Projet Chef d'oeuvre

DrumAI

Avril 2021

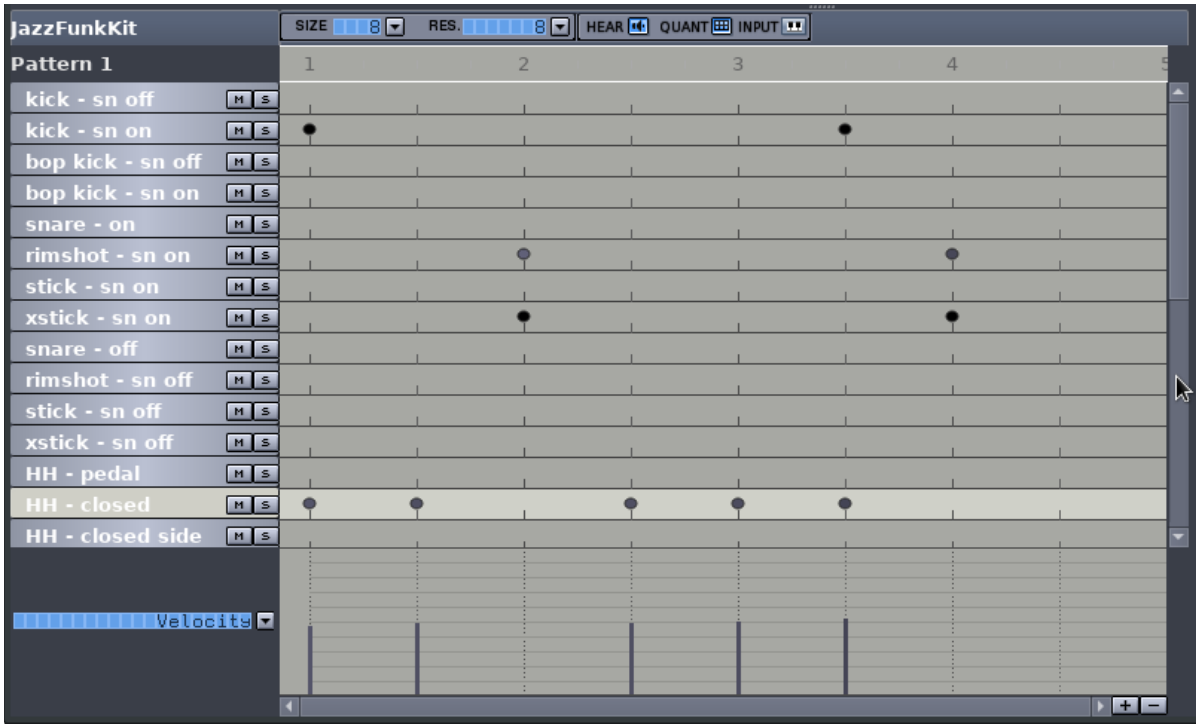
GUILLAUME MEURISSE
ECOLE IA MICROSOFT POWERED BY SIMPLON

Table des matières

Problématique

Programmation de batterie

La programmation de batterie est le processus de composition et de création de motifs rythmiques. Cela consiste à organiser et superposer différents samples (échantillons sonores) séquentiellement. Ces différents échantillons sont obtenus au travers de banques de sons qui peuvent être payantes ou gratuites.
Ces banques peuvent contenir deux types de samples. Les samples one-shot, correspondant à un événement sonore unique, et les boucles, qui forment un motif rythmique complet.
Dans le cadre de la création de motifs rythmiques, on ne s'intéressera pas à cette dernière catégorie.
Afin de créer la piste rythmique appropriée à chaque projet musical, il est important de choisir les bons samples. L' utilisation d'un grand nombre de banques peut venir compliquer ce travail de sélection.



Hydrogen Drum Machine (éditeur de pattern)

Objectifs

Afin de faciliter la sélection des samples, on pourra proposer un outil de classification automatique de samples one-shot. Cette classification pourra permettre la création d'un moteur de recherche dans une collection de samples.
L'application sera développée en python/tkinter. Elle fera appel à une api flask qui utilisera l'intelligence artificielle pour classifier les samples.
L'application permettra de :

- Classer une banque de son selon la nature de chaque son (caisse claire, cymbale...)
- Visualiser les samples dans un espace timbral. Cette visualisation tient compte des propriétés perceptives des samples et réalise une projection dans le plan. Des sons proches dans cet espace seront proches perceptivement.
- Acquérir et stocker des métadonnées de samples de manière continue.
- Recueillir les feedbacks utilisateurs sur les erreurs de classification.
- Réentrainement du modèle.

Analyse des besoins

L'application est destinée aux musiciens qui construisent leurs pistes de batterie à l'aide de samples de percussions.

Spécifications fonctionnelles

- En tant que musicien, je veux pouvoir classer les samples de mes banques de sons selon leur nature.
- Je veux visualiser les sons dans l'espace en fonction de leur timbre.
- Je veux pouvoir importer mes samples dans le logiciel client en sélectionnant un répertoire.
- Je veux pouvoir corriger les erreurs de classification
- Afin de recueillir les erreurs de classifications et améliorer l'application, quand un utilisateur propose une correction, on doit pouvoir recueillir les features du sample et la classification proposée par l'utilisateur.
- Afin de ne pas enfreindre le copyright des banques de sons commerciales, aucune donnée permettant de reconstruire les samples d'origines ne sera enregistrée ni stockée.
- Afin de limiter les erreurs parmi les corrections apportées par les utilisateurs, les corrections de classification devront tenir compte des corrections apportées par plusieurs utilisateurs sur un même sample.

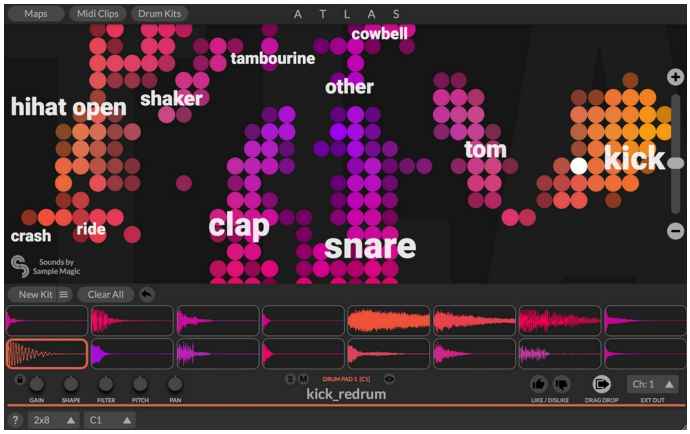
Existant

Applications existantes

Algonaut Atlas

https://www.algonaut.tech/

- Labellisation automatique
- Visualisation



Atlas

XLN Audio XO

https://www.xlnaudio.com/products/xo

- Clusterisation, recherche de similarité
- Visualisation 2D

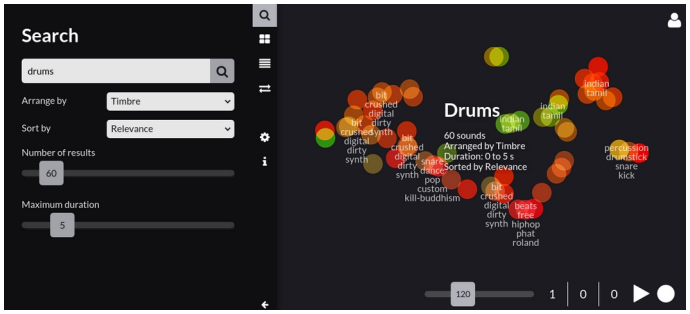
XLN Audio XO

Freesound Explorer et Timbral explorer

https://labs.freesound.org/fse/
https://labs.freesound.org/apps/2019/01/30/timbral-explorer.html

- Visualisation 2d de sons de la bibliothèque Freesound dans un espace timbral (timbral models)

Ces outils sont des démonstrations des modèles de timbre développés pour le projet Audio Commons. Ces modèles ont pour objectif de caractériser les sons selon leur propriétés de timbres.



Freesound explorer

Littérature scientifique

- Classification:
  - Automatic Classification of Drum Sounds: A Comparison of Feature Selection Methods and Classification Techniques
  - Automatic Classification of Drum Sounds with Indefinite Pitch
- Transcription automatique:
  - Improving Perceptual Quality of Drum Transcription with the Expanded Groove MIDI Dataset

Ces Publications ont permis d'identifier des descripteurs audio pertinents pour des tâches de classifications de sons de percussions. Elles ont également permis d'identifier des datasets potentiels.

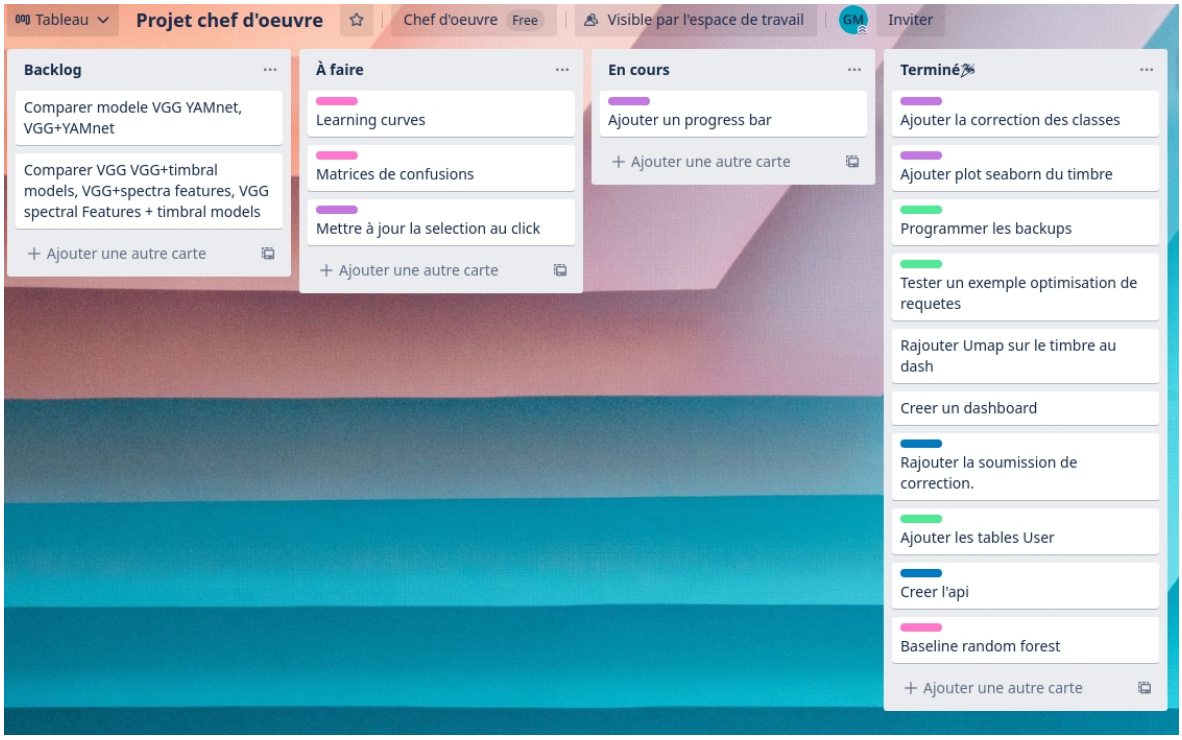
Modèles pré entraînés

YAMNet
https://github.com/tensorflow/models/tree/master/research/audioset/yamnet
VGGish
https://github.com/tensorflow/models/tree/master/research/audioset/vggish

Gestion de projet

Trello

Un kanban sur Trello a été initié. Le code couleur représente différents aspects du projet: Base de données, API, Modèles d'IA et application client.



Trello

GIT

Le projet est disponible sur un repo git.
https://github.com/Simplon-IA-Bdx-1/chef-doeuvre-guitoo

Intelligence artificielle

On cherche à développer un modèle de classification. Le modèle d'IA doit pouvoir prédire la classe d'un sample de percussion.

Datasets

Les datasets ont été choisis pour leurs présences dans la littérature scientifique de classification de percussions:

- Wavegan drum dataset
- Musicradar essential drumkit
- Musicradar 1000
- 200 Drum Machines
- MDLib2.2 (non retenu)
- Fraunhofer IDMT SMT (non retenu)
- Hydrogen main sound libraries

Les datasets MDLib2.2 et Fraunhofer IDMT SMT n'ont pas été retenus. Ils contiennent seulement des sons correspondants aux classes majoritaires et enregistrés sur un seul et même kit de batterie offrant une faible variété de samples et pouvant induire un overfitting et une fuite de données.
Aux datasets issus de la littérature, les datasets Hydrogen et musicradar 1000 ont été ajoutés.

Labels

Choix des classes

Instruments principaux par ordre d'abondance dans les datasets recueillis :

- Snare (caisse claire)
- Kick (grosse caisse)
- Hat (Charleston)
- Tom
- Cymbal (cymbale)
- Clap (claquement des mains)
- Cowbell (cloche)
- Conga
- Shaken ('instruments secoués)
- Tambourin
- Bongo
- Agogo
- Clave
- Timbales (timbales cubaines)
- Djembe
- Guiro
- Cajon
- Cuica
- Timpani (timbales d'orchestre)
- Tabla
- Triangle
- Gong
- Darbuka
- Clave

Certains de ces instruments peuvent produire une variété de sons différents. Certains instruments représentent une famille d'instruments divers. Il est possible d'affiner la classification précédente.

Caisse claire:
Une caisse claire dispose d'un timbre consistant en une série de ressorts positionnés contre la peau inférieure. Elle peut être désenclenchée.
Selon la position avec laquelle on frappe la caisse claire avec les baguettes, on pourra obtenir des sons différents.

- Snare_On (Snare avec timbre)
- Snare_Off (Snare sans timbre)
- Snare_Rim (rimshot)
- Snare_Side (sidestick)
- Snare_Brush
- Snare_Flam

Charleston:
Un charleston dispose d'une pédale servant à ouvrir ou fermer 2 cymbales l'une sur l'autre. Selon que la position est ouverte ou non, l'instrument produira un son différent lorsqu' on le frappe. Le charleston peut aussi produire un son de lui-même lorsque l'on passe rapidement d'une position ouverte à fermée.

- Hat_Open
- Hat_Close
- Hat_foot

Cymbales:

- Cymbale_Crash
- Cymbale_Ride
- Cymbale_China (ou trash)
- Cymbale_Splash

Instruments secoués:

- Shaken_Shaker
- Shaken_Cabasa
- Shaken_Maracas

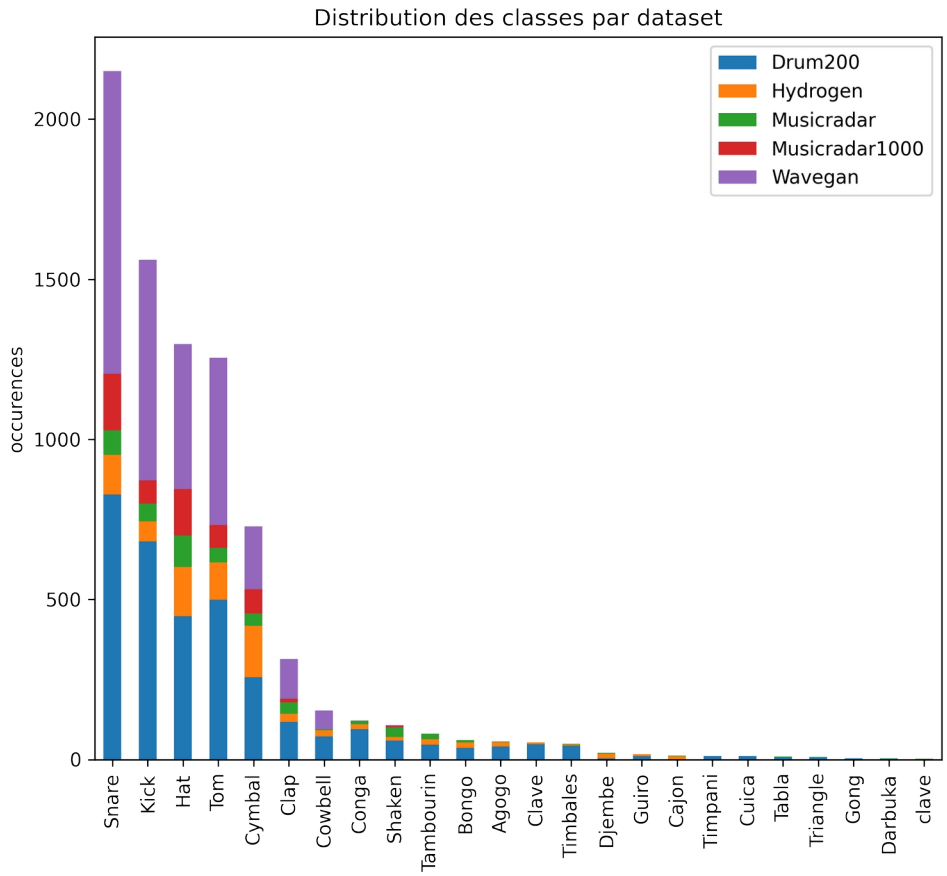
Labellisations

Afin de labelliser ces banques de sons, des expressions régulières sur les noms de fichiers ont été utilisées.
Ces expressions régulières ont été définies indépendamment pour chaque dataset.
Cette labellisation a ensuite été corrigée de manière itérative avec une succession d'entraînement et d'identification des erreurs de classification parmi les plus mauvaises prédictions faites par le modèle.

Exemple de regex sur les charlestons du dataset drum200:

- Ouvert:
  - op?[\.\-_]?(hi)?hat
  - hat[\.\-_]?o
  - ope?n
  - open hi
  - o[\.\-_]?hh
  - hh[\.\-_]?o
- Fermé:
  - ((cl?)|^op)[\.\-_]?(hi)?hat
  - hat[\.\-_]?c
  - close
  - c[\.\-_]?hh
  - hh[\.\-_]?c
- Pédale:
  - pedal
  - foot
  - hat[\.\-_]p

Distributions des classes



Features

Descripteurs

Les descripteurs audio ont été calculés en utilisant la librairie librosa (https://librosa.org/doc/latest/feature.html)

Les descripteurs sont calculés sur 1 seconde avec une fréquence d'échantillonnage de 16 kHz, une taille de FFT de 2048 et un pas de 512. Les features obtenues comptent 32 pas de temps (frame).

Zero crossing rate (zcr)

Le zero crossing rate mesure le nombre de fois que la courbe du signal traverse 0 par seconde. On obtient une valeur de zcr par frame.
Dimension de la feature: (32)

Spectral flatness

La flatness est une indication du niveau de bruit pour chaque frame.

Dimension de la feature: (32)

Spectral contrast

Le contraste indique le niveau de bruit par bande de fréquence.
Le contraste est calculé sur 8 bandes de fréquences et 32 frames.

Dimension de la feature: (8, 32)

Modèle de timbre

https://github.com/AudioCommons/timbral_models
La bibliothèque 'Timbral models' propose 7 modèles de régression sur des paramètres de timbres: hardness, depth, brightness, roughness, warmth, sharpness, et booming.

Dimension: 7 features scalaires

Mel frequency cepstrum coefficients (MFCC)

Le MFC est une représentation des sons qui met en valeur les propriétés de périodicité du spectre (harmonicité et fréquence fondamentale).
Elles sont abondamment utilisées dans le deep learning sur les sons.
Les Mfcc sont calculées sur 1 seconde de son avec un pas de 160, 64 bandes, une taille de FFT de 512 et une taille de fenêtre de 400, pour un total de 96 frames.
Dimensions: (96,64)

Embedding issu de modèles pré entraînés

Modèles YAMNet et VGGish
Features: Ces modèles sont des CNN qui prennent une mfcc en entrée

YAMNet

- Dataset et Ontologie: AudioSet (https://research.google.com/audioset/)
- Ontologie de 521 classes
- Espace latent de taille 1024
- https://github.com/tensorflow/models/tree/master/research/audioset/yamnet

VGGish

- Dataset et Ontologie: YouTube-8M (http://research.google.com/youtube8m/)
- Ontologie de 1000 classes
- Espace latent de taille 128
- https://github.com/tensorflow/models/tree/master/research/audioset/vggish

Dimensions (1024) et (128)

Importation en base de données

Le calcul des features sur l'ensemble des données prend un temps excessivement long. Pour cette raison, les features sont pré-calculées et stockées en base de données.

```
def fill_feature(key, fun, session):
  sample_ids = set(session.query(SamplePath.sample_id).all())
  filled_ids = set(
     session.query(Features.sample_id)
     .filter(getattr(Features, key).isnot(None)).all()
  )
  ids = sample_ids - filled_ids
  ids = list(ids)

  q = (
     session.query(SamplePath.path, Features)
     .filter(SamplePath.sample_id == Features.sample_id)
     .filter(Features.sample_id.in_(ids))
     .options(load_only(Features.id))
  )
  paths_and_features = q.all()

  for path, feature in tqdm(paths_and_features):
      setattr(feature, key, fun(get_full_path(path)))
      session.merge(feature)
      session.commit()
```

Code d'insertion d'une feature calculée par la fonction '*fun*' et mise en base dans la colonne '*key*'.

Méthodes

Les classes sont fortement déséquilibrées. Cela va impacter le choix de métriques et la méthode de split. Pour garantir la même proportion des classes entre les ensembles de train et validation on procède à un split stratifié (20% pour la validation) sur les classes ou les sous classes.
À l'entraînement, les 'loss' sont pondérés par des poids dépendants de la prévalence de chaque classe pour garantir que le modèle ne prédit pas spontanément les classes majoritaires.
Le macro F1 et le macro AUC (one versus one) ont été retenus comme métriques. Ils permettent de mesurer le score indépendamment de la prévalence des classes.

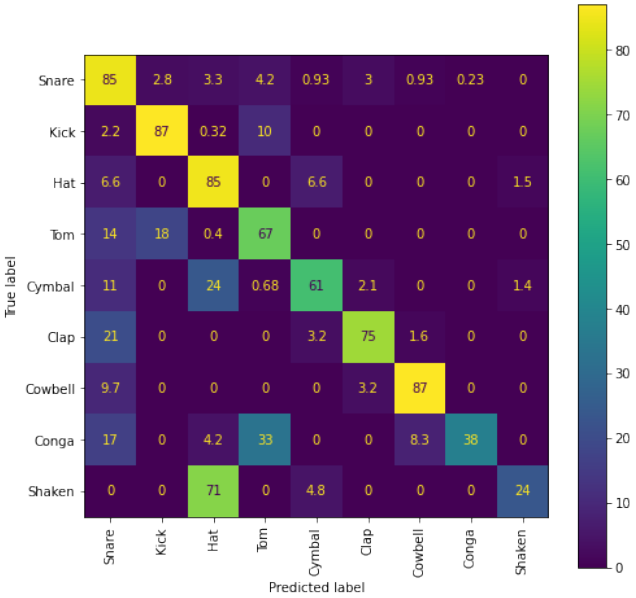
```
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=classes,
    y=y_class_train.idxmax(axis=1))

class_weights = dict(zip(range(len(classes)), class_weights))
```

Calcul des pondérations de chaque classe (scikit learn)
La liste des poids est transformée en dictionnaire pour être compatible avec Tensorflow.

Modèles

Modèle baseline: random forest

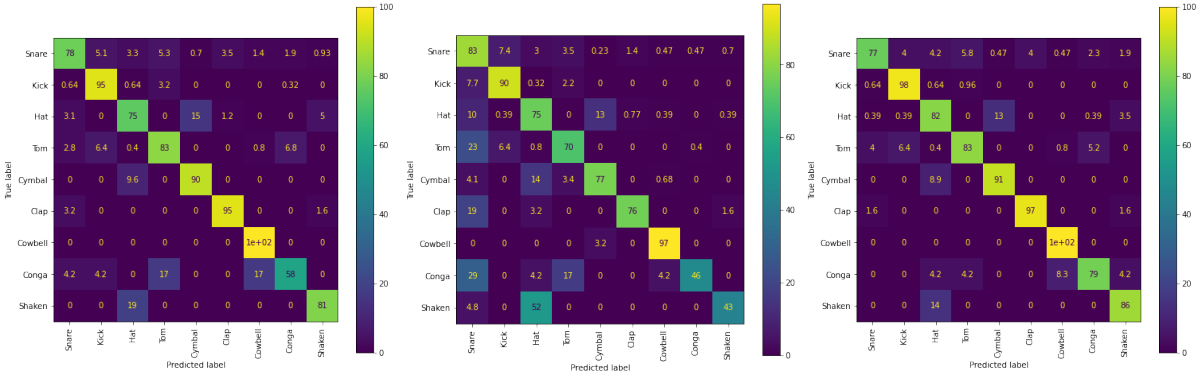
Un random forest a été entraîné sur les features timbrales. Les matrices de confusion sont normalisées pour faire mieux transparaître les performances sur les classes minoritaires.



Random Forest (normalisé)
Macro F1: 0.698
Macro AUC: 0.959

Transfer learning:

VGGish vs YAMNet



VGG (normalisé)
Macro F1: 0.77

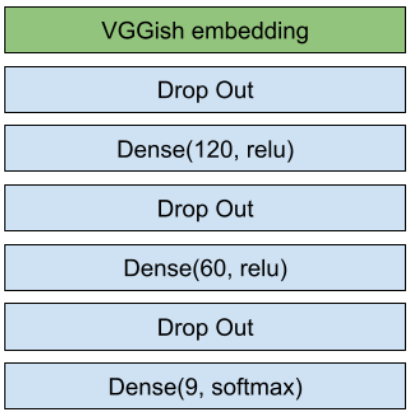
YAMNet (normalisé)
Macro F1:0.75

VGG+YAMNet (normalisé)
Macro F1:0.81

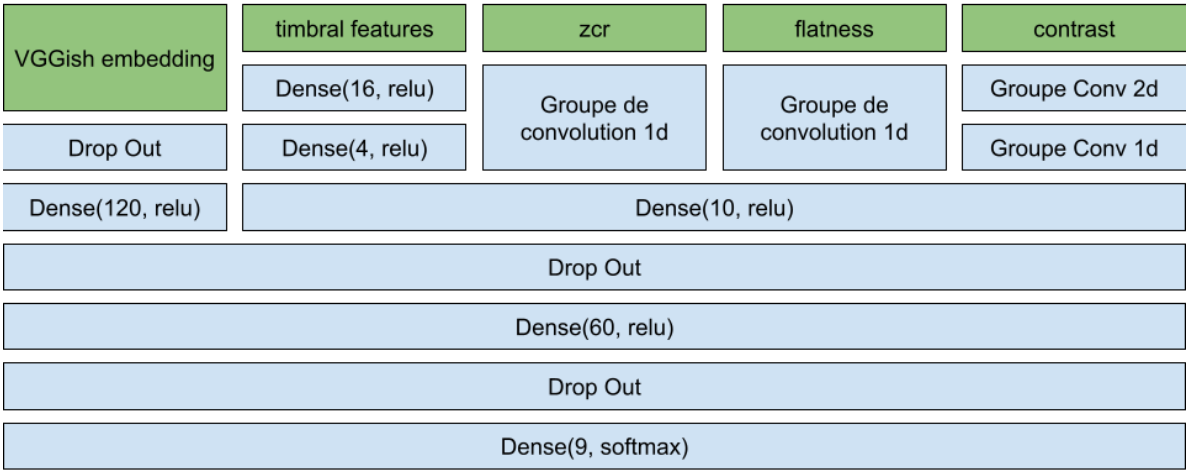
Les modèles VGGish et Yamnet ont été pré-entraînés sur des datasets différents et montrent une complémentarité sur des sons de percussions. Un modèle combinant les deux précédents possède des performances supérieures à chacun de ces modèles.
Le modèle YAMNet possédant un embedding de plus grande taille est davantage sujet à l'overfitting et par conséquent, un drop out plus important est utilisé.

Embedding, descripteurs et timbre

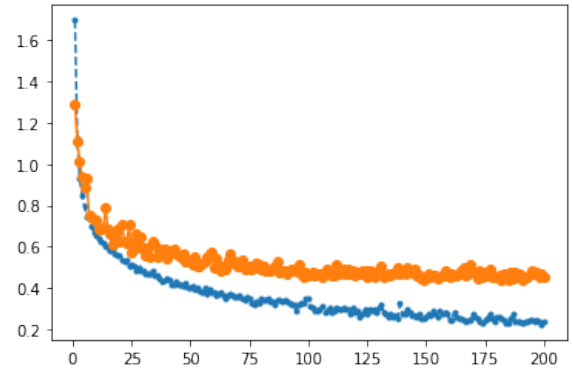
Un modèle basé sur VGGish (VGG) à été comparé avec un modèle utilisant à la fois l'embedding VGGish, les features spectrales et les features timbrales (VGG+).



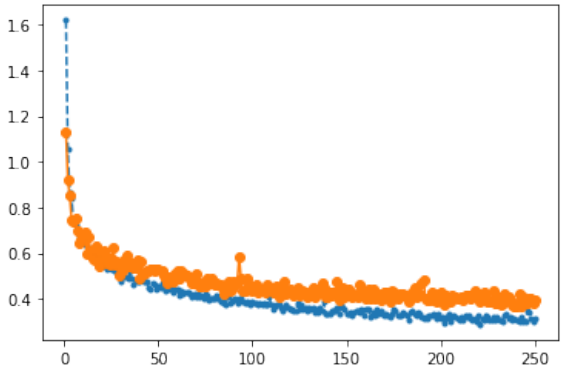
VGG



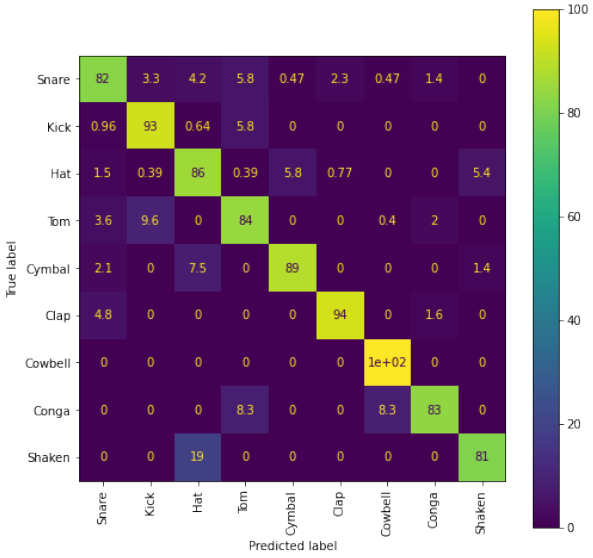
VGG +



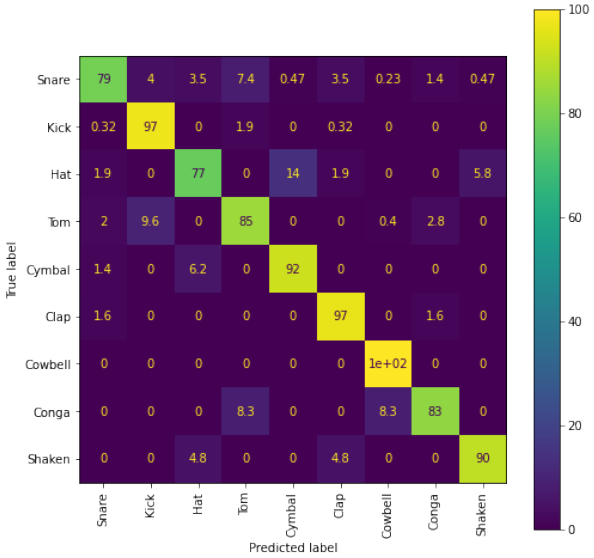
VGG



VGG +



VGG (normalisé)



VGG + (normalisé)

	Macro F1	Macro AUC
train	0.917	0.998
valid	0.835	0.986

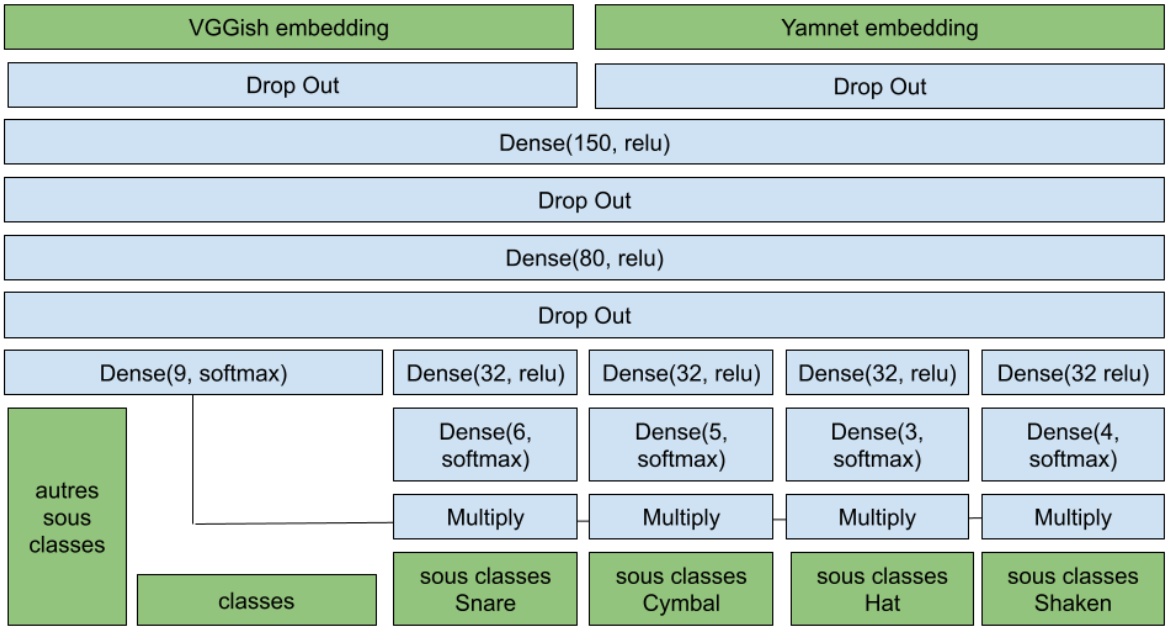
	Macro F1	Macro AUC
train	0.869	0.997
valid	0.827	0.991

Les performances de ces modèles sont très sensibles au facteur de drop out.

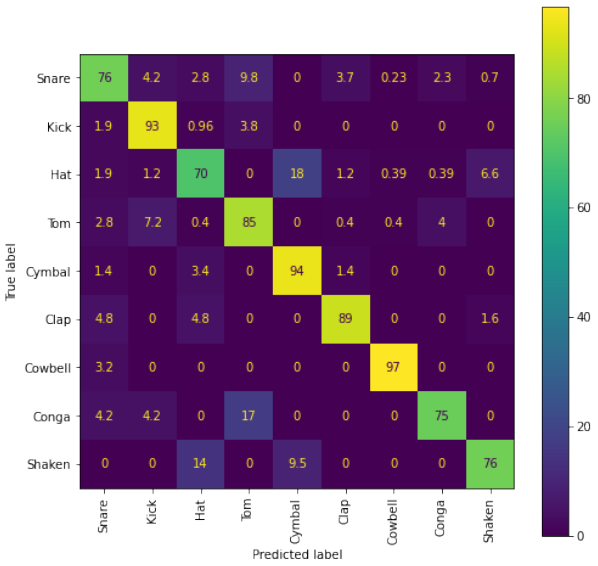
Pour comparer au mieux ces modèles, il sera nécessaire de procéder à un grid search sur les hyper-paramètres, en particulier le drop-out et le learning rate.

Modèle Hiérarchique

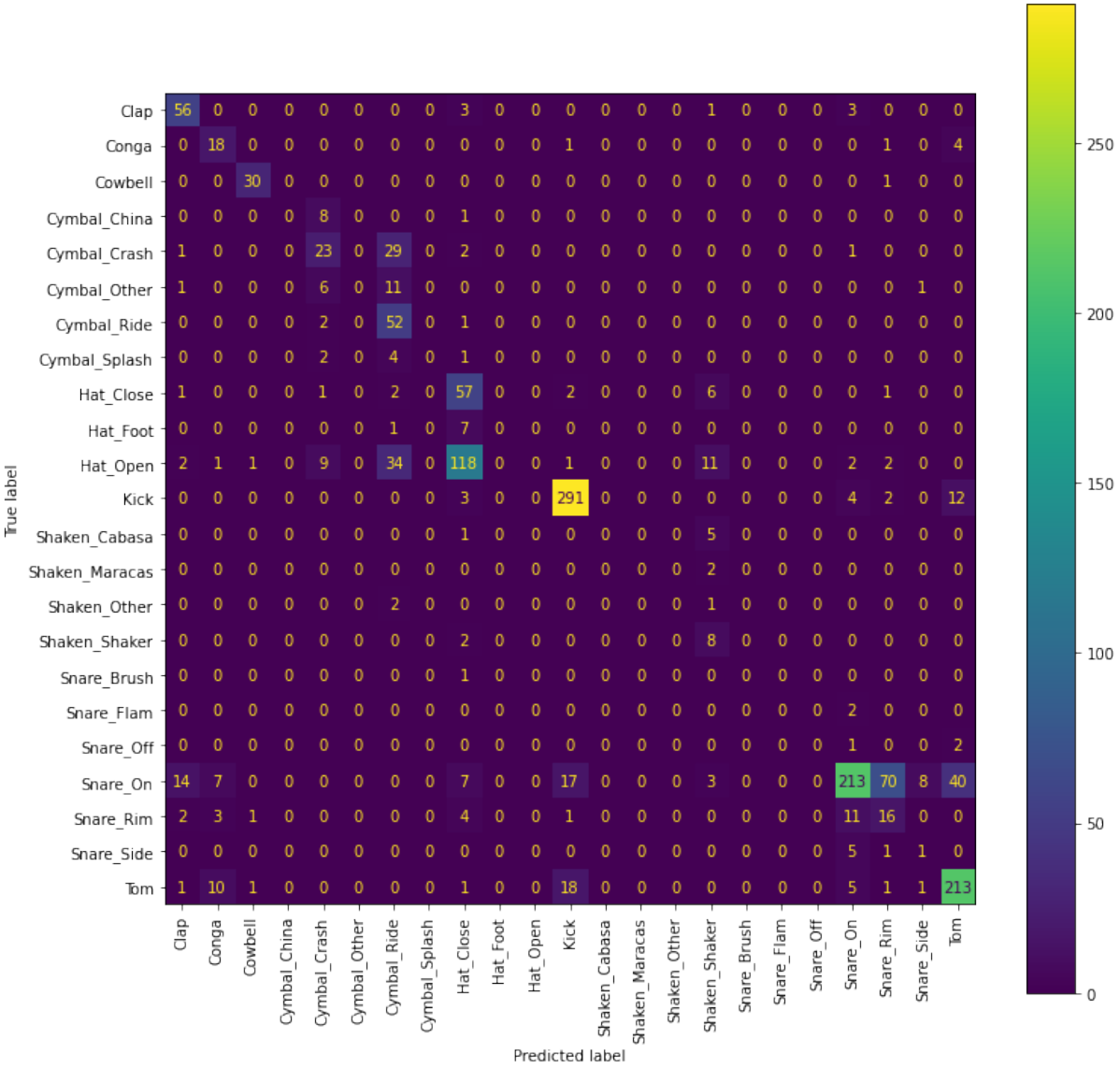
Ce modèle est conçu pour prédire simultanément les classes et sous-classes. La prédiction des classes est utilisée pour pondérer les résultats sur les sous-classes. Par exemple, la prédiction de la classe Snare sert à pondérer les scores des sous classes Snare. De cette façon, la somme des scores sur les sous classes fait toujours 1. Certaines classes sont très hétérogènes. Un modèle hiérarchique permettrait d' obtenir de meilleurs score de prédiction sur les classes en permettant au modèle de tirer parti des sous classes.



Modèle Hiérarchique



classes (normalisé)



sous classes

Les faibles performances du modèle hiérarchique peuvent s'expliquer par le très grand déséquilibre des sous-classes et des erreurs éventuelles de labélisations.. Afin d'améliorer ce modèle, il faudra procéder à un examen des erreurs de prédiction sur les ensembles de train et validation. Il faudra également compléter les données sur certaines sous-classes.

Application

Architecture

Le projet est constitué d'un client, d'un webservice API, d'une base de données relationnelle et d'un dashboard.

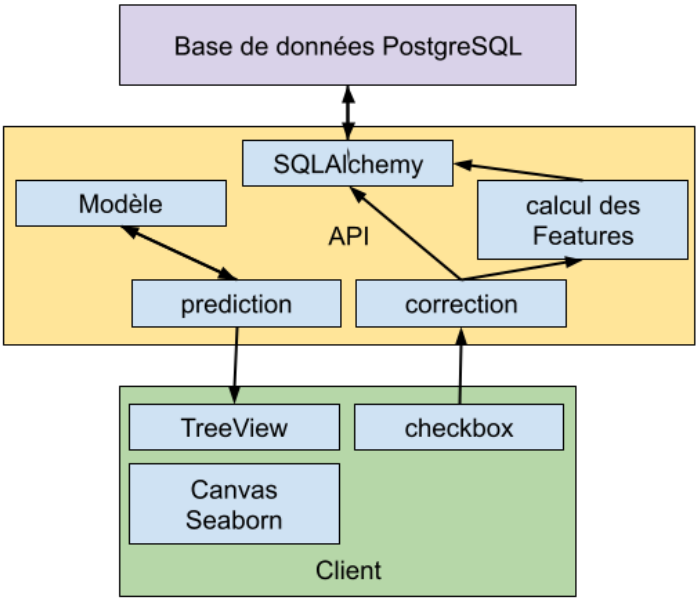
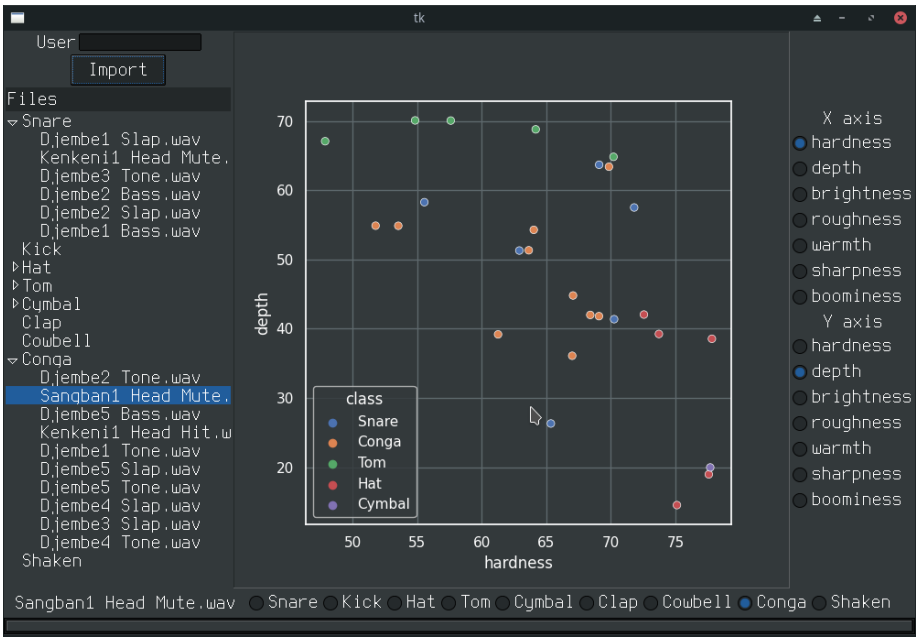


schéma fonctionnel

Les différents services ont été conteneurisés avec Docker.

Client

L' application client doit permettre l'organisation de samples de percussion selon leur classe. L'utilisateur peut corriger les classes prédites. Cette correction sera remontée via le webservice afin de compléter les données.

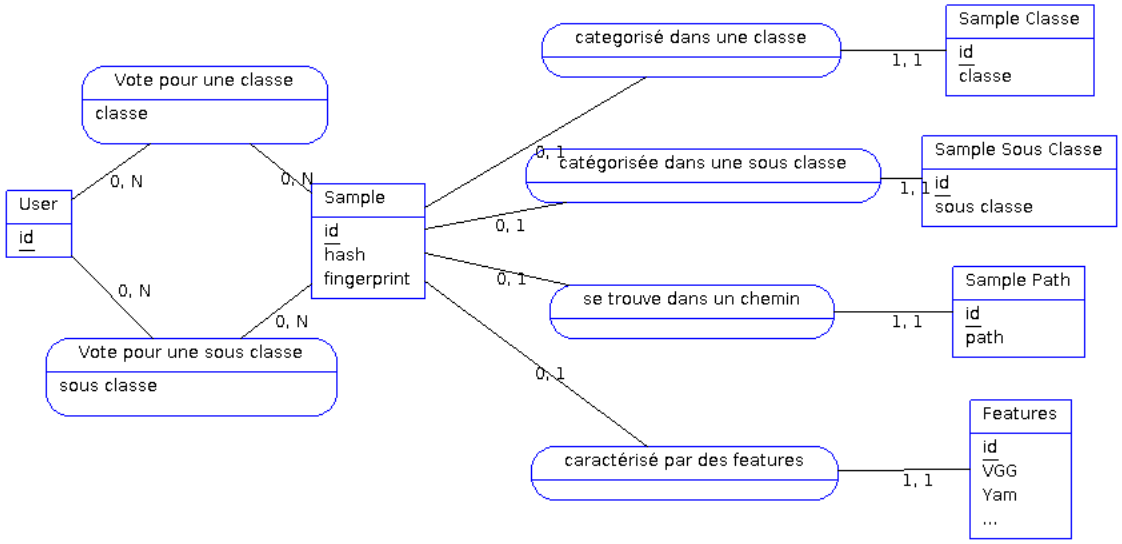


Client python/tkinter

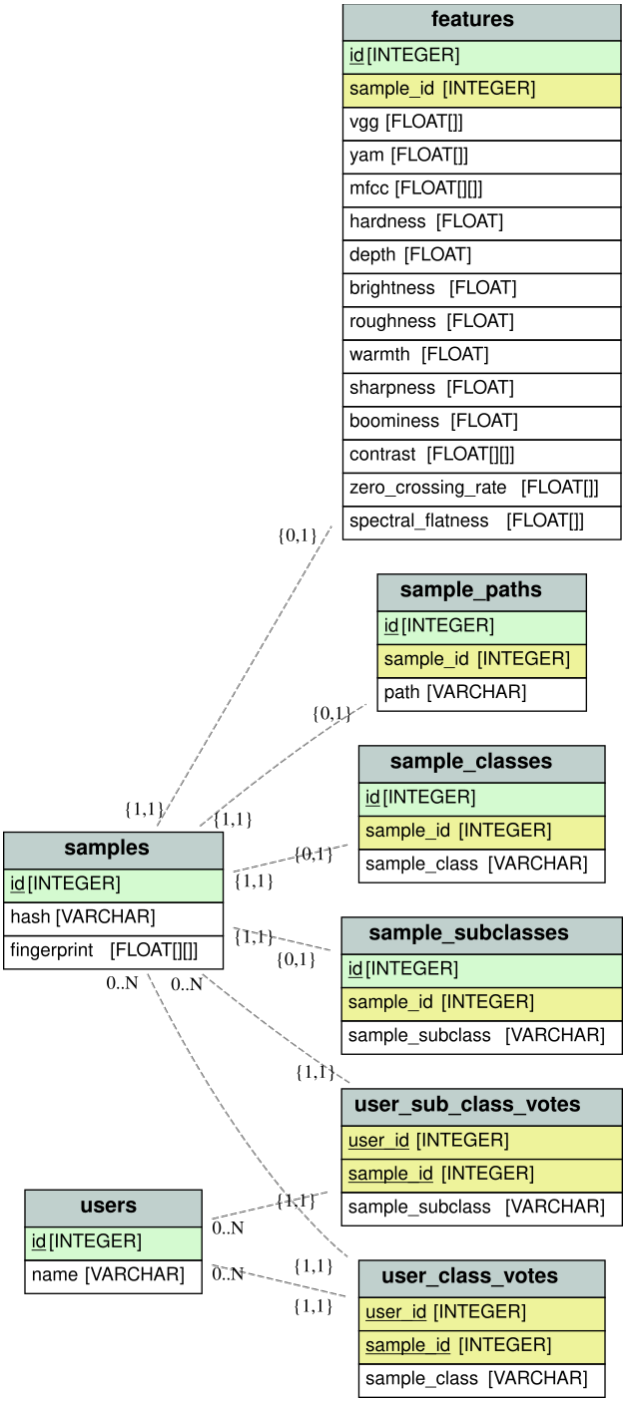
Base de données

Conception

Les features mélangent données scalaires et tabulaires. PostgreSQL a été choisi pour son support des Arrays.
La bibliothèque SQLAlchemy à été utilisée pour permettre une transformation automatique bi-directionnelle entre la base de données et des tableaux numpy.
On doit pouvoir stocker les features, les classes et les sous classes des samples.
La base de données doit permettre aux utilisateurs de soumettre des corrections. Plusieurs utilisateurs peuvent posséder les mêmes banques de son et soumettre une correction différente sur les mêmes samples.
Les tables sample_classes, sample_subclasses et sample_paths sont utilisées pour recueillir les informations des données initiales. Les tables user_class_votes et user_subclasse_votes sont utilisés pour recueillir les données soumises par les utilisateurs.
Pour identifier de manière unique les samples, une empreinte sonore est calculée. Sur cette empreinte on va ensuite calculer un hash. L'utilisation d'une empreinte permettra de rendre l'identification plus robuste aux légères transformations. Dans un premier temps, une MFCC basse résolution a été utilisée.



Modèle conceptuel de données



Modèle physique des données

Les relations '*N,N*' des votes utilisateurs sur les classes et sous classes ont été implémentées sous la forme de deux tables possédant une clé primaire composite (*user_id*, *sample_id*).

```
def addapt_numpy_float64(numpy_float64):
   return AsIs(numpy_float64)
def addapt_numpy_float32(numpy_float32):
   return AsIs(numpy_float32)
register_adapter(np.float64, addapt_numpy_float64)
register_adapter(np.float32, addapt_numpy_float32)
```

```
        return np.array(value)

    def __str__(self):
        return "FLOAT[][]"
```

Classe Array2D permettant la conversion de la base de donnée vers un tableau numpy et adaptateurs des types float numpy.

La base de données, ainsi que sa structure, sont mises en place par un script python qui utilise SQLALchemy.

```
metadata = MetaData()

sample_table = Table("samples", metadata,
       Column('id', Integer, primary_key=True),
       Column("hash", String, unique=True),
       Column("fingerprint", postgresql.ARRAY(Float(precision=2),
                                              dimensions=2)),
   )
...

def create_tables(engine, drop=False):
   if drop:
       classes_table.drop(engine, checkfirst=True)
       subclasses_table.drop(engine, checkfirst=True)
       path_table.drop(engine, checkfirst=True)
       features_table.drop(engine, checkfirst=True)
       sample_table.drop(engine, checkfirst=True)

   metadata.create_all(engine)
```

Script de création des tables

Backup

La programmation des sauvegardes est réalisée à l'aide d'un service ofelia. Une fois par semaine, le service exécutera la commande suivante sur le container postgres:

"sh -c 'FILE=/backup/$(date+\"%Y-%m-%d\").bak; pg_dumpall -U dev > $FILE'"

```
postgres:
  image: postgres
  volumes:
     - ./postgres:/data/postgres
     - ./db-backup:/backup
  ...
  labels:
    ofelia.enabled: "true"
    ofelia.job-exec.backup.schedule: "0 0 * * 7"
    ofelia.job-exec.backup.command: "sh -c 'FILE=/backup/$$(date +\"%Y-%m-
%d\").bak; pg_dumpall -U ${POSTGRES_USER:-dev} > $$FILE'"

ofelia:
```

```
  image: mcuadros/ofelia:latest
  depends_on:
    - postgres
  command: daemon --docker
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock:ro
```

Extrait du docker-compose.yml

Performances

```
query = (
    session.query(
        Sample.id,
        Features.hardness,
        Features.depth,
        Features.brightness,
        Features.roughness,
        Features.warmth,
        Features.sharpness,
        Features.boominess,
        Features.vgg,
        Features.yam,
        Features.contrast,
        Features.zero_crossing_rate,
        Features.spectral_flatness,
        SampleClass.sample_class,
        SampleSubClass.sample_subclass
    ).select_from(Sample)
    .join(Sample.features)
    .join(Sample.sample_class)
    .join(Sample.sample_subclass)
)
# data = pd.read_sql(query.statement, engine)
```

```
%time data = pd.read_sql(query.statement, engine)
```

```
CPU times: user 2.66 s, sys: 60.2 ms, total: 2.72 s
Wall time: 3.78 s
```

Le chargement des 15 features sur 8080 samples prend environ 4 secondes.

Service API

Le webservice proposant l'API utilise FastAPI. Il permet de mettre à disposition le modèle pour effectuer une prédiction.
Deux endpoints sont définis:

- Un service de prédiction qui reçoit un samples et qui retourne une prédiction ;
- Un service permettant la soumission de corrections qui reçoit un sample et une classe. Les features sont calculées et envoyées en base de données ainsi que la classe corrigée. Le fichier son en lui-même n'est pas stocké.

Le service est mis en place via un container.

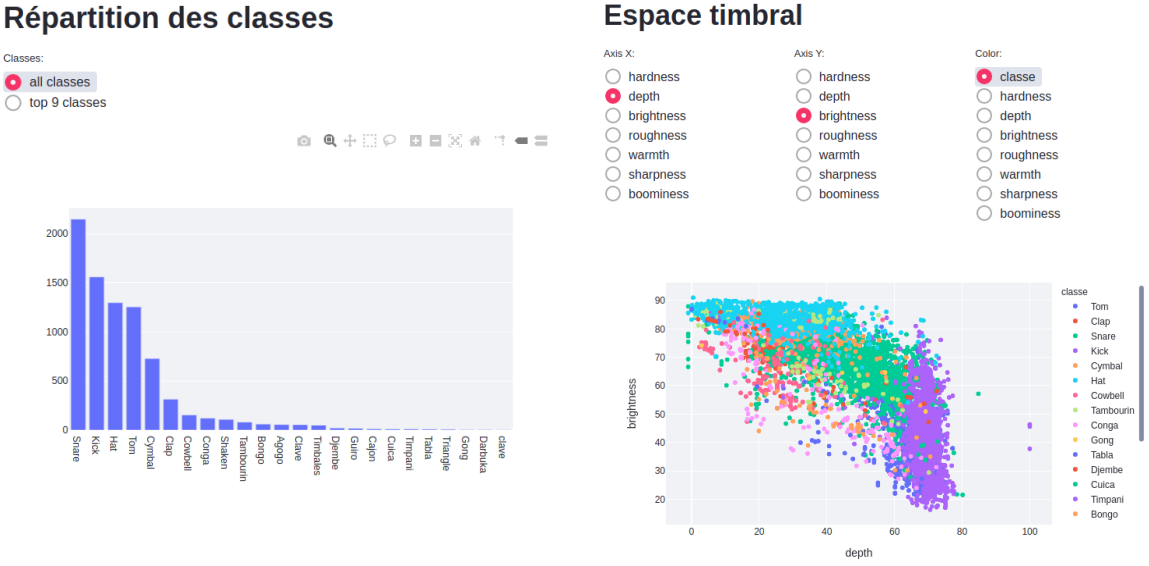
```python
@app.post("/predict")
async def predict(samplefile: UploadFile = File(...)):
  global model
  try:
      suffix = Path(samplefile.filename).suffix
      with NamedTemporaryFile(delete=False, suffix=suffix) as tmp:
              shutil.copyfileobj(samplefile.file, tmp)
              tmp_path = Path(tmp.name)
      vgg_emb = features.vggish_embedding(tmp_path)
      tmp_path.unlink()
      result = model.predict(vgg_emb.reshape(1,128))
      class_ = class_names[result.argmax()]
      return {
          "result": str(result),
          "class": class_,
          "classes": class_names
      }
  except Exception as e:
      return JSONResponse(
          status_code=400,
          content={"message": str(e)},
      )
```

Endpoint de prédiction

La détection du type de fichier à nécessité de passer par un fichier temporaire nommé.

Dashboard

Le dashboard permet de visualiser la répartition des classes et la répartition des samples dans l'espace de timbre. Il devra aussi permettre de visualiser les soumissions des utilisateurs.



Bilan

Axes d'amélioration

Afin d'améliorer les différents modèles, on devrait procéder à un tuning des hyper paramètres (Gridsearch). Des learning curves pourront être réalisées pour identifier les éventuels overfits. On aurait pu utiliser de la data augmentation. En revanche, le surcoût computationnel n'est pas négligeable et nécessiterait d'utiliser des ressources cloud comme des instances de calcul Azure.. Au niveau de l'application, on pourrait mettre en place un ré-entraînement automatique des modèles. Cela nécessite d'implémenter un modèle registry. Ce modèle registry devrait être capable de tenir compte des features utilisées en entrée de chaque modèle. Les performances de ces modèles pourront être monitorées et la sélection du meilleur modèle pourra être automatisée.

Conclusion

Ce projet a été l'occasion d'expérimenter des solutions pouvant répondre à un cas d'usage concret. Cela m'a donné l'occasion de faire le lien avec les compétences acquises au cours de mes précédentes expériences professionnelles. J'ai pu monter en compétences sur des outils comme SQLAlchemy que j' utilise aujourd'hui sur des projets en entreprise.