

# Projet chef d'oeuvre

## *Cycling Travel Time*


Vincent Lagache

# Sommaire

<b>Cycling Travel Time</b>	<b>1</b>
<b>Sommaire</b>	<b>2</b>
<b>Introduction</b>	<b>3</b>
Problématique	4
Analyse des besoins	5
<b>Gestion de projet</b>	<b>5</b>
Acteurs du projet	6
Méthode	6
Etapas	7
<b>Intelligence artificielle</b>	<b>8</b>
Dataset	8
Description	8
Collecte	9
Exploration des données	10
Nettoyage des données	12
Stockage et importation en base de données	13
Méthodes	15
Algorithme naïf	15
Features	16
Split Train/Test	17
Transformation du label	18
Modèles	19
Segmentation de la donnée à prédire	20
Axes d'amélioration	22
<b>Application</b>	<b>23</b>
Description	23
Base de données	24
Choix du type de base de données	24
Schéma de la base de données	25
Utilisation	25
Mise en place	27
Backup	28
Performance	30
Backend	30
Description	30
Intégration de l'IA	31
Frontend	33
Description	33
Screenshots	33
<b>Axes d'amélioration</b>	<b>35</b>
<b>Conclusion</b>	<b>36</b>

# Introduction

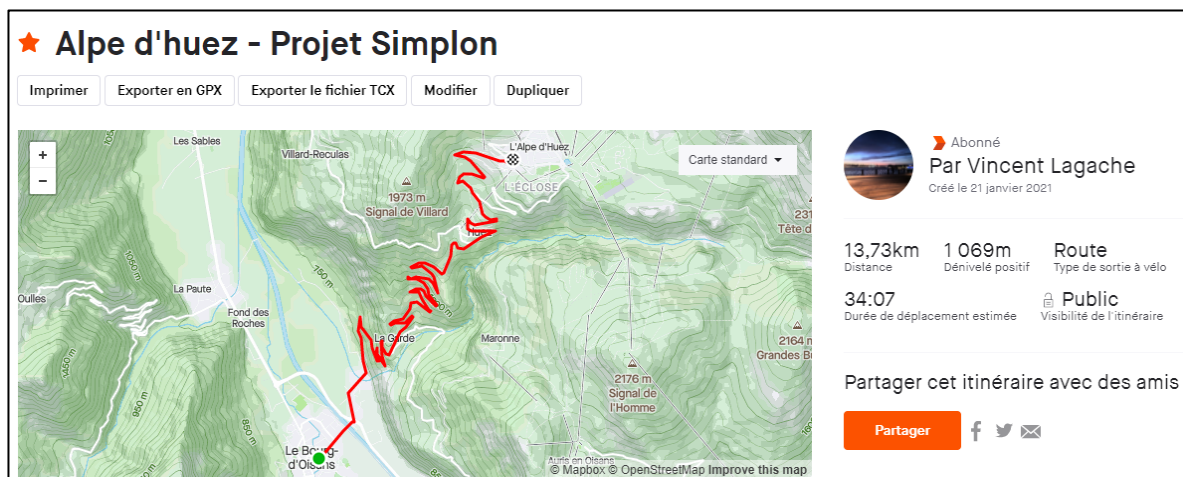
Étant cycliste amateur, j'utilise l'application Strava à chacune de mes sorties. Pour rappel Strava est un site internet et une application mobile utilisée pour enregistrer des activités sportives via GPS. Le cyclisme et la course à pied concentrent la majorité des activités du site<sup>1</sup>. L'application enregistre de nombreuses informations sur l'activité réalisée.

44,04 km	1:36:28	175 m	319
<a href="#">Distance (?)</a>	Durée de déplacement	<a href="#">Dénivelé (?)</a>	Mesure d'effort : historique
197 W	1 017 kJ	103	80%
Puissance moy. pondérée	Effort total	Charge d'entraînement	Intensité
	Moy.	Max.	Réduire
Vitesse	27,4 km/h	42,1 km/h	
Fréquence cardiaque	174 bpm	195 bpm	
Cadence	79	171	
Puissance	176 W	837 W	
Calories	1 005		
Température	12 °C		
Temps écoulé	1:36:28		
 Quelques nuages		Ressenti	18 °C
Température	18 °C	Vitesse du vent	18,9 km/h
Humidité	58%	Direction du vent	ESE
Wahoo ELEMNT		Vélo: Giant	

Informations fournies par Strava après une activité

Strava propose aussi un accès Premium à son service qui permet de bénéficier de plus de fonctionnalités. L'une d'entre elle permet à l'utilisateur de définir un parcours qu'il souhaite réaliser sur une carte et par la suite de l'exporter ou le partager. Un des principaux intérêts de cette fonctionnalité est de préparer en amont une future sortie dans un lieu que l'on ne connaît pas forcément, pour ensuite importer ce fichier dans un compteur qui vous indiquera la route à suivre une fois sur votre vélo.

1 <https://fr.wikipedia.org/wiki/Strava>



*Parcours crée grâce à la fonctionnalité de création d'itinéraire de Strava*

Lors de la création d'un tracé sur le site web comme on peut le voir sur la capture d'écran ci-dessus, Strava vous indique la distance totale, le dénivelé positif et négatif mais aussi ce qui nous intéresse plus dans le cadre de ce projet, la durée de déplacement estimée. Cette information pourrait de prime abord se révéler intéressante pour l'utilisateur (par exemple pour connaître avant de réaliser son activité, le temps que cela prendra approximativement, s'il ne dispose pas de beaucoup de temps ou qu'il fait du sport dans une région inconnue) néanmoins on s'aperçoit rapidement du problème de cette estimation.

## Problématique

La création du tracé qui nous sert d'exemple représente approximativement l'ascension de la mythique montée de l'Alpe D'Huez qui est un des passages les plus connus du Tour de France dont le dénivelé est de 1090 mètres pour un peu moins de 14 kilomètres. L'Italien Marco Pantani (pour rappel il est l'un des meilleurs grimpeurs de l'histoire du cyclisme, sanctionné pour dopage en 1999 et mort d'un overdose en 2004) détient depuis 1995 le record de cette ascension en 36min40. Or comme on peut le voir, Strava me prédit une durée de déplacement estimée sur le même parcours d'un peu plus de 34 minutes. Strava semble donc se tromper très largement dans l'estimation fournie à l'utilisateur. Sans connaître leur méthode de calcul on peut "deviner" rapidement un des problèmes en s'apercevant que l'application nous fournit le même temps pour le même parcours dans l'autre sens (dans le sens de la descente) : cela ne tient pas compte du dénivelé, l'estimation fournie (14 km en 34 min) représente une vitesse moyenne d'un peu moins de 25 km/heure qui est la vitesse moyenne qu'un cycliste relativement peu entraîné atteindrait sur un parcours plat.

L'idée de ce projet serait de développer une application utilisant l'IA et le Machine Learning afin d'obtenir une estimation plus réaliste de la durée de déplacement lors d'une future activité en tenant compte de la topographie de la route mais aussi des anciennes sorties sportives (et donc des anciennes performances sur des tracés similaires). On peut aussi imaginer dans le futur après confirmation de la réussite du concept par la réalisation d'un MVP qui ferait des prédictions plus réalistes que l'existant, agréger d'autres données dans l'entraînement d'un modèle (estimation de la forme actuelle à travers divers indicateurs, poids, matériel), le sport étant un domaine très riche en data.

## Analyse des besoins

L'application est destinée à un cycliste, utilisateur de Strava, et ayant idéalement un certain nombre d'activités enregistrées sur le site pour servir d'entraînement au modèle de Machine Learning. L'utilisateur pourra enregistrer par le biais de l'application toutes ses activités sur une base de données afin d'entraîner un modèle sur ses données sportives. Par la suite il fournira à l'application un itinéraire qu'il souhaite effectuer et se verra prédire une durée estimée de déplacement. L'utilisation de l'application se fera pour le moment uniquement dans un contexte local, car il semble plus judicieux de stocker des activités d'un utilisateur sur son propre ordinateur pour lui garantir la confidentialité de ses données. En effet même si les données sportives ne comprennent pas d'information très sensibles, il est par exemple possible de connaître l'adresse du domicile d'une personne assez facilement si toutes les activités partent du même point géographique. Il serait souhaitable d'entraîner de nouveaux modèles au fur et à mesure de l'enregistrement de nouvelles activités pour coller au mieux à la réalité des performances de l'utilisateur.

## Gestion de projet

**Note :** Cette partie à pour but d'illustrer la gestion de projet s'il avait eu lieu dans un contexte professionnel et ne colle pas exactement à la réalisation réelle du projet. Il serait en effet étrange dans le cadre d'un contrat avec Strava que le produit fini, soumis au client, soit une application tournant dans un Docker destiné à un utilisateur unique et pas une amélioration de leur application...

## Acteurs du projet

Le client sur ce projet est Strava qui souhaite améliorer sa fonctionnalité de prédiction de durée de déplacement à l'aide du Machine Learning. L'équipe est composée d'un chef de projet et de deux développeurs (un tech lead, et un développeur moins expérimenté).

# Méthode

La méthode Kanban a été utilisée pour le volet gestion de ce projet. Il est possible de créer un tableau de type Kanban sur Github. Chaque fonctionnalité à ajouter ou problème à régler est répertoriée et labellisée en fonction de sa nature. Ces tâches à réaliser sont séparées en colonnes.

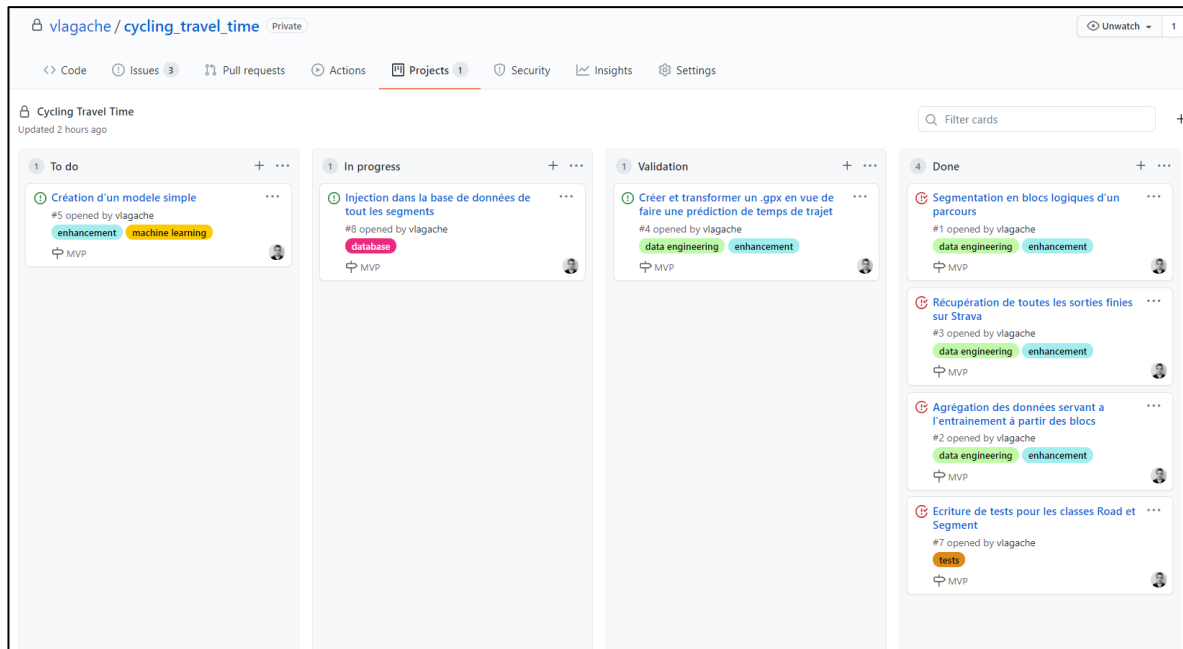


Tableau "Kanban" sur Github

- To do : tâches à réaliser lors du sprint en cours
- In Progress : tâches qu'un des développeurs est en train de réaliser. On crée une branche à partir de la dernière version du code stable sur laquelle on va développer cette nouvelle fonctionnalité.
- Validation : lorsqu'un développeur a fini la tâche en cours, il la soumet à validation au "Tech Lead" du projet (un développeur expérimenté) qui vérifiera le bon fonctionnement du code produit. On soumet aussi une merge request.
- Done : le travail réalisé par le développeur est validé par le Tech Lead, on fusionne alors la branche dans la branche principale.

Ce tableau est la principale source de communication entre les équipes. Chaque jour a lieu un "daily", courte réunion, où ce tableau est mis à jour. Chacun décrit le travail qu'il a pu

effectuer depuis le dernier daily sur la tâche qui lui est assignée. C'est aussi l'occasion de se voir octroyer une nouvelle tâche à réaliser par le Tech Lead ou se l'octroyer soit même en fonction de son expérience.

La fin d'un sprint amène à une présentation du travail réalisé par l'équipe de développement. C'est l'occasion de vérifier auprès des utilisateurs finaux et des experts métiers que ce qui a été développé correspond bien à ce qui a été demandé. C'est aussi l'occasion pour eux de proposer des modifications qui pourront être intégrés (ou pas) aux prochains sprints. Il s'agit aussi à la fin d'un sprint de faire une rétrospective au sein de l'équipe de développement pour que chacun puisse exprimer d'éventuelles remarques sur tous les volets de la réalisation du projet (choix technique, rythme de travail, communication avec le client ou au sein de l'équipe etc.). Cette rétrospective vise à améliorer la productivité et l'efficacité des prochains sprints en corrigeant ce qui a pu éventuellement ralentir l'équipe ou au contraire en renforçant ce qui s'est bien passé.

Le but souhaité lors de la fin d'un sprint est que chaque tâche qui était présente au début dans la colonne To Do soit passée à Done.

## Etapes

- Analyse du besoin client, à savoir proposer une prédiction d'un temps de trajet à l'aide du Machine Learning, en prenant en compte le dénivelé de la route et l'état de forme de l'athlète.
- Analyse de la faisabilité des points cruciaux du projet comme la récupération des données, la segmentation d'une route à prédire en morceaux logiques ainsi que la manière pour l'utilisateur de définir une route sur laquelle il veut recevoir une prédiction.
- Développement d'une application permettant la récupération des données et leur persistance en base.
- Création d'une interface graphique permettant à l'utilisateur de réaliser les différentes actions nécessaires au fonctionnement de l'application (récupération des données, entraînement d'un modèle, prédictif) de manière plus intuitive.
- Développement de la partie Machine Learning (analyse des données, nettoyage, création de modèles et intégration de ce travail dans l'application de manière à pouvoir reproduire le travail de data science par action de l'utilisateur)

Le projet a été réalisé de fin Novembre 2020 à fin Mars 2021 soit approximativement 4 mois. La durée des sprints était d'une semaine.



# Intelligence artificielle

L'IA développée dans le cadre de ce projet a pour but de prédire un temps de trajet en secondes sur un trajet. Le but initial de ce projet étant de tenter d'améliorer l'outil dont dispose Strava, le trajet à prédire sera découpé en segments logiques de montée, plat ou descente afin de prendre en compte le dénivelé moyen de chacun des segments. L'IA prédira un temps pour chacun de ces segments et en additionnant l'ensemble de ces prédictions, il sera possible de retourner un temps total de trajet à l'utilisateur ramené pour plus de lisibilité en heures/minutes/secondes.

## Dataset

### Description

A l'aide de l'API Strava, il est possible de récupérer toutes les sorties réalisées par un athlète avec son autorisation sous la forme d'un JSON qui résume chaque activité (distance, vitesse, puissance, coordonnées géographiques etc...). Chaque JSON comprend aussi tous les segments de l'activité. Les segments désignent des portions spécifiques d'un itinéraire. Il est possible pour chaque utilisateur de créer ses propres segments au moyen de l'outil prévu à cet effet sur le site web de Strava. Dès que l'on termine un segment (créé par nous ou un autre utilisateur), le temps réalisé est enregistré afin que l'on puisse comparer à nos performances passées à ou à celle de nos amis et d'autres sportifs.

		Arrivée course de Villenave Chambéry 0,22km 13m 5,6 %	30s	27,1km/h	242W	173bpm
---	---	--	-----	----------	------	--------

*Segment Strava*

Même si la somme de tous les segments d'une activité ne représente pas la totalité du parcours, cela devrait constituer un dataset assez varié pour illustrer la diversité des performances passées d'un athlète (quelle performance dans une portion de route à x % de dénivelée, de y km par exemple.). En conséquence le dataset utilisé dans ce projet sera constitué de l'ensemble des informations sur l'ensemble des segments que j'ai personnellement parcouru depuis que j'utilise Strava (2027 segments à l'heure où ces lignes sont écrites)

```
"id": 2787395491216040000,
```



```

"resource_state": 2,
"name": "Epic KOM",
"activity": {
  "id": 4658163829,
  "resource_state": 1
},
"athlete": {
  "id": 10944546,
  "resource_state": 1
},
"elapsed_time": 2000,
"moving_time": 2000,
"start_date": "2021-01-21T17:14:49Z",
"start_date_local": "2021-01-21T18:14:49Z",
"distance": 9410.5,
"start_index": 555,
"end_index": 2555,
"average_cadence": 73.5,
"device_watts": true,
"average_watts": 193.9,
"average_heartrate": 170.8,
"max_heartrate": 193,
...


```

*Une partie des informations que contient un segment Strava*

## Collecte

Les données ont été collectés comme évoqué précédemment avec l'aide de l'API Strava. Strava fournit aux développeurs l'accès à une API permettant de récupérer toutes les informations du compte d'un utilisateur avec son autorisation. Cet accès est basé sur le protocole OAuth qui est protocole libre qui permet d'autoriser une application à utiliser l'API sécurisée d'un autre site web pour le compte d'un utilisateur, c'est un protocole de délégation d'autorisation

Je reviendrais plus en détail dans la partie Application sur comment ceci est implémenté dans le code, mais brièvement en voici le fonctionnement général. L'utilisateur démarre l'application, il est redirigé vers la page OAuth du site Strava ou il va devoir entrer les identifiants de son compte. Il lui sera notifié que l'application qu'il utilise souhaite pouvoir accéder à son compte et dans quelle mesure (accès uniquement aux informations publiques ou à toutes les informations du compte par exemple).



Autoriser Cycling Travel Time à se  
connecter à Strava

Prédiction d'un temps de trajet

<http://www.vincentlagache.com/>

---

**Cycling Travel Time pourra :**

- ☐ Consulter les données de votre profil public (obligatoire)
- ☒ Consulter l'intégralité de votre profil Strava
- ☒ Consulter les données de vos activités privées

**Autoriser**

**Annuler**

*Page OAuth du site Strava*

L'utilisateur confirme qu'il est d'accord pour nous autoriser à accéder à ses données et il est alors redirigé vers notre URL, Strava nous fournit à ce moment là un code d'autorisation pour sceller cet accord. Suite à ça notre application échange ce code contre deux tokens, un d'accès et un de rafraîchissement. En effet le token d'accès n'est pas valable indéfiniment et il faut le renouveler toutes les 6 heures à l'aide du "refresh\_token". A partir de là, notre application peut récupérer toutes les informations du compte de l'utilisateur dont elle a besoin pour fonctionner.

## Exploration des données

L'ensemble du dataset comporte à l'heure où ces lignes sont écrites 215 activités et 2027 segments. Ce nombre est susceptible d'évoluer puisque notre application est capable de mettre à jour la base de données à chaque fois qu'une nouvelle activité est réalisée.

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2027 entries, 0 to 2026
Data columns (total 17 columns):
#   Column                Non-Null Count  Dtype
---  -
0   id                     2027 non-null   int64
1   activity_id            2027 non-null   int64
2   athlete_id            2027 non-null   int64
3   name                   2027 non-null   object
4   type                   2027 non-null   object
5   elapsed_time           2027 non-null   int64
6   moving_time            2027 non-null   int64
7   distance               2027 non-null   float64
8   average_cadence        1109 non-null   float64
9   average_watts          2027 non-null   float64
10  average_grade           2027 non-null   float64
11  maximum_grade           2027 non-null   float64
12  climb_category          2027 non-null   int64
13  average_heart_rate      1186 non-null   float64
14  max_heart_rate          1186 non-null   float64
15  start_time              2027 non-null   object
16  start_date              2027 non-null   object
dtypes: float64(7), int64(6), object(4)
memory usage: 269.3+ KB

```

*Description du dataset*

La description du dataset permet dans un premier temps de se rendre compte que certains segments ne comportent pas des informations qui auraient pu nous intéresser afin de traduire un état de forme comme par exemple la cadence moyenne de pédalage ou les battements cardiaques moyens sur un segment. La raison étant, et c'est un des principaux problèmes de ce projet que toutes les données n'ont pas été enregistrées à partir du même appareil. Si la capture de l'activité se fait par le biais d'un téléphone mobile, Strava estimera la puissance moyenne développée mais ne pourra logiquement pas deviner à quelle cadence vous avez pédalé ou quelles étaient vos données cardiaques. Lorsque j'ai débuté ma pratique du vélo, j'utilisais un téléphone portable, mais dernièrement j'utilise un compteur couplé à une ceinture cardiaque ainsi qu'un capteur de puissance sur une des pédales de mon vélo, tout les segments du dataset ne sont donc pas aussi "riches" en informations.

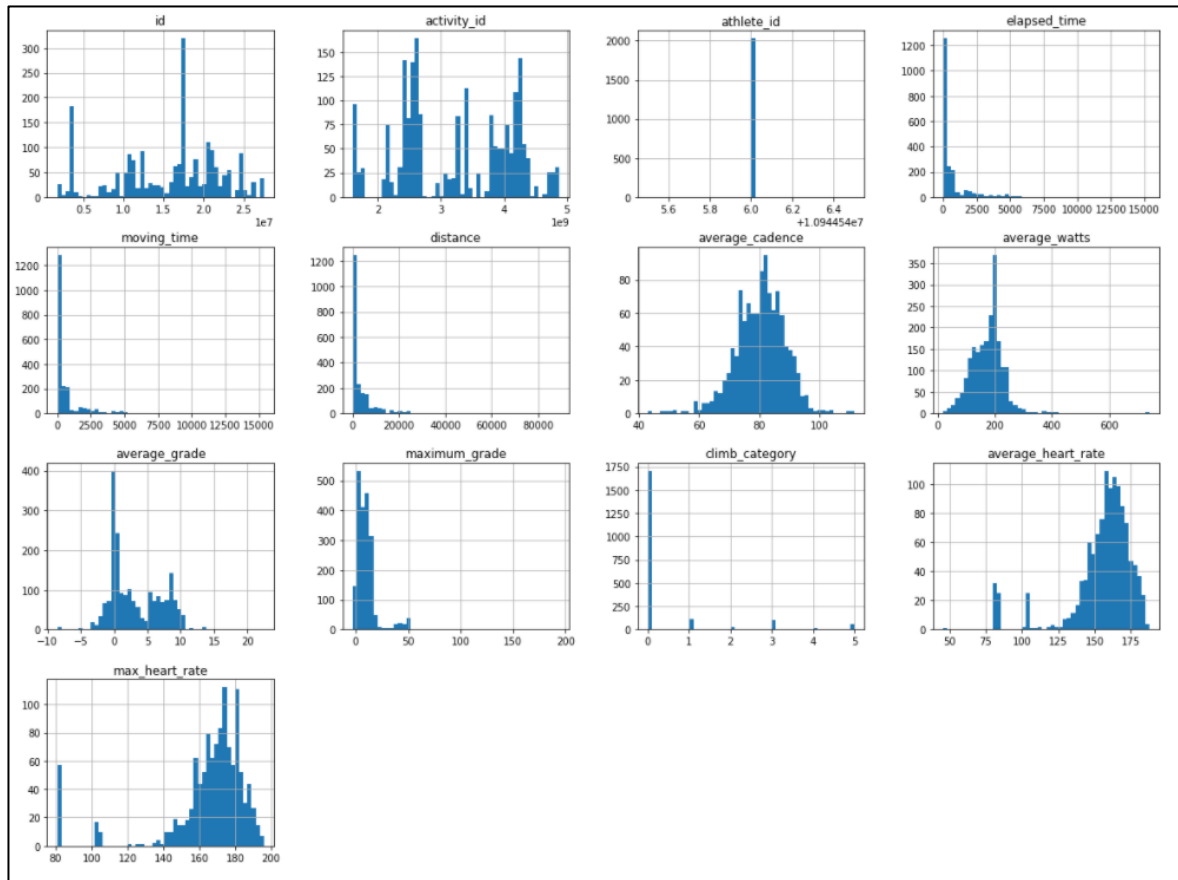
```

VirtualRide    1084
Ride           943
Name: type, dtype: int64

```

*Distribution velo réel et home trainer*

Une des particularités du projet qui m'intéressait également était de pouvoir différencier la pratique sportive "réelle" et celle faite sur un home-trainer



*Visualisation du dataset*

Cette visualisation des données a permis de constater plusieurs "problèmes". En effet le dataset étant constitué de mes données personnelles, il est plus aisé de détecter des informations aberrantes.

## Nettoyage des données

Certains segments ont un dénivelé maximum supérieur à 50%, et même pour l'un d'entre eux 193% ce qui semble très irréaliste à moins de faire du vélo dans un looping... Bien que cela ne soit pas une feature que je vais utiliser dans la suite de mon projet, j'ai pris le parti de supprimer ces segments (ceux dont le dénivelé maximum est supérieur à 50%) du dataset en imaginant que le reste de la donnée ait pu avoir été faussé. Le même constat a été fait sur les battements cardiaques moyens de certains segments qui ne dépassaient pas les 100bpm quand bien ce fut sur de longues distances

Je pensais initialement que la donnée serait "propre" du fait de sa provenance (API Strava) , j'ai pu constater que certains segments étaient en double au sein de la même activité, ils ont donc été supprimés

	Nom	Temps	Vitesse	Puissance / Intensité	Vit. ascens.	FC
☆	Rising Empire Pens to Banner 20,54km 128m 0,0 %	51:05	24,1km/h	158W		—
☆	Rising Empire Pens to Banner 20,54km 128m 0,0 %	51:05	24,1km/h	158W		—
☆	NYC KOM 1,36km 88m 6,4 %	4 6:26	12,8km/h	210W	819	—
☆	NYC KOM 1,36km 88m 6,4 %	4 6:26	12,8km/h	210W	819	—
☆ 🏅	NYC KOM Reverse 1,14km 68m 5,9 %	5:19	12,9km/h	195W		—
☆	NYC KOM Reverse 1,14km 68m 5,9 %	5:19	12,9km/h	195W		—

*Segments en double au sein d'une même activité résultant d'un bug*

De plus, et j'y reviendrais plus en détails dans la suite du rapport, mais certains segments sont très long comme on peut le voir sur la visualisation des données, tous les segments de plus de 25 kilomètres ont donc été supprimés Le nettoyage des données conduit au total à la suppression de 85 segments du dataset.

## Stockage et importation en base de données

Le dataset est constitué par le biais d'une requête à l'API Strava pour récupérer toutes les activités d'un athlète donné, chaque activité est retournée par l'API par un JSON. Cette requête se fait au sein d'une classe *ImportStrava*.

```
def storage_of_new_activities(self) -> int:
    """
    Stores on the database all new activities
    Return number of activities added for frontend
    """
    activities_ids_to_added = self.get_new_activities_ids()
    activities_added = 0
    if len(activities_ids_to_added) != 0:
        for activity_id in activities_ids_to_added:
            activity_json =
self.get_activity_by_id(activity_id=activity_id)
            activity_ = adapter_data.AdapterActivity(activity_json).get()
            activity.repository.save(activity_)
            activities_added += 1
```

```

        logging.info(f"Activity {activity_id} added in database "
                     f"[{round((activities_added *
100)/len(activities_ids_to_added),2)}%]")
        logging.info(f'{activities_added} activities added to the
database')
    return activities_added

```

*Récupération des activités qui ne sont pas déjà présentes en base sous forme de JSON, transformation en objet Activity et stockage.*

Ce JSON ne va pas être stocké tel quel dans notre base de données, mais va être transformé en Objet Python au sein de notre application par une classe (*AdapterActivity*) destiné à faire la transformation JSON/Object tout en gardant uniquement les informations qui nous intéressent.

```

def get(self) -> Activity:
    return Activity(
        id=self.id(),
        athlete_id=self.athlete_id(),
        name=self.name(),
        distance=self.distance(),
        moving_time=self.moving_time(),
        elapsed_time=self.elapsed_time(),
        total_elevation_gain=self.total_elevation_gain(),
        type=self.type(),
        start_date_local=self.start_date_local(),
        average_speed=self.average_speed(),
        average_cadence=self.average_cadence(),
        average_watts=self.average_watts(),
        max_watts=self.max_watts(),
        suffer_score=self.suffer_score(),
        calories=self.calories(),
        segment_efforts=self.segment_efforts(),
        average_heart_rate=self.average_heart_rate(),
        max_heart_rate=self.max_heart_rate()
    )

```

*Méthode.get() de la classe AdapterActivity qui est chargée de récupérer toutes les clés du JSON d'entrée qui nous intéresse et qui retourne un objet Activity*

Une fois que ce JSON est transformé en Objet, il est sérialisé en JSON à l'aide de la librairie jsonpickle pour être sauvegardé dans notre base de données dans la classe Elasticsearch.

```
def save(self, athlete: Athlete):
    return self.elastic.store_data(
        data=jsonpickle.encode(athlete),
        index_name=self.index,
        id_data=athlete.id
    )
```

*Sauvegarde d'un objet Athlete en base de données, l'objet est sérialisé en JSON et sauvegardé*

La transformation inverse sera effectuée lorsqu'on voudra récupérer la donnée de la base vers l'application afin de le re-transformer en Object.

```
def get(self, id_) -> Athlete:
    result = self.elastic.search_by_id(index_name=self.index,
                                       id_data=id_)
    athlete = jsonpickle.decode(read(result.get("_source")))
    return athlete
```

*Récupération d'un JSON Athlete en base de données par son id. Le JSON est désérialisé pour être re-transformé en objet Athlete dans notre code.*

Si de prime abord ce traitement peut paraître laborieux, cela me semble à terme plus robuste de manipuler la donnée dans notre application sous forme d'objets dont chaque attribut est bien défini plutôt que de manipuler un JSON gargantuesque et ses clés dont les  $\frac{3}{4}$  ne nous servent pas. La manipulation est décrite pour les activités, mais toute la donnée provenant de l'API Strava subira le même traitement

## Méthodes

### Algorithme naïf

J'ai tout d'abord mis en place un algorithme naïf dans le but de servir de baseline pour la suite du développement de la partie Machine Learning. Le but de cet algorithme est de faire une prédiction basée sur un apprentissage relativement "stupide" et sa performance consistera une performance à battre par tout les futurs modeles entraînés. Dans le cas où nos futurs modèles ne dépassent pas l'algorithme naïf, ils devraient être ignorés.

Notre algorithme naïf calcule la vitesse moyenne de déplacement des données qui lui servent d'entraînement et prédit donc un temps de trajet équivalent à la distance divisée par cette vitesse moyenne.

**Calculation of the average speed of the train segments**

Entrée [12]: 

```
mean_speed_train = X_train['average_speed'].mean()
mean_speed_train
```

Out[12]: 5.945320881522815

**Prédiction**

Entrée [13]: 

```
X_test['elapsed_time_pred'] = X_test['distance'] / mean_speed_train
y_pred = X_test['elapsed_time_pred'].values
```

*“Entrainement “ de l’algorithme naïf*

On calcule ensuite diverses métriques afin de pouvoir permettre une comparaison avec nos futures modèles

```
naive_mae : 258.3308683024614
naive_mape: 0.36804839590606236
naive_rmse: 559.4760824114861
```

*Performance de l’algorithme naïf*

A noter qu’en testant cette algorithme sur le parcours de l’Alpe d’Huez qui sert un peu de fil rouge à ce projet, j’ai obtenu une prédiction de 38min30sec ce qui semble tout aussi irréaliste que la prédiction fourni par Strava

## Features

Je n’ai conservé que 3 des features initiales de chaque segment qui sont “elapsed\_time” (ce que nous cherchons à prédire), “ distance “ et “ climb category” qui est une feature créée par Strava pour catégoriser les montées. Cette dernière se calcule en multipliant la longueur du segment par son dénivelée moyen et en fonction du résultat le segment est catégorisé avec une valeur allant jusqu’à 5 si le résultat est supérieur à 80 000.

Les autres features utilisées sont construites à partir des données connues et ont pour but de traduire un état de forme de l’athlète :

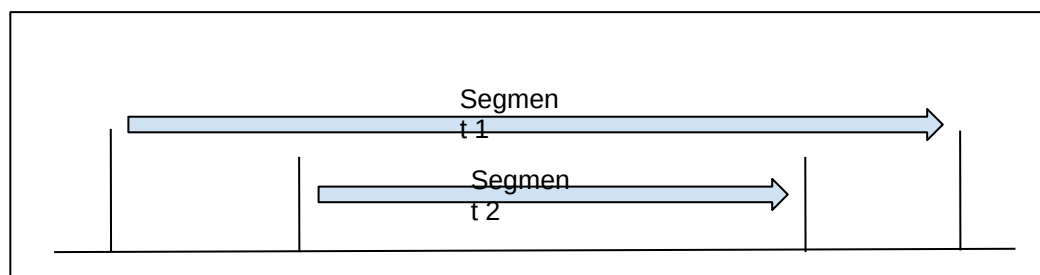
- time\_activities\_last\_30d, le temps d’activité en minutes lors des 30 derniers jours
- days\_since\_last\_activity, le nombre de jours depuis la dernière activité



- `type_virtual_ride`, est ce que le segment a été réalisé dans la vraie vie ou sur un home trainer
- `average_climb_cat_last_30d`, la catégorie moyenne des ascensions lors des 30 derniers jours
- `average_speed_last_30d`, la vitesse moyenne lors des 30 derniers jours d'activités en m/s

## Split Train/Test

Au cours de la conception de la partie IA, j'ai pu m'apercevoir d'un problème avec les données qui me servent de dataset. Comme évoqué précédemment les données de mon dataset sont des portions d'une activité sportive, qui sont créés par des utilisateurs de l'application STRAVA. Comme leur création est libre, il se trouve que certains segments sont compris dans d'autres segments ce qui amène à une probable fuite de données lors du split train/test



*Segments imbriqués les uns dans les autres au sein d'une même activité*

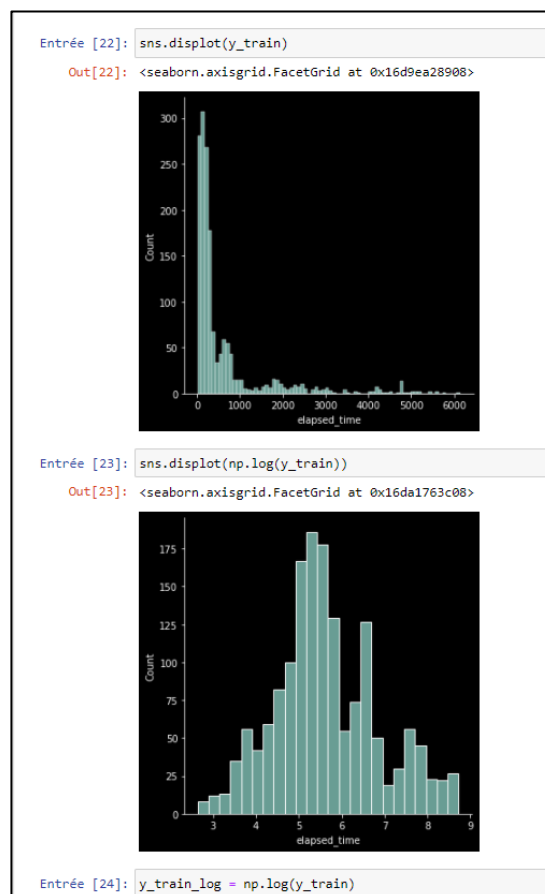
Dans cet exemple qui représentent deux segments d'une même activité, lors du split, si le segment 1 se retrouve dans le train set et le segment 2 dans le test set on peut imaginer d'une certaine manière que la performance prédictive du modèle lorsqu'il faudra donner un temps de trajet pour le segment 2 pourra être biaisé car une information relativement similaire lui aura servi d'entraînement. Pour pallier à ce problème et dans l'impossibilité de modifier la nature du dataset par manque de temps, j'ai choisi de réaliser un split train/set "maison"

On ajoute comme "feature" à chaque ligne du dataset à partir de la date de l'activité, son jour calendaire et l'année (Exemple : le 01/01/2021 aura pour valeur 012021). Suite à cela on crée une liste de toutes les valeurs possibles de ces clés numériques présentes dans le dataset et on en choisit une portion (20 %) au hasard. Tous les segments ayant comme clé numérique une de celles choisies au hasard constitueront le test set, les autres iront dans le train set. De cette manière on s'assure que les segments d'une même activité ne puissent pas se retrouver à la fois dans le train et le test set.

Cette méthode même si elle constitue une alternative relativement viable peut sembler ne pas être optimale mais je n'en ai trouvé aucune assez robuste dans le temps qu'il me restait.

## Transformation du label

Le label (elapsed\_time) a été transformé dans le train set pour avoir une distribution plus centrée que l'originale



*Distribution du label "elapsed\_time" et log*

## Modèles

3 modèles de régression ont été entraînés pour comparer leurs performances respectives sur nos données : LinearRegression, XGBRegressor et RandomForestRegressor.

- LinearRegression ajuste un modèle linéaire avec des coefficients  $w = (w_1, \dots, w_p)$  pour minimiser la somme résiduelle des carrés entre les cibles observées dans l'ensemble de données et les cibles prédites par l'approximation linéaire.
- XGBRegressor, implémentation open source optimisée de l'algorithme de boosting de gradient, dont le principe est de combiner les résultats d'un ensemble de modèles plus simples et plus faibles afin de fournir une meilleure prédiction.
- 
- RandomForestRegressor, Une forêt aléatoire est un méta-estimateur qui ajuste un certain nombre d'arbres de décision de classification sur divers sous-échantillons de l'ensemble de données et utilise la moyenne pour améliorer la précision prédictive et contrôler le sur-ajustement

Les performances de l'entraînement sont les suivantes

	MAE	MAPE	RMSE
xgbreg	117.565773	0.185264	309.343164
forrest	106.915398	0.192333	248.118892
naive	258.330868	0.368048	559.476082
linreg	623.807504	0.751434	2245.175991

*Performance de nos différents modèles*

On peut constater que deux des trois modèles que nous avons entraînés dépassent les performances de notre baseline. Le problème évoqué précédemment sur la nature du dataset, rend compliqué le tuning des différents modèles. L'amélioration des hyperparamètres avec GridSearch utilise la cross validation et du fait de cette probable fuite de données, je ne saurais conclure sur le fait que la performance du modèle s'est réellement amélioré ou si une des cross validation a segmenté le dataset d'une manière qui permet d'avoir à prédire des données qui ont plus ou moins servi à l'entraînement. La métrique qui serait considéré pour la suite du projet sera la MAPE (Mean Absolut Percentage Error)

## Segmentation de la donnée à prédire

Suite à l'entraînement de ces modèles, j'ai souhaité leur faire prédire mon temps de trajet sur le parcours que j'évoque depuis le début de ce rapport, l'Alpe d'Huez, afin de m'assurer que les prédictions étaient relativement cohérentes.

Les données sur lesquelles nous allons faire des prédictions ne sont pas des segments comme les données d'entrainements, mais des routes que nous allons devoir segmenter nous même en morceau logique en fonction de leur dénivelé moyen. L'application reçoit les informations sur la route à prédire de la part de l'API Strava sous forme de.gpx. Le.gpx est un format de fichier basé sur le XML qui permet l'échange de coordonnées GPS. Il se présente comme un ensemble de points présentant diverses informations (latitude, longitude, élévation etc...). L'ensemble de ces points représente le trajet effectué

```
<trk>
  <name>Alpe d'huez - Projet Simplon</name>
  <link href="https://www.strava.com/routes/2787335981548134218"/>
  <type>Vélo</type>
  <trkseg>
    <trkpt lat="45.05476" lon="6.031770000000001">
      <ele>722.0200000000001</ele>
    </trkpt>
    <trkpt lat="45.054750000000006" lon="6.03199">
      <ele>721.9700000000001</ele>
    </trkpt>
  </trkseg>
</trk>
```

*Fichier.gpx*

Le.gpx est ensuite parsé avec la librairie python gpxpy ce qui permet de le transformer en liste de dictionnaire dont chacun comprend la latitude, la longitude et l'élévation. Afin de connaître la distance en mètres entre chaque point, j'ai utilisé une fonction de haversine qui permet de déterminer la distance entre deux points d'une sphère à partir de leurs longitudes et latitudes

	latitude	longitude	elevation	distance_to_last_point	total_distance
0	45.054760	6.03177	722.02	0.00	0.00
1	45.054750	6.03199	721.97	17.32	17.32
2	45.054890	6.03240	721.56	35.77	53.09
3	45.055110	6.03276	721.56	37.39	90.48
4	45.055505	6.03322	721.31	56.87	147.35

*Calcul des distances entre deux points*

Suite à ces calculs, l'idée était de déterminer des segments à partir de cette collection de point, afin de se retrouver avec une information du type " segment 1, tant de % de dénivelée, x mètres " et ainsi de suite. J'ai choisi d'utiliser une fenêtre glissante de taille 6 et de pas 1 pour comparer les différences d'altitude entre les extrémités de cette fenêtre afin de ne pas me retrouver avec des segments trop petits. En effet, la taille de la fenêtre glissante permet d'absorber les petites variations de dénivelé.

	latitude	longitude	elevation	distance_to_last_point	total_distance	rw_altitude_gain
0	45.054760	6.031770	722.02	0.00	0.00	NaN
1	45.054750	6.031990	721.97	17.32	17.32	NaN
2	45.054890	6.032400	721.56	35.77	53.09	NaN
3	45.055110	6.032760	721.56	37.39	90.48	NaN
4	45.055505	6.033220	721.31	56.87	147.35	NaN
5	45.055900	6.033680	721.18	56.87	204.22	-0.84
6	45.056370	6.034200	721.22	66.33	270.55	-0.75
7	45.056840	6.034720	721.69	66.33	336.88	0.13
8	45.057430	6.035395	723.39	84.35	421.23	1.83
9	45.058020	6.036070	725.51	84.35	505.58	4.20

*Fenêtre glissante pour calculer les différences d'altitude*

il s'agit ensuite de regrouper les points en segments en fonction de la colonne "rw\_altitude\_gain", à chaque changement de signe de la valeur de cette colonne, c'est un nouveau segment pour obtenir un résultat ressemblant à ceci

	latitude	longitude	elevation	distance_to_last_point	total_distance	rw_altitude_gain	segment
0	45.054760	6.031770	722.02	0.00	0.00	NaN	0.0
1	45.054750	6.031990	721.97	17.32	17.32	NaN	0.0
2	45.054890	6.032400	721.56	35.77	53.09	NaN	0.0
3	45.055110	6.032760	721.56	37.39	90.48	NaN	0.0
4	45.055505	6.033220	721.31	56.87	147.35	NaN	0.0
5	45.055900	6.033680	721.18	56.87	204.22	-0.84	0.0
6	45.056370	6.034200	721.22	66.33	270.55	-0.75	0.0
7	45.056840	6.034720	721.69	66.33	336.88	0.13	1.0
8	45.057430	6.035395	723.39	84.35	421.23	1.83	1.0
9	45.058020	6.036070	725.51	84.35	505.58	4.20	1.0
10	45.058160	6.036250	726.08	21.03	526.61	4.90	1.0
11	45.058535	6.036815	726.85	60.89	587.50	5.63	1.0

*Segmentation*

Pour finir il faudra juste calculer pour chaque segment sa distance ainsi que son dénivelé puis rajouter à ces informations les features que nous avons calculées pour l'entraînement, de manière à retrouver une donnée similaire à notre dataset.

	distance	altitude_gain	vertical_drop	all_points
0	270.55	-0.80	-0.30	[[45.05476, 6.031770000000001], [45.0547500000...
1	422.22	4.33	1.03	[[45.056370001328915, 6.034199995890939], [45....
2	427.89	-5.51	-1.29	[[45.0590500000000006, 6.0379000000000005], [45...
3	12615.51	1063.27	8.43	[[45.0627200000000006, 6.03701], [45.062850000...

*Finalité de la segmentation*

De cette manière nous pouvons comparer les prédictions de nos différents modèles sur la route qui constitue notre fil rouge et constater que cela semble déjà bien plus cohérent que les prédictions fournies par Strava.

	Prediction	Mean_speed(km/h)
naive	0h38m30sec	21.403155
linreg	2h7m9sec	6.481848
xgbreg	1h6m1sec	12.483203
forrest	1h22m40sec	9.968394
strava	0h31m39s	26.040133

*Prédiction de temps de trajet sur l'Alpe d'Huez*

## Axes d'amélioration

Le principal axe d'amélioration de la partie IA de mon projet consisterait à changer la nature du dataset et que cela ne soit plus des segments obtenus de l'API Strava, mais l'ensemble du trajet effectué que je segmenterais moi même de la même manière que la donnée à prédire. L'activité sportive est "capturée" par un appareil comme un téléphone mobile ou un compteur puis est transmise à Strava sous forme d'un fichier.gpx qui va le traiter pour afficher les informations qui l'intéresse sur son site (par exemple si vous avez fait une activité sans capteur de puissance, Strava va estimer votre puissance) . Une fois que ce processus a eu lieu, il est assez compliqué de récupérer le fichier initial qui m'aurait permis de segmenter les routes moi même et d'avoir un dataset plus cohérent. Malheureusement je me suis rendu compte de ce problème assez tard dans la réalisation de ce projet, et j'ai préféré présenter un projet relativement consistant plutôt que d'entamer un gros chantier a quelques semaines de l'échéance qui aurait été de modifier mon code en profondeur pour changer la nature des données obtenues.

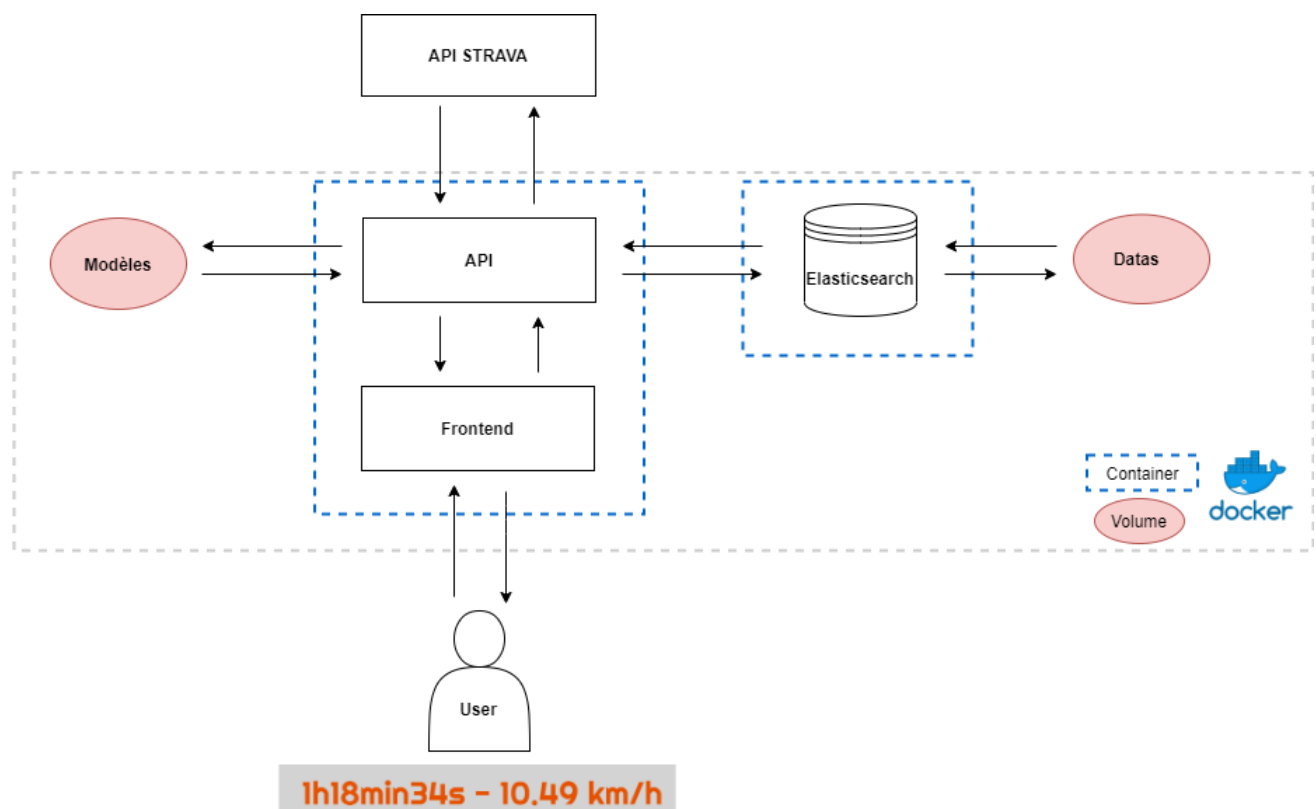
# Application

## Description

J'ai choisi de d'essayer de créer une application qui permette à l'utilisateur de réaliser lui-même, et de la manière la plus simple possible pour lui, toutes les actions nécessaires au fonctionnement global de mon projet. En conséquence l'utilisateur peut :

- Authentifier l'application en son nom pour que celle ci puisse accéder à ses données STRAVA et ce de manière automatique, une fois que le token d'accès est expiré, l'application le rafraîchit automatiquement
- Importer toutes les données (activités, routes) utiles au fonctionnement de l'application de l'API Strava dans la base de données, d'un simple clic.
- Entraîner des modèles à partir des données d'entraînements toujours d'un simple clic.
- Sélectionner une des routes importées de l'API Strava et demander une prédiction à l'application (qui sélectionne le meilleur modèle entraîné)

L'ensemble des composants de l'application (api, base de données) se trouveront dans des containers Docker permettant leur utilisation par n'importe qui sous réserve que l'utilisateur dispose d'un compte Strava.



## Base de données

### Choix du type de base de données

J'ai choisi d'utiliser Elasticsearch pour stocker les données de mon application pour diverses raisons :

- J'avais une connaissance préalable d'Elasticsearch car j'utilise cette technologie dans le cadre de mon alternance et cela me paraissait plus judicieux de capitaliser sur les nombreuses recherches que j'avais pu faire dans le cadre de mon travail pour les utiliser dans la réalisation de ce projet.
- La donnée reçue de l'API Strava est sous forme de JSON, hormis la transformation en Objet Python au sein de l'application, je peux stocker directement la donnée dans ma base sans d'autres transformations. La mise en place de la base, l'importation de données, et leurs stockage se faire par le biais d'un code beaucoup plus concis et rapide à mettre en place du fait de l'absence de tables et de relations (bien qu'on puisse au besoin quand même créer des relations entre les données par l'intermédiaire d'une clé commune par exemple)
- L'absence de jointures permet un accès à la donnée plus rapide, rendant l'utilisation de l'application plus fluide et agréable pour l'utilisateur, comme par exemple pour l'affichage des cartes qui sont créés en temps réel à partir d'une requête à la base de données
- La nature même des données que j'utilise, comme les traces GPS, me semblait être un frein à l'utilisation d'une base de données relationnelle, il aurait fallu que chaque ligne d'une table représente un unique point de coordonnées GPS avec une clé étrangère le reliant à une autre table représentant l'activité dont le point est issu, et ceci multiplié par des dizaines de milliers de points.

### Schéma de la base de données

Comme évoqué précédemment, n'utilisant pas une base de données relationnelles, je n'ai donc pas de tables. J'ai uniquement créé des index, un par type de données :

- *index\_activity*, comprenant toutes les activités cyclistes réalisés par l'athlète avec un JSON par activité
- *index\_route* comprenant toutes les routes créées par l'utilisateur sur le site de Strava, et pour lesquelles il voudra obtenir une prédiction avec un JSON par route.



- *index\_model*, comprenant toutes les métadonnées des entraînements de modèles (données qui ont servi à l'entraînement, résultat du nettoyage de données, features ajoutés, transformation, métriques, type de modèle, ratio train/test, date d'entraînement, durée d'entraînement...) avec un JSON par entraînement de modèle
- *index\_athlete*, comprenant les informations sur l'athlète (l'utilisateur) essentiellement les tokens d'accès à son compte Strava que l'application utilisera pour mettre à jour ses activités et routes.

## Utilisation

Les principales requêtes d'Elasticsearch ont déjà été décrites dans la partie "Stockage et Importation en base de données". Chaque donnée est représentée dans le code en objet Python et dispose d'une classe dédiée à ses relations avec la base de données. Cette classe s'appelle *Repository* et elle est composée de méthodes qui décrivent les requêtes qui peuvent être faites soit pour le stockage soit pour l'importation.

```
from typing import List

class Athlete:

    def __init__(self, id_: int, refresh_token: str, access_token: str,
                  token_expires_at: int, firstname: str, lastname: str):
        self.id = id_
        self.refresh_token = refresh_token
        self.access_token = access_token
        self.token_expires_at = token_expires_at
        self.firstname = firstname
        self.lastname = lastname

class AthleteRepository:

    def get(self, id_) -> Athlete:
        raise NotImplementedError()

    def get_all(self) -> List[Athlete]:
        raise NotImplementedError()

    def save(self, athlete: Athlete):
        raise NotImplementedError()

    def search_if_exist(self, firstname, lastname) -> Athlete:
        raise NotImplementedError()
```

```
repository: AthleteRepository
```

*Classe Athlete et son Repository définissant les “ modalités ” de communication avec la base de données.*

La classe réellement chargée des actions avec la base de données héritera de cette classe *Repository*. Ce concept m'a été présenté et expliqué par un collègue de travail et j'ai souhaité essayer de l'implémenter dans mon application. L'intérêt principal est que les classes “ métier ” comme *Athlète*, *Activity*, *Route* ne dépendent pas de la base de données. Si nous décidions de changer de type de base de données à l'avenir, il n'y aurait pas à modifier en profondeur le code mais uniquement redéfinir les méthodes de la classe *Repository* dans la nouvelle classe chargée des requêtes de la base de données.

```
class ElasticAthleteRepository(AthleteRepository):
    index = "index_athlete"

    def __init__(self, local_connect=True):
        self.elastic = Elasticsearch(local_connect=local_connect)
        self.elastic.add_index(self.index)

    def get(self, id_) -> Athlete:
        result = self.elastic.search_by_id(index_name=self.index,
                                           id_data=id_)
        athlete = jsonpickle.decode(read(result.get("_source")))
        return athlete

    def get_all(self):
        return self.elastic.search_index(index_name=self.index)

    def save(self, athlete: Athlete):
        return self.elastic.store_data(
            data=jsonpickle.encode(athlete),
            index_name=self.index,
            id_data=athlete.id
        )
    ...
```

*Classe ElasticAthleteRepository qui hérite de AthleteRepository et qui définit ses méthodes. En cas de modification du type de la base de données, il suffirait de redéfinir ces méthodes : la classe Athlete ne dépend pas de la base de données.*

## Mise en place

La base de données à été mise en place très simplement au moyen d'un docker-compose. J'ai aussi utilisé Kibana pendant le temps de développement afin de pouvoir visualiser les données dans ma base et m'assurer que tout se déroulait correctement lors du stockage des données.

```

elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:7.7.1
  environment:
    - node.name=es01
    - node.master=true
    - node.data=true
    - cluster.name=elk-docker-swarm-cluster
    - bootstrap.memory_lock=false
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    - "discovery.type=single-node"
  ports:
    - "9200:9200"
  expose:
    - "9200"
  networks:
    - elastic
  volumes:
    - elastic-data:/usr/share/elasticsearch/data
    - ./snapshots:/mnt/snapshots

- ./custom_elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml

```

*Définition du service elasticsearch dans le docker-compose*

## Backup

Bien que la création d'un backup dans le contexte de mon application ne me semble pas réellement nécessaire (En cas de crash de la base de données, la donnée n'est pas réellement perdue puisque ce n'est qu'une "copie" de la donnée de votre compte Strava, et la récupération dans la base de données n'est pas très longue, il me semblerait plus logique pour l'utilisateur de relancer la récupération de ses données Strava en cas de perte plutôt que de créer un backup en local de la base de données de l'application), il est possible de créer des instantanés de votre base Elasticsearch à l'intérieur du container Docker, monté en local (ou à un autre endroit).

On spécifie tout d'abord dans un fichier de configuration l'endroit où seront stockés les snapshots à l'intérieur du container

```

#custom.elasticsearch.yml

cluster.name: "docker-cluster"
network.host: 0.0.0.0
path.repo: ["/mnt/snapshots"]

```

*Fichier de configuration "custom" pour spécifier au container elasticsearch où seront stockés les snapshots*

Dans la définition des volumes du container elasticsearch, on va spécifier le directory local ou seront “dupliqués” les snapshots (ici./snapshots) ainsi que remplacer le fichier de configuration du container par celui créé plus haut

```
volumes:
- elastic-data:/usr/share/elasticsearch/data
- ./snapshots:/mnt/snapshots
- ./custom_elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml
```

*Définition des volumes du container elasticsearch*

Il faut ensuite, une fois le container allumé, spécifier où on veut conserver les snapshots qui seront fait de la base de données, j'utilise ici l'outil “ Dev Tools” de Kibana pour plus de simplicité, mais il est possible de directement requêter l'API Elastic.

```
PUT /_snapshot/my_repository
{
  "type": "fs",
  "settings": {
    "location": "/mnt/snapshots",
    "compress": "true"
  }
}
```

*Spécification de la localisation des snapshots*

En résumé, tous les snapshots seront conservés dans le container dans le dossier /mnt/snapshots mais aussi dans le dossier local /snapshots.

Pour la sauvegarde il faut une nouvelle fois requêter l'API Elastic pour créer une sauvegarde que nous appelons ici “ snapshot\_1 ”.

```
PUT /_snapshot/my_repository/snapshot_1?wait_for_completion=true
```

*Réalisation d'un snapshot*

Ce PC > Windows (C:) > simplon > cycling_travel_time > snapshots				
	Nom	Modifié le	Type	Taille
	indices	27/03/2021 09:38	Dossier de fichiers	
	index.latest	27/03/2021 09:38	Fichier LATEST	1 Ko
	index-4	27/03/2021 09:38	Fichier	2 Ko
	meta-DDDPn2ktQ5KSh_rVvi55hg	27/03/2021 09:38	Fichier DAT	10 Ko
	snap-DDDPn2ktQ5KSh_rVvi55hg	27/03/2021 09:38	Fichier DAT	1 Ko

*Dossier local avec la sauvegarde de notre base de données*

Ces fichiers représentent l'état de notre base de données au moment où nous avons réalisé la sauvegarde, ils ne sont pas lisibles et ne permettent pas de consulter la donnée présente en base, mais ils permettront de restaurer les données en cas de problèmes. Pour l'exemple nous avons un index\_athlete constitué d'un JSON représentant un utilisateur. On le supprime pour simuler un problème, il est alors possible de spécifier une restauration de cet index particulier (ou de tous les indexs) et retrouver l'état de la donnée initiale.

```
DELETE index_athlete

POST /_snapshot/my_repository/snapshot_1/_restore
{
  "indices": "index_athlete",
  "ignore_unavailable": true,
  "include_global_state": true
}
```

*Suppression d'un index et de la donnée qu'il contient et restauration de l'état dans lequel était cet index lors de la réalisation du snapshot\_1*

## Performance

Elasticsearch renvoie dans la réponse d'une requête le temps de d'exécution en millisecondes, renseigné dans la clé "took ". J'ai réalisé des tests pour requêter mon index comportant le plus de données (index\_activity) et le temps de réponse est régulièrement inférieur à 20 millisecondes. Je n'ai donc pas cherché à optimiser quoi que ce soit de ce côté là pour l'instant.

```
{"took":12,"timed_out":false,"_shards":{"total":1,"successful":1,"skipped":0,"failed":0},"hits":{"total":{"value":215,"relation":"eq"},"max_score":1.0,"hits":[{"_index":"index_activity","_type":"_doc","_id":"48
```

## Backend

### Description

Le backend de l'application a été développé avec Fast API qui est un framework de création d'API avec Python. Il est séparé en deux parties:

- Domain, qui regroupe les classes Python représentant le " métier " en l'occurrence *Activity*, *Segment*, *Route*, *Model*, et *Predict*.
- Infrastructure avec les classes Python gérant la persistance des objets métiers ainsi que les accès à la base de données, en l'occurrence Elasticsearch (base de données), *ImportStrava* (chargée de tout l'import des données depuis l'api Strava), *AdapterData* (chargée de transformer la donnée provenant de l'API Strava en objet Python) et enfin webservice qui n'est pas une classe mais une description des différentes routes qui composent notre API.

Ceci est une application modeste du Domain Driven Design (approche de développement centrée sur le métier au travers de design patterns) qui m'a été présentée dans le cadre de mon alternance toujours par le même collègue évoqué précédemment.

### Intégration de l'IA

Le contexte de mon application veut qu'un utilisateur puisse entraîner un modèle sur ses données pour recevoir une prédiction. En conséquence, il me semblait impératif qu'il puisse de lui-même entraîner ce modèle. De ce fait il m'a paru nécessaire d'intégrer l'entraînement du modèle au backend de l'application au lieu de juste charger un modèle pré-entraîné depuis l'extérieur.

J'ai repris mon travail qui avait été fait précédemment dans des notebooks (partie IA/Méthodes) pour l'intégrer au code de mon application. Le déclenchement de l'entraînement d'un modèle par l'action de l'utilisateur amène donc aux actions suivantes au sein de la classe *Model* :

- Chargement des activités présentes en base de données
- Nettoyage des données
- Ajout des features présentées précédemment
- Split maison

- Log Label
- Calcul des métriques
- Calcul du temps d'entraînement
- Enregistrement du modèle entraîné en pickle dans le volume du container API

Chacun de ses traitements est consigné et enregistré en base de données dans l'index\_model. De cette manière on peut conserver une trace de l'entraînement de chacun des modèles, des données qui ont servi à l'entraînement, des features utilisées, du type de modèle utilisé.

```
27-Mar-21 11:02:13 - INFO : Initial Features - ['distance', 'climb_category']
27-Mar-21 11:02:13 - INFO : Data Cleaning - initial shape : (2027, 17) end
shape: (1933, 17)
27-Mar-21 11:02:13 - INFO : Features added - ['time_activities_last_30d',
'days_since_last_activity', 'type_virtual_ride', 'average_climb_cat_last_30d',
'average_speed_last_30d']
27-Mar-21 11:02:13 - INFO : Ratio train_set/total - 0.81
27-Mar-21 11:02:13 - INFO : Label - elapsed_time
27-Mar-21 11:02:13 - INFO : Features Train - ['distance', 'climb_category',
'time_activities_last_30d', 'days_since_last_activity', 'type_virtual_ride',
'average_climb_cat_last_30d', 'average_speed_last_30d']
27-Mar-21 11:02:13 - INFO : Processing - ['log_label']
27-Mar-21 11:02:13 - INFO : Model - XGBRegressor
27-Mar-21 11:02:13 - INFO : Metrics - MAE : 107.11091829475254, MAPE :
0.1837562680659872, RMSE : 323.11686128409207
27-Mar-21 11:02:13 - INFO : Training Time : 0:00:01
```

#### *Exemple de logs de l'entraînement d'un modèle*

De l'autre côté au moment de la prédiction, la classe *Predict* lors de son initialisation utilisera cette classe *Model* afin de réaliser sur la donnée à prédire exactement les mêmes traitements comme dans les exemples ci dessous

```
if "climb_category" in self.model.features_train:
    data_to_predict = self.compute_climb_category(data_to_predict)
```

*Ajout d'une feature dans la donnée à prédire si elle a servie à l'entraînement*

```
if "log_label" in self.model.processing:
    prediction = sum(np.exp(prediction_segments))
else:
    prediction = sum(prediction_segments)
```

*Retour de la prédiction sous forme d'exponentielle si on a utilisé le logarithme sur le label*

Un des axes d'amélioration serait que cette classe *Model* comporte également le modèle sous forme de pickle dans l'un de ses attributs, j'ai réalisé plusieurs essais pour essayer d'enregistrer le pickle du modèle entraîné dans ma base de données, mais ce ne fut pas concluant et j'ai préféré enregistrer les métadonnées d'un côté dans ma base, et le modèle sérialisé de l'autre dans un volume du container. Le lien entre les deux se fait par le biais d'un identifiant unique (l'ID du Json des méta données d'entraînement est aussi le nom du fichier.pickle dans le volume)


## Frontend

### Description

Le frontend à été développé avec Jinja qui est un moteur de template, ainsi que JQuery pour gérer les différentes actions de l'utilisateur et leur résultat graphiquement parlant comme par exemple un écran de chargement ou des notifications de l'impossibilité de réaliser telle ou telle action, et pour finir un peu de CSS (à l'aide du framework CSS Materialize) pour essayer d'homogénéiser dans la mesure du possible l'aspect graphique de l'application.

### Screenshots





## Cycling Travel Time

Cycling Travel Time est un projet étudiant, dont le but est de prédire une durée de déplacement à vélo par l'entraînement d'un modèle de Machine Learning sur les données d'entraînements passés. L'utilisation de cette application nécessite d'avoir un **compte Strava** sur lequel des activités ont été enregistrées.

Prénom

Nom

**CONNEXION**

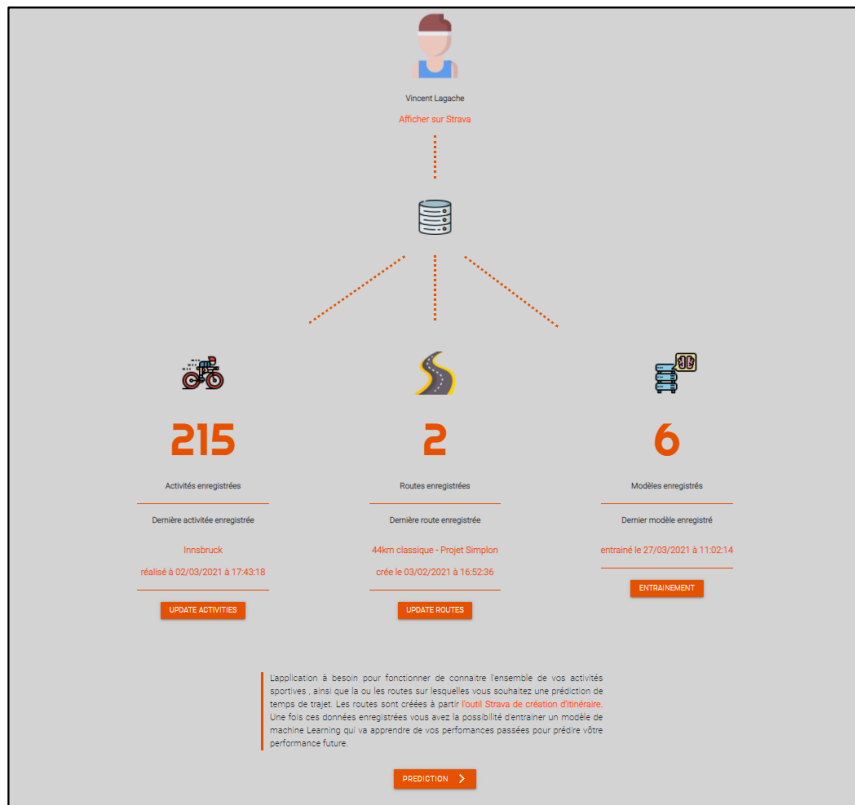
Page d'accueil avec une description de l'utilité de l'application. L'utilisateur va entrer son nom et son prénom. Cela a pour but de vérifier que l'application possède déjà en base données des tokens d'accès à l'API Strava pour cet utilisateur.

L'utilisation de cette application nécessite que vous l'autorisiez à accéder à vos données Strava.

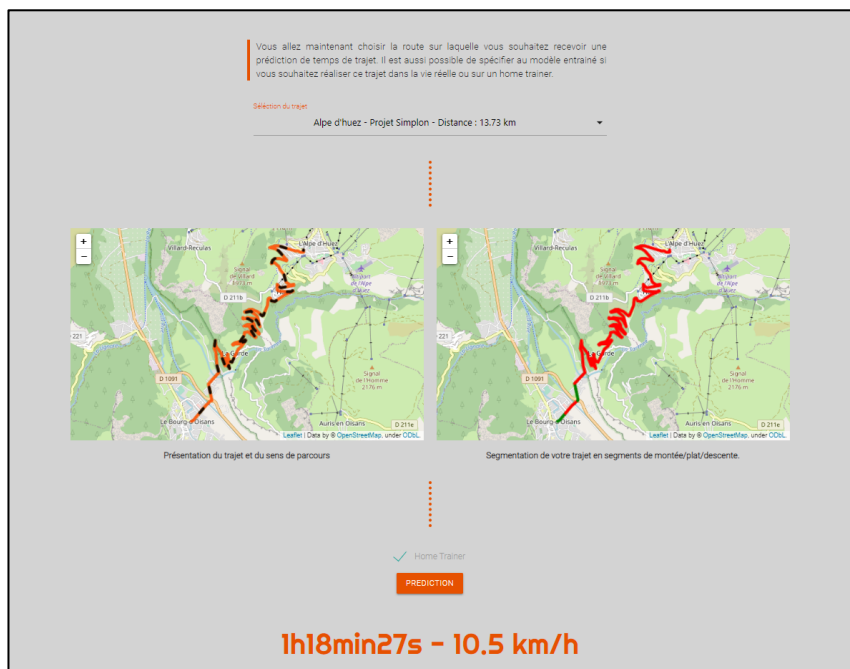
**Connect with STRAVA**

**< RETOUR**

Si l'application ne dispose des tokens d'accès, l'utilisateur est invité à donner accès à son compte Strava.



Page de l'utilisateur authentifié qui lui permet de voir le nombre des ses activités, routes, et modèles entraînés en base de données mais aussi de mettre à jour ces informations ou d'entraîner de nouveaux modèles



Page où l'utilisateur choisit la route sur laquelle il veut recevoir une prédiction et précise s'il désire réaliser cette activité dans la vraie vie ou sur un home trainer. L'application charge deux cartes représentant le trajet et la segmentation réalisé et fournit une prédiction.

# Axes d'amélioration

De nombreux axes d'amélioration ont déjà été évoqués mais pour résumer si je continuais à travailler sur ce projet, j'axerai mes efforts futurs sur :

- La modification du dataset qui sert à l'entraînement pour ne plus récupérer les segments d'activités, mais trouver un moyen de récupérer l'activité initiale avant son traitement par Strava afin de la segmenter de la même manière que la donnée à prédire pour éviter les segments imbriqués les uns dans les autres. Cela permettrait de construire des modèles plus cohérents.
- L'accès à cette application autrement que dans un contexte local dans un Docker, même si je ne suis pas sûr de la viabilité d'un tel projet avec une base de données distante. Est ce qu'un utilisateur accepterait l'export de ses données dans une base de données inconnue ? . Cela amènerait donc aussi à gérer le multi-utilisateur, car l'application au jour d'aujourd'hui ne gère qu'un unique utilisateur (exemple : lors de l'entraînement, on récupère toutes les données " activités " en partant du principe qu'elles appartiennent toutes à la même personne)
- Amélioration de la maintenabilité du code en particulier par la rédaction de plus nombreux tests, et commentaires de fonctions
- Amélioration de la maintenabilité et cohérence de la partie IA. C'est un sujet qui m'intéresse fortement dans la suite de mon parcours professionnel (MLOps). J'ai essayé dans cette application de poser la première brique de la gestion du cycle de vie d'une IA en conservant des données d'entraînements nombreuses permettant en théorie de pouvoir reproduire ces entraînements et d'éviter les modèles " boîte noire ", mais il reste encore de nombreux points à améliorer comme par exemple la gestion des modèles entraînés qui sont tous conservés pour l'instant.
- Changement du type de base de données, un des formateurs que nous avons eu au cours de cette année m'a fait remarquer lorsque je lui ai présenté mon projet que l'utilisation d'Elasticsearch était peut être un peu exagérée pour l'usage que j'en faisais. En effet c'est à la base un moteur de recherche dotés de multiples fonctionnalités que j'utilise ici uniquement pour stocker des JSON.

## Conclusion

Ce projet a été l'occasion de mettre en œuvre tout ce que j'avais pu voir ou apprendre que ce soit dans le cadre de la formation ou de l'alternance cette année. Je suis globalement satisfait du résultat obtenu qui fait office selon moi de preuve de concept suffisante pour répondre au postulat de départ qui était d'améliorer une prédiction complètement

incohérente de temps de trajet sur la route de l'Alpe D'Huez. Les prédictions fournies sont plus réalistes.

Néanmoins de nombreuses difficultés imprévues sont survenues au cours de la réalisation de ce projet et me laissent un petit goût d'inachevé en particulier sur la nature du dataset qui rend compliqué toute amélioration des modèles ou même toute certitude sur la nature des résultats obtenus.