Rapport individuel - Web Client

Github du projet: https://github.com/si5-web-project/covid19-visualization/
Identifiant Github: **yursilv** (yury.silvestrov-henocq@etu.unice.fr)

1. Tâches effectuées

Dans ce projet je me suis principalement occupé du traitement des données statistiques côté client et de leur affichage sur une carte des régions de France. Les tâches indiquées plus bas sont suivies par un pourcentage du temps, 100% correspondant au temps total passé sur le projet.

1.1. Carte des régions de France (20% du temps)

Au début du projet nous avons décidé d'utiliser les données de santé mentale en France (lien) pendant la période du Covid-19 pour montrer son évolution et sa corrélation possible avec le nombre de nouveaux cas. L'idée était de mettre en avant un affichage dynamique de données sur une carte. Il fallait donc trouver une solution pour la carte sachant que les fonctionnalités dont nous avions besoin étaient:

- les popovers (fenêtre qui apparaît au survol d'un élément de la carte)
- la possibilité de positionner des éléments graphiques sur la carte
- la possibilité d'ajouter plusieurs couches de couleurs

Malheureusement les bibliothèques trouvées étaient soit trop pauvres en fonctionnalités (exemple), soit payantes (exemple), soit utilisaient des vieilles technologies comme jQuery (exemple) qu'on ne voulait pas intégrer dans notre projet. En plus nos données étaient en fonction des régions et non pas en fonction des départements ce qui n'était pas le cas pour la plupart des solutions de carte trouvées.

Finalement j'ai décidé de mettre en place ma propre solution. J'ai d'abord récupéré un fichier svg des régions de France (lien). Même si j'aurais pu parser ce fichier de manière automatique, avec seulement 13 éléments j'ai considéré que c'était plus rapide de le faire à la main. Finalement j'ai obtenu un tableau avec les données de toutes les régions: code, nom de la région et son "path", une donnée qui permettra d'afficher la région au sein d'un élément **<svg>** sur la page.

Une fois les données récupérées j'ai mis en place un composant **Region** ayant pour responsabilité d'intégrer toute la logique d'affichage concernant une région.

1.2. Visualisation du nombre de cas (10% du temps)

L'idée pour représenter les nouveaux cas par jour était d'utiliser l'attribut **fill** de **<path>** pour remplir la région avec une couleur. Pour faire varier la couleur j'ai utilisé l'attribut **opacity** en calculant l'opacité en fonction du nombre de cas grâce à une simple fonction linéaire.

La difficulté principale consistait à trouver une façon de représenter à la fois la différence du nombre de cas entre les régions à une date précise et l'évolution globale du nombre de cas en fonction du temps. La solution trouvée était d'utiliser 2 couches de couleurs transparentes: une couche rouge pour représenter la différence entre les régions à une date précise (le nombre de cas min et max à cette date jouant le rôle de coefficients de la fonction linéaire) et une couche noire pour représenter l'évolution globale (le nombre de cas min et max sur toutes les dates jouant le rôle de coefficients de la fonction linéaire).

1.3. Visualisation de la santé mentale (40% du temps)

Notre objectif initial était de visualiser le nombre de cas et la santé mentale en même temps. Comme la représentation par couleurs était déjà réservée au nombre de cas, il fallait trouver une autre solution pour la santé mentale. Au début j'ai essayé d'utiliser le hachurage, mais les couleurs se mélangeaient trop et le rendu n'était pas satisfaisant. Finalement j'ai pensé à une visualisation plus parlante et originale qui consistait à représenter la santé mentale dans la région par un smiley souriant ou triste. Je suis parti d'un smiley au format svg.

La difficulté principale consistait à bien positionner le smiley dans la région. Les balises **<path>** que j'ai utilisées pour afficher les régions ne permettaient pas de facilement calculer le milieu de l'élément et d'y placer un objet en superposition. Finalement j'ai repris la carte svg initiale que j'avais utilisée pour extraire les données concernant les régions. J'ai placé les smileys sur cette carte aux endroits souhaités et j'ai enregistré l'image carte+smileys au format svg. Grâce à ce fichier j'ai pu récupérer le code svg des smiley placés sur les région et je l'ai ajouté ensuite au tableau des régions créé précédemment (avec code, nom, path).

La seule façon que j'ai trouvée pour injecter ce code dans la page était d'utiliser une nouvelle balise **<svg>** placée à l'intérieur de **<path>** représentant la région. J'ai utilisé l'attribut **dangerouslySetInnerHTML** proposé par React pour injecter le code du smiley dans le **<svg>**.

Finalement il fallait aussi trouver un moyen pour faire varier le sourire du smiley en fonction des données. Pour ce faire j'ai mis en place une fonction qui parse le code svg du smiley pour trouver les points dont la variation des coordonnées fait monter ou descendre le sourire et augmente ou descend ces points toujours à l'aide d'une fonction linéaire.

La solution alternative pourrait être de créer un ensemble de smileys au format png chacun avec un sourire légèrement différent et de charger la bonne image en fonction de la santé mentale. Cette solution a plusieurs inconvénients. Premièrement ce serait une représentation discrète limitée par le nombre d'images, contrairement à un svg où il suffit de calculer les coordonnées de certains points grâce à une fonction linéaire donnant un rendu continu. Deuxièmement, cela aurait un impact négatif sur les performances, car il faudrait render de nouvelles images à chaque changement de date (qui peut être assez rapide).

1.4. Slider de date (25% du temps)

Le composant **Region** mis en place pour la visualisation de tout ce qui concerne une région a besoin des données de la santé mentale et du nombre de cas à une date précise. Pour lui fournir ces données et mettre en avant un affichage dynamique j'ai décidé de créer un slider permettant de faire varier la date rapidement. Je me suis basé sur le slider fourni par la bibliothèque material-ui (lien). La difficulté principale ici venait de l'intégration de la notion de date dans ce slider qui fournit en sortie un nombre entre un nombre minimum et maximum (par exemple 0 et 100). Il fallait donc déduire une date en fonction de ce nombre. Ma façon de procéder était de:

- fixer le minimum du slider à 0 et le maximum à la différence en jours entre la date de début des données et la date de fin
- récupérer la valeur fournie par le slider
- considérant cette valeur comme nombre de jours et les ajouter à la date de début des données grâce à la bibliothèque moment.js (lien)
- déduire une nouvelle date qui est la date recherchée

Une autre difficulté consistait à fournir les données de santé mentale par jour, parce que les données fournies présentaient des enregistrements ponctuels avec des écarts de 3-4 semaines entre eux. Pour en déduire les données par jour, j'ai encore une fois utilisé des fonctions linéaires.

1.5. Détection de la région de l'utilisateur (5% du temps)

Pour détecter la région de l'utilisateur j'ai d'abord utilisé l'api du navigateur permettant de récupérer la longitude et la latitude de l'utilisateur. Ensuite en s'appuyant sur ces données j'ai utilisé "api-adresse.data.gouv.fr/reverse/?lon=...&lat=..." pour déterminer le département et ensuite "https://geo.api.gouv.fr/departements/[departementCode]" pour récupérer les données précises concernant ce département, y compris le code de la région associée. Une fois ce code récupéré, je pouvais faire clignoter la région de l'utilisateur que ce soit sur la carte ou en mode d'affichage liste.

2. Gestion des versions

Voici le protocole que nous avons défini au début du projet pour la gestion des versions (A et B - membres de l'équipe) :

1. A crée une issue et précise dans l'issue ce qu'il va faire
2. A s'assigne sur l'issue et ajoute les bons labels (feature, bug ou task)
3. A crée une branche pour travailler sur l'issue
4. Pendant le développement, A merge régulièrement **main** dans sa branche pour gérer les conflits au fur et à la mesure
5. Une fois le développement terminé, A crée une Pull Request, ajoute une description, un screenshot, assigne B en revue et écrit dans le channel Slack "Pull Request prête pour revue par B: *lien vers la Pull Request*"
6. B fait la revue de code (et la revue fonctionnelle si nécessaire)
7. S'il n'y a pas de retours à faire, B merge la branche dans **main** et ferme l'issue, sinon il demande à A de faire des corrections
8. A corrige et redemande une revue à B etc. jusqu'à ce que le merge ne soit effectué

Tout d'abord cette façon de procéder nous a permis de détecter et corriger certains problèmes liés à la qualité du code (style, optimisation, utilisation de bonnes pratiques) et d'éliminer les bugs évidents. Un autre avantage important des revues vient du fait qu'on voit ce que les autres membres de l'équipe ont fait et avec cette vision plus globale on comprend mieux où on en est et comment intégrer notre travail.

3. Code

Calcul de date élégant

```
const dateChanged = (event, newValue) => {
    if (newValue !== currentSliderValue) {
        setCurrentSliderValue(newValue);

        const newValueDate = moment(firstDatasetDate).add(newValue, 'days').toDate();
        setSelectedDate(newValueDate);

        onDateChange(newValueDate);
    }
}
```

Cette fonction est appelée lorsque l'utilisateur change la valeur du slider de date. Je suis content d'avoir trouvé une solution assez simple pour transformer le nombre fourni par le slider en date. Comme expliqué dans la partie "Slider de date" j'ai utilisé moment.js pour ajouter des jours à la première date des données et en déduire la date choisie par l'utilisateur.

Passage de props plutôt lourd

```
const Map = ({
                userRegionId,
                newCasesNow,
                minNewCasesNow,
                maxNewCasesNow,
                minNewCasesEver,
                maxNewCasesEver,
                mentalHealthNow,
                minMentalHealthEver,
                maxMentalHealthEver
            }) => {
    return (
        <svg width="578px" height="544px" viewBox="0 0 578 544">
            <g id="carte" transform="translate(12.000000, 12.000000)">
                {
                    REGIONS.map(region =>
                        <Region
                            id={region.id}
                            isUserRegion={userRegionId === region.id}
                            key={region.id}
                            path={region.path}
                            smiley={region.smiley}
                            name={region.name}
                            regionNewCases={newCasesNow.find(r => region.id + "" === r.regionId)?.newCases}
                            minNewCasesNow={minNewCasesNow}
                            maxNewCasesNow={maxNewCasesNow}
                            minNewCasesEver={minNewCasesEver}
                            maxNewCasesEver={maxNewCasesEver}
                            mentalHealthNow={mentalHealthNow.find(r => region.id + "" === r.regionId)}
                            minMentalHealthEver={minMentalHealthEver}
                            maxMentalHealthEver={maxMentalHealthEver}>
                        </Region>)
```

Dans le composant **Region** j'avais besoin d'un grand nombre de données différentes pour faire des calculs graphiques. Ces données venaient du composant **RegionsStats** en passant par le composant **Map**. Je pense que le passage de ces props qui se sont accumulées pourrait être optimisé au moins en regroupant certaines d'entre elles dans un seul objet.