séparées par des virgules correspond aux différentes marges de tolérance à tester.

```
"RANSAC": [150, 175, 200, 225, 250],
```

**anchoring\_score**: Score maximum d'appariement. Au-delà de ce score, l'appariement est considéré comme un échec. Pour augmenter la tolérance à l'erreur, augmentez cet indice. À l'inverse réduisez-le si vous ne souhaitez conserver que les meilleurs résultats. Les valeurs pertinentes se situent entre 0.5 et 2.0

**secondary\_anchor\_score**: Score maximum d'appariement d'une carte avec l'ancre pour que cette carte soit considérée comme une ancre secondaire. S'il est fixé à 0, il n'y aura pas d'ancres secondaires.

```
"anchoring_score": 1.0,  
"anchoring_score": 0.6
```

## Contact

Pour signaler un bug, résoudre un problème ou simplement poser une question, n'hésitez pas à me contacter :

Rémi Petitpierre

+41 79 913 12 55

[remi.petitpierre@alumni.epfl.ch](mailto:remi.petitpierre@alumni.epfl.ch)

Vous pouvez aussi poser votre question sur le Github du projet JADIS