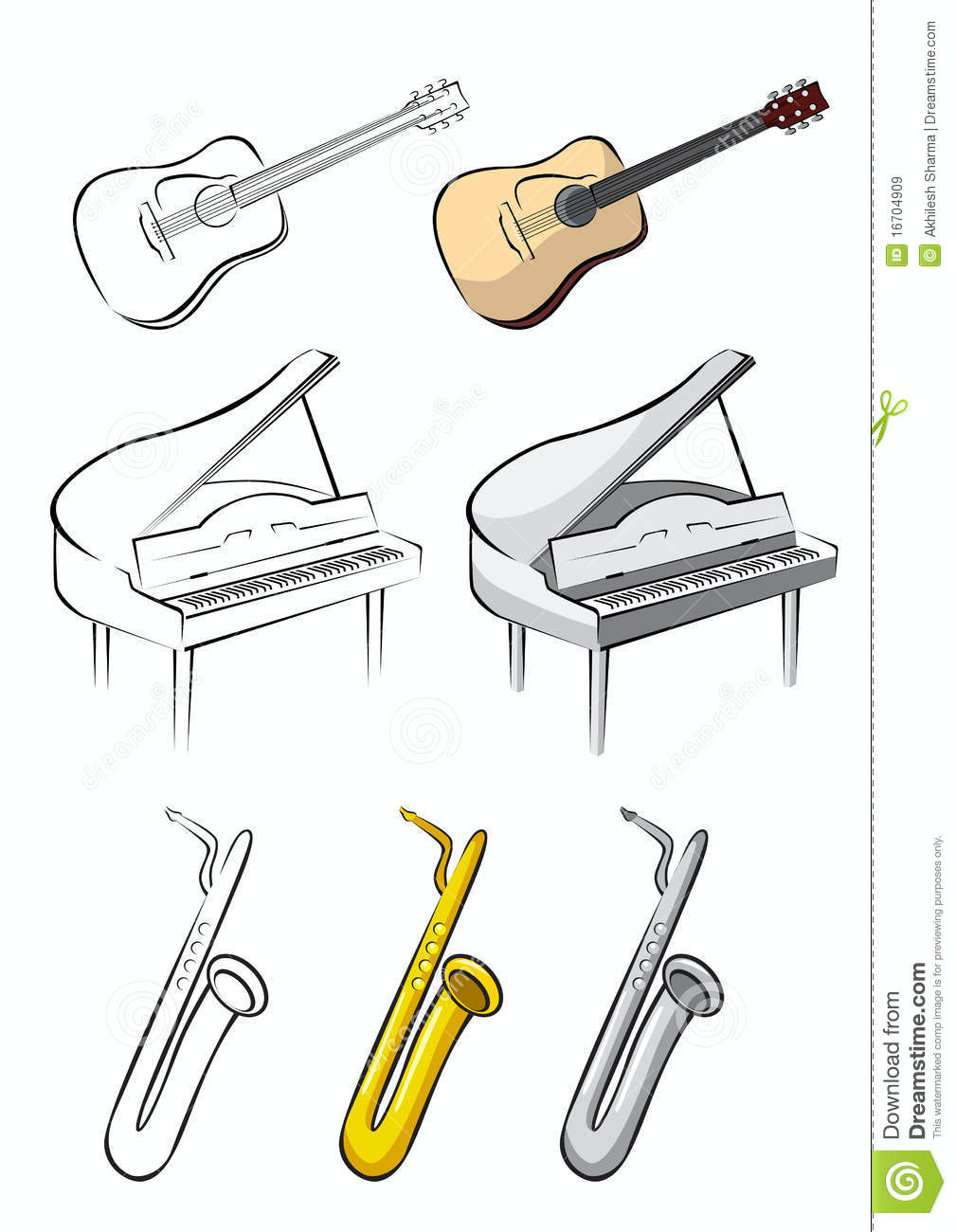
PROJET ANNUEL IABD

**Sujet** : Classification d’instruments (Piano, Saxophone, Guitare)



**Participants** : KHAOUANI Ahmed / RENOIR Marvin / CLODION Quentin

[**1. Introduction**](#_5bgpqnqr6j44) **2**

[**2. Construction du Dataset**](#_adkdf5mjheb1) **2**

[**3. Music Creator**](#_frbv92o2t4a1) **5**

[**4. Les différents modèles**](#_4ap9dcdygchs) **6**

[Modèle Linéaire](#_1wucmrmv2mno) 6

[**Perceptron Multicouche**](#_ctvgbxabav9b) **8**

[RBF Naïf](#_r0q9hnozmne0) 10

[SVM](#_taol345ijg2z) 11

[**5. Explorations et Résultats**](#_h7qletwbl0vl) **13**

[Fourier](#_6me4989yhx9k) 13

[Spectrogramme](#_h6x1tenohien) 13

[Spectrogramme (studio)](#_kxn1whmvsx7f) 13

[**6. Application**](#_xa57c5v8f90k) **14**

[**7. Axes d’amélioration**](#_akb1cbcu2am4) **15**

### 

### **1. Introduction**

L’objectif de ce projet est la classification de sons provenant de trois instruments, le piano, la guitare et le saxophone.

Ce qui a influencé notre choix des instruments c’est dans un premier temps la popularité des instruments, car le but est de récupérer suffisamment de données afin de constituer un Dataset suffisamment riche et diversifié. Puis dans un second temps, la différence à l’écoute entre ces instruments.

Il faut savoir que ce sont trois instruments de familles distinctes (léger rapprochement entre le piano et la guitare)

**Piano** : Famille des instruments à cordes

**Guitare** : Famille des instruments à cordes pincées

**Saxophone** : Instruments à vent de la famille des bois

### **2. Construction du Dataset**

Après avoir choisi nos trois instruments, il fallait trouver comment on allait constituer notre dataset. Nous avons donc décidé de récupérer un maximum de piste audio pour chacun des instruments et d’ensuite de réaliser des cuts de 2 secondes sur chacun des morceaux.

Dans un premier temps nous avons récupérés les notes des instruments en allant voir un ami qui possédait ces différents instruments.

Puis comme il nous fallait plus de données sur les différents types de guitare, piano et saxo et venant de micros différents, nous avions trois choix de source de données : Youtube, Spotify, Soundcloud. Nous nous sommes concentrés dans un premier temps sur SoundCloud avant de se rendre compte que de nombreuses pistes audio comprenaient des droits d’auteur dont il fallait s'acquitter. Quand à Spotify, il n’était pas possible de récupérer les morceaux via les API. Nous nous sommes donc tournés vers Youtube.

Afin de récupérer nos pistes audio nous avions eu l’idée de créer un scrapper en Python qui serait capable de récupérer les X premières vidéos d’une recherche Youtube. Ce procédé est intéressant mais il comporte deux problèmes majeurs.

Le premier étant son temps de réalisation, le second est que nous n’avons pas la visibilité de ce que le scrapper va récupérer. En tapant sur youtube “Piano” nous pouvons avoir comme résultat des tutos, des présentations de piano etc.. ce qui ne nous intéresse pas pour récupérer les données sur les instruments.

Il aurait donc fallu inspecter toutes les données récupérées et couper les parties du fichier de ne contenant pas l’instrument.

Nous avons donc décidé de récupérer à la main des vidéos Youtube qui nous semblaient les plus pertinentes. On pourra ainsi avoir des données de meilleurs qualités et identifier les parties ou l’instrument n’était pas jouer pour les supprimer. Pour récupérer les vidéos nous avons mis en place un système permettant d’avoir un Dataset diversifié.

* Pour chaques instruments, nous avons plusieurs sous-familles (piano à queue, piano droit…)
* Pour chaque sous-famille, il faut prendre en compte trois types d'enregistrements (studio, amateur, ou en plein air).
* Pour chaque type d'enregistrement de chaque sous-famille, récupérer trois vidéos d’au moins 5 minutes.

En respectant ce système de sélection, nous avons donc atteint un total de 81 vidéos. Après avoir récupérer toutes ces vidéos il a fallu faires un traitement car elles ne sont pas directement exploitables.

**1ère étape** : Identifier les passages de la vidéos à exclure (voix, générique, blanc...)

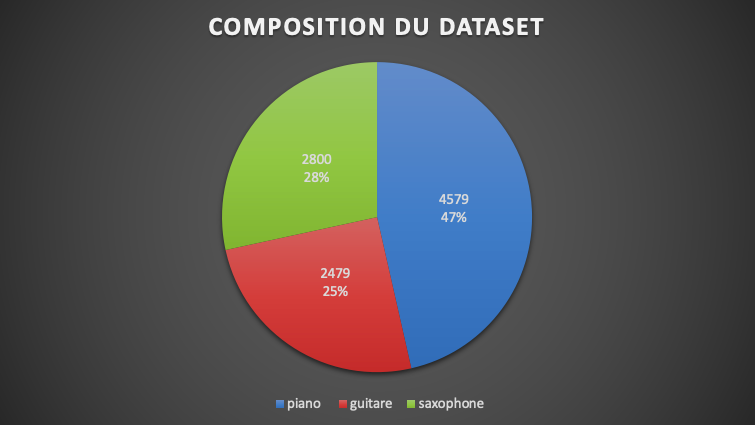
**2ème étape** : Passer les fichiers audio en .wav et en Mono pour les rendres lisibles et moins volumineux.

**3ème étape** : Réaliser des cuts de 2 secondes sur chaque vidéo. Pour réaliser ces cuts et enlever les parties indésirables on a donc construit une application permettant de faire différentes manipulations sur des fichiers audio sur de grandes quantités de données.

Afin d'accroître notre efficacité nous avons chacun choisi un instrument chacun :

* ***KHAOUANI Ahmed*** *: Piano*
* ***CLODION Quentin*** *: Guitare*
* ***RENOIR Marvin*** *: Saxophone*

Voici un schéma représentant notre Dataset composé d'extraits audio de 2 sec :



Nous avons à l’heure actuelle, 3 pistes pour la formation du dataset. Ces 3 pistes ont été implémentées pour que l’on puisse rapidement adapter le dataset en fonction de la piste la plus concluante.

1 - Utiliser les données des fichiers wav :

La première idée et la plus simple consiste à utiliser les données d’un fichier .wav de deux secondes en mono et avec une fréquence de 20000HZ pour diminuer la taille des données en entrées. On pense que cette solution n’est pas pertinente et ne permettra pas d’obtenir les meilleurs résultats possibles.

2 - Transformation de Fourier :

La seconde hypothèse consiste à utiliser la transformation rapide de Fourier pour récupérer l’amplitude associée à chaque fréquence de l’enregistrement. Cette technique devrait plutôt bien fonctionner car la quantité des données utilisées en entrée du modèle seront bien moins astronomiques.

3- Spectrogramme :

Enfin la dernière idée consistait à utiliser le spectrogramme qui comporte les mêmes données que le spectre de Fourier mais cette fois-ci, en fonction du temps. Aujourd’hui, on ne pense pas que cette donnée (le temps) sera vitale. De plus, cela ajouterait un paramètre à prendre en compte ce qui allongerait le temps de calcul.

Ces 3 idées peuvent aussi être traiter comme de la classification d’image en prenant respectivement, le spectre d’un fichier audio, le spectre de Fourier du morceau et enfin, le spectrogramme. Cela ne nous semble pas particulièrement être une bonne idée pour les deux premiers datasets.

### **3. Music Creator**

Cette application a évolué au fur et à mesure du projet pour pouvoir permettre la modification en masse de fichiers pour constituer le dataset.

Au départ, il s’agissait d’une application console en C# qui permettait de comprendre le mécanisme des fichiers .wav en proposant la lecture et la réécriture de fichiers sans s’appuyer sur des frameworks ou librairies existants.

Par la suite, on a commencé à ajouter plusieurs opérations sur cette application tel que splitter les canaux d’un fichier wav, le découper en plusieurs fichiers d’une certaine durée, ou superposer des morceaux sur un même fichier.

Cette application nous a donc permis dans un premier temps, de mieux comprendre le fonctionnement d’un fichier .wav et la manipulation de spectres audio pour par la suite, pouvoir les manipuler plus facilement si besoin. Etant donné que l’application commençait à avoir de nombreuses fonctionnalités, on a décidé de l’adapter en Winform pour pouvoir l’utiliser plus simplement pour la construction du dataset.

Aujourd’hui on l’utilise donc pour écrire le contenu des fichiers .wav dans des fichiers csv, scinder les canaux, changer la fréquence d’un morceau (avecFFmpeg) ou bien couper un morceau en différents morceaux d’un durée donnée. et analyser le spectre Fourier.

Pour la partie création de dataset en CSV avec Fourier et de fichiers .PNG pour spectrogrammes, on a utilisé du python afin de convertir les fichiers traités précédemment avec Music Creator au format souhaité.

Voici ci-dessous, la page d’accueil permettant d’ajouter des fichiers et de pratiquer des opérations variées sur ces dernières

### 

Ici se trouve la partie permettant de visualiser le morceau sous la forme d’un spectre audio et sa transformation de Fourier :

### 

### **4. Les différents modèles**

#### **Modèle Linéaire**

Régression Linéaire

La régression linéaire est une modélisation linéaire qui va permettre de prédire la valeur Y en fonction d’un paramètre X, en s’appuyant sur les différentes données pour connaître la fonction affine de la forme :

y = ax + b.

On utilisera donc la formule suivante pour trouver les arguments a et b (respectivement W[0] et W[1])

****

Nous n’avons pas rencontrés de problèmes particulier en implémentant les modèles linéaires. Voici le résultat d’un Test Case avec la librairie en Rust :

### 

Classification linéaire (de Rosenblatt)

La classification va affecter un input à un label en jouant sur le poids des neurones.

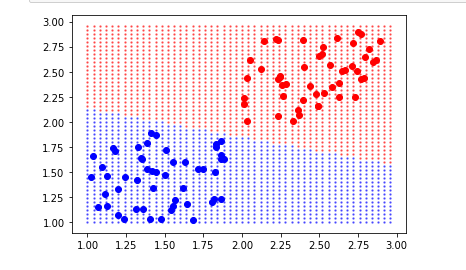
La règle de Rosenblatt *(formule ci-dessous)* va mettre à jour le poids d’un neurone en fonction de la sortie assignée à la donnée

****

avec :

* - W : le poids du neurone
* - α : le pas d’apprentissage
* - X : la valeur d’entrée (les pixels)
* - Y : la valeur de sortie attendu (1 ou -1)
* - g(X) : valeur calculé par le perceptron

Test Case en Rust :

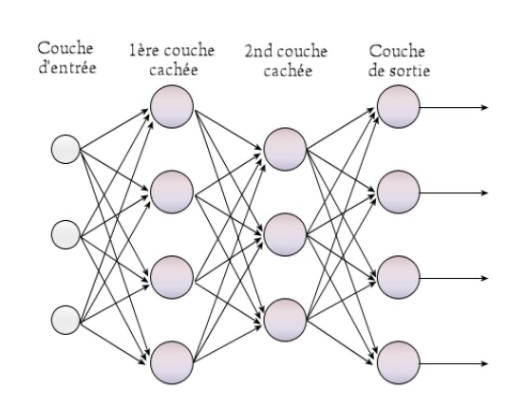


#### **Perceptron Multicouche**

Perceptron multicouche est un réseau de neurones organisé en plusieurs couches : une couche d’entrée, N couches cachées et une couche de sortie.

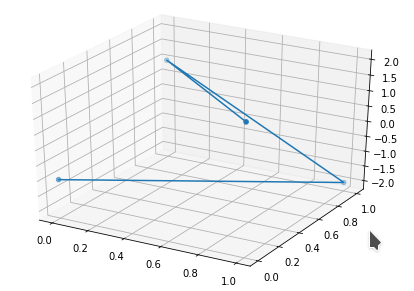
L’image ci-dessous est un MLP *(Multi Layers Perceptron)* à 4 couches avec une couche d’entrée à 3 neurones, deux couches cachées constituées respectivement de 4 neurones et 3 neurones et une couche de sortie de 4 neurones. Dans cet exemple, avec un MLP bien entraîné, on pourra alors prédire en sortie 4 possibilités (​*Exemple* : donner une couleur en entrée et prédire si la couleur sera Rouge, Jaune, Bleu ou Verte)

Pour une tâche de classification on utilisera une fonction d’activation tanh, tandis qu’on utilisera une fonction linéaire pour la régression.

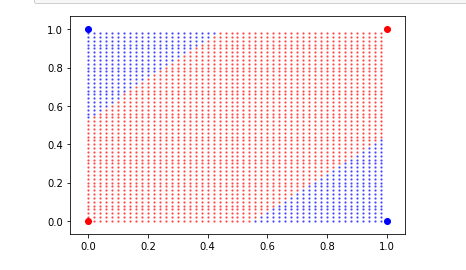
****

L’implémentation du perceptron multicouche a été plus compliquée que celle des modèles linéaires. Il nous a fallu repasser de nombreuses fois sur le code pour pouvoir corriger les problèmes qui étaient rencontrées. On a eu aussi quelques problèmes concernant la communication avec python tel que le typage et les Vector que nous avons dû remplacer par des tableaux à une dimension.

Régression

****

Classification

****

#### 

#### **RBF Naïf**

Le calcul du modèle RBF (Radial Basis Function) Naïf est une combinaison linéaire ou le signe des fonctions gaussiennes :

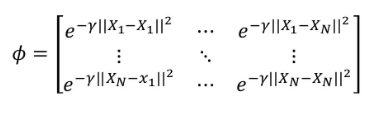
régression classification

### 

Elle se fait par le biais de la formule :



Avec φ, une matrice carré de taille Xn :



L’implémentation en elle même n’a pas été très compliquée, mais un problème que nous n’avions pas réussi à identifier empêchait le modèle de prédire correctement les résultats des tests case.

En essayant le modèle sur différents tests cases, nous avons pu voir que le choix de gamma impacte grandement sur les résultats de notre modèle.

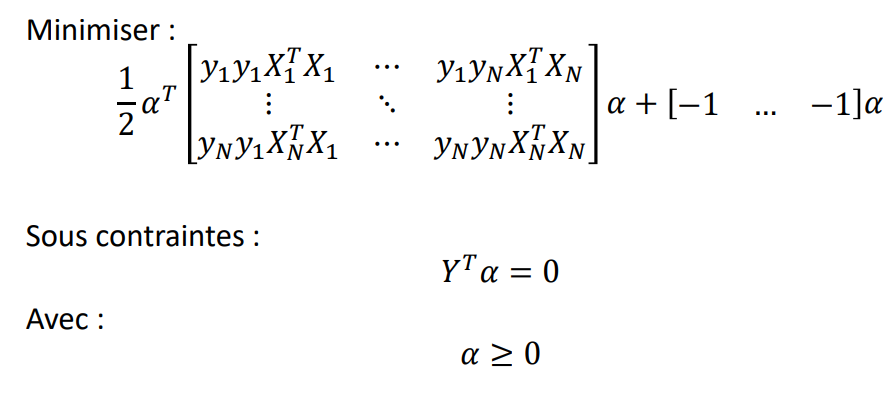
Ci-dessous un test case réalisé avec un modèle RBF de notre librairie :

#### 

#### **SVM**

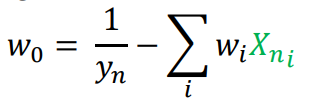
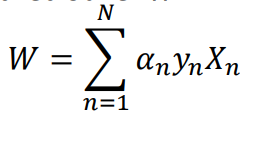
Le SVM avec un kernel basique permet de séparer linéairement les données en trouvant la position de la droite permettant de maximiser le plus possible, la marge qui la sépare des données *(cela se fait grâce à l’optimisation quadratique)*.

Le problème est posé sous la forme suivante :



Avec X pour les données, Y pour ses labels respectifs et alpha qui est la valeur que l’on va chercher à trouver. Il est possible d’utiliser ici un kernel trick pour modifier les paramètres de l’optimisateur quadratique.

Une fois le vecteur alpha récupéré, on peut alors calculer les poids W et la valeur du biais à partir d’un vecteur support avec les formules ci-dessous :

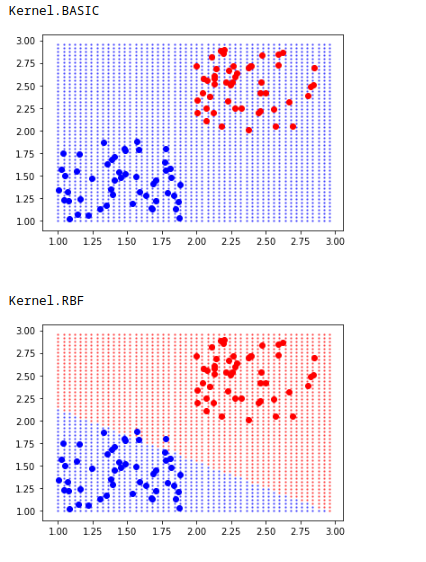


Nous avons le code du SVM, mais en voyant les résultats sur les tests cases, nous savons que ce dernier ne fonctionne pas correctement.

L’implémentation du SVM nous a confrontés à de nombreux problèmes. Dans un premier temps, la compréhension du modèle, l’utilisation d’un solver quadratic et le rôle des kernels ont été difficiles à comprendre. Ensuite, le code a été fait assez rapidement avec la possibilité d’utiliser plusieurs kernels comme le quadratique, le rbf et le kernel de base.

Cependant lors de l’utilisation de notre librairie sur les tests case, nous avons constaté des problèmes d'exécutions. En debugant nous avons vu que les bugs étaient dû à une mauvaise initialisation des paramètres pour la fonction d’optimisation quadratique. Après quelques jours à essayer de trouver la solution, nous avons décidé d’adapter le code pour réaliser l’optimisation quadratique en python. Cela a été plus simple car les paramètres demandés étaient plus approprié à notre problème.

Malgré cela, les prédictions sont toujours mauvaises sur les tests case comme on peut la voir ci-dessous :



Nous n’avons pas pu identifier clairement si le problème venait des poids du modèle, du résultat de la fonction d’optimisation quadratique ou de la prédiction.

### 

### 

### **5. Explorations et Résultats**

#### **Fourier**

Le dataset a été construit en utilisant le module python scipy.fftpack sur l’ensemble du dataset déjà prédécoupé et splité. Pour organiser nos données, on avait le choix entre tracer un graphique avec matplotlib et l’enregistrer au format .png ou bien récupérer les amplitudes correspondant aux fréquences et les écrire dans un fichier .csv.

On a fait le choix d’utiliser le format .csv en pensant que les données seraient plus précises et donc plus adaptés pour le modèle. Etant donné que chaque morceau était ramené sur 18 000Hz, que la précision était de 0.5Hz et qu’il faut prendre la fréquence et l'amplitude, la taille d’un paramètre est donc la suivante :

18 000 \* 2 / 0.5 = 72 000

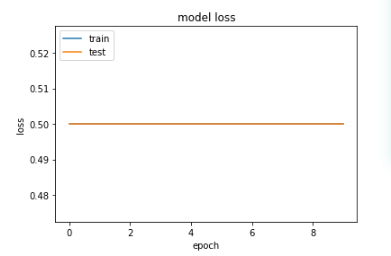
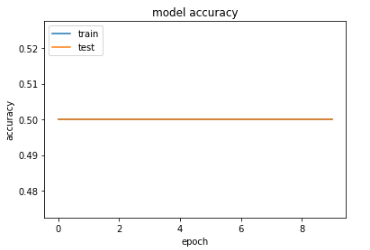
Ces données étant beaucoup trop volumineuses, nous avons décidé de récupérer la partie la plus intéressante de la fréquence et de diminuer la précision. La taille d’un paramètre est alors devenu la suivante :

8000 \* 2 / 1 = 16 000

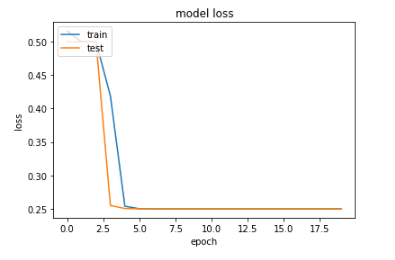
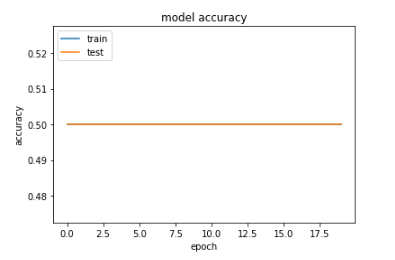
Voyant que de hautes valeurs pouvaient poser problème au modèle et qu’il fallait lui donner des valeurs entre 0 et 1, nous avons aussi diviser l’ensemble des valeurs pour qu’elles soient entre 0 et 1.

Résultats obtenu sur Keras :

Linéaire (guitare/saxo)



MLP (guitare/saxo)



On peut ici se douter du fait que le modèle n’arrive pas à apprendre et pour garder un score de prédiction correct, le modèle renvoi toujours la même classe.

Résultats obtenu sur la librairie Rust

Linéaire (guitare/saxo)

Predict validation -> -1.0 : 48 of 150 / 1.0 : 130 of 150

Predict train -> -1.0 : 265 of 800 / 1.0 : 715 of 800

ou

Predict validation -> -1.0 : 150 of 150 / 1.0 : 0 of 150

Predict train -> -1.0 : 800 of 800 / 1.0 : 0 of 800

Comme on peut le voir ci-dessus, notre modèle linéaire produit de meilleurs résultats avec un score de 61% de réussite avec les données d’entraînement et de 59% avec les données de validation. Nous avons essayé de reproduire ce résultat avec Keras mais sans réussite.

MLP (guitare/saxo)

predict -1 -> 340 sur 340

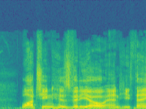
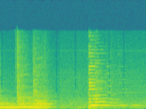
predict 1 -> 0 sur 334

Les résultats sont les mêmes avec 3 classes, en effet le modèle retourne toujours une seule des 3 classes. Le taux de bonnes prédictions est donc de 33% (sur un dataset équilibré)

On pense que les données étaient trop volumineuses pour être correctement traitées par nos modèles. Une solution pourrait être, de garder seulement les pics de fréquence au lieu de garder toutes les valeurs. Mais cela peut poser un problème car la taille de nos données ne sera pas homogène, pour contrer à cela il faudrait garder les X fréquences et amplitudes les plus élevées.

#### **Spectrogramme**

Les spectrogrammes ont été construits en appliquant la fonction specgram() du module matplotlib.pyplot aux fichiers .wav. En vérifiant le dataset, on a pu remarquer que certaines photos étaient justes blanches, on a alors fait un tri sur tout le dataset pour supprimer ces photos.

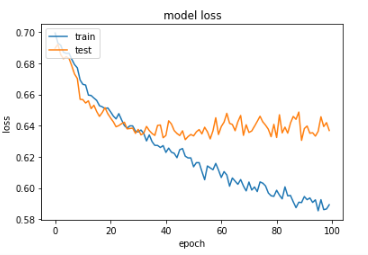
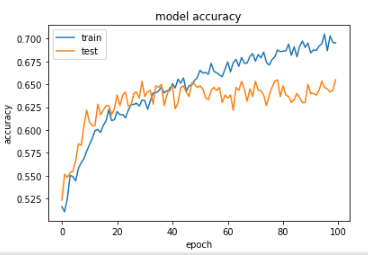


Après plusieurs tests, on a pu voir que le modèle avait des statistiques similaires en travaillant les spectrogrammes en 32x32 ou en 64x64.

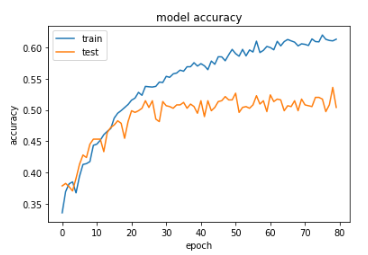
Nous avons donc choisi d’utiliser les images en 32x32 en rgb et avec des valeurs comprises entre 0 et 1 au lieu de 0 à 255.

Résultats obtenu sur Keras :

MLP (piano/saxo)

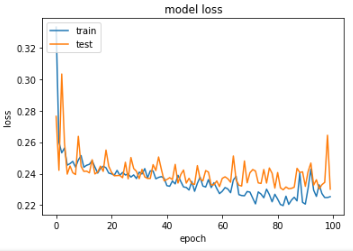
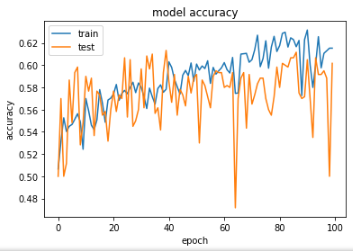


MLP (piano / saxo / guitare)



On peut constater que le modèle fait rapidement face à du sur-apprentissage. Que ce soit avec 2 ou 3 classes.

MLP (piano/saxo avec Dense Layer)



Résultats obtenu avec la librairie Rust :

MLP (piano /saxo)

Test :

Predict of -1.0 : 300 of 300

Predict of 1.0 : 0 of 300

En partant sur les mêmes couches cachées qu’avec Keras, on peut voir que notre modèle a de moins bons résultats. En effet, le notre se contente de prédire toujours la même classe. Tandis que le modèle fait avec Keras même s’il est peu performant, arrive à faire des prédictions non aléatoires.

RBF (piano /saxo avec gamma = 1000)

Test :

Predict of -1.0 : 181 of 300

Predict of 1.0 : 139 of 300

Train :

Predict of -1.0 : 895 of 1500

Predict of 1.0 : 635 of 1500

guitare / saxo

Test :

Predict of -1.0 : 163 of 300

Predict of 1.0 : 143 of 300

Train :

Predict of -1.0 : 870 of 1500

Predict of 1.0 : 595 of 1500

guitare / piano

Test :

Predict of -1.0 : 184 of 300

Predict of 1.0 : 128 of 300

Train :

Predict of -1.0 : 797 of 1500

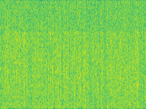
Predict of 1.0 : 714 of 1500

Voici le modèle que nous avons utilisé en production. Etant donné que le MLP retourne toujours la même sortie avec 2 ou 3 classes, ce modèle était le plus intéressant avec la classification linéaire sur le dataset de Fourier. Même s’il a un taux de prédiction situé entre 50% et 60% entre chaque classe et que cela n’est pas suffisant, il a l’avantage d’essayer de prédire et de ne pas retourner la même valeur. En plus de cela, le fait qu’il utilise les spectrogrammes rend l'exécution plus rapide

**Spectrogramme (studio)**

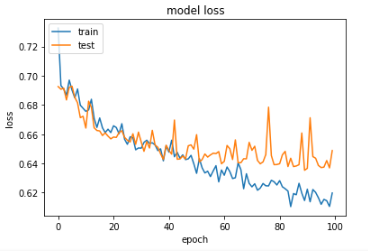
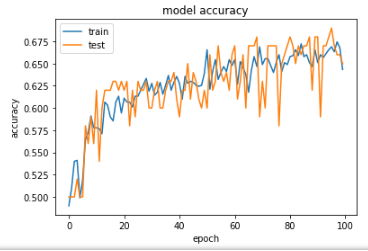
Nous avons pensé que les résultats médiocres pouvaient être dûes à la différence de qualité entre les vidéos récupérées. Il a donc été décidé de construire un dataset moins fourni que celui-ci mais, qui serait seulement composé des vidéos avec un son de bonne qualité. Cela pourrait nous permettre de déterminer si c’était bien ce facteur qui était à l’origine des mauvaises prédictions.

On a donc construit un dataset avec 500 enregistrements par classe. En utilisant les mêmes modèles que ceux pour le spectrogramme, les résultats ont été équivalents. Un petit tri a ensuite été effectuée sur le dataset pour enlever les images qui pouvaient comporter des anomalies comme celle ci-dessous :

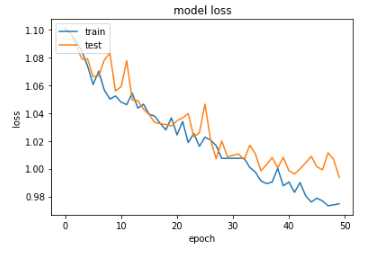
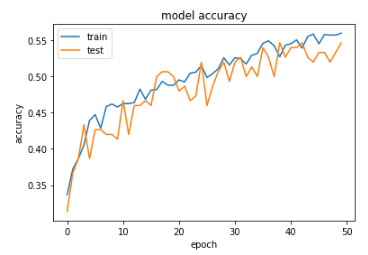


Résultats obtenu sur Keras :

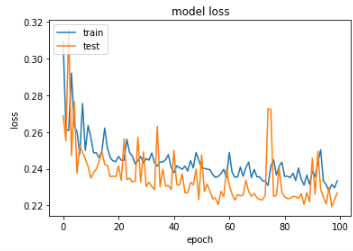
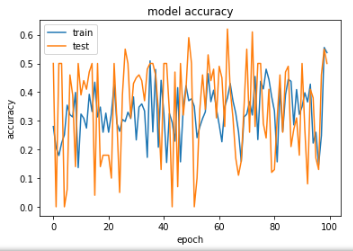
MLP (piano/saxo)



MLP (piano / saxo / guitare)



MLP (piano/saxo avec Dense Layer)



Résultats obtenu avec la librairie Rust :

MLP (piano /saxo)

Test :

Predict of -1.0 : 50 of 50

Predict of 1.0 : 0 of 50

Train :

Predict of -1.0 : 450 of 450

Predict of 1.0 : 0 of 450

RBF (piano /saxo)

Test :

Predict of -1.0 : 25 of 50

Predict of 1.0 : 32 of 50

Train :

Predict of -1.0 : 145 of 450

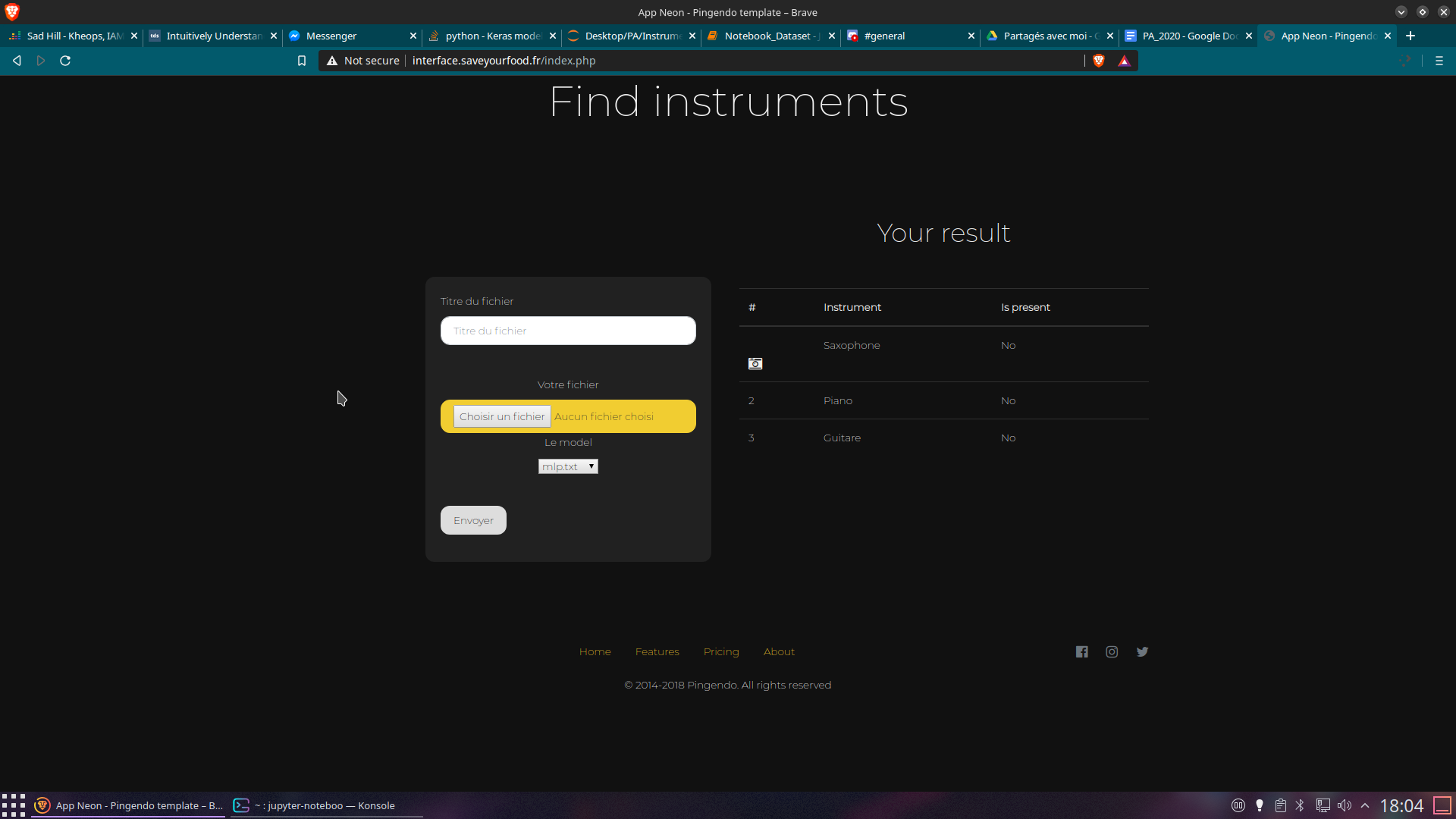
Predict of 1.0 : 316 of 450

Comme on peut le voir, les résultats ne sont pas forcéments plus élevées que sur le dataset précédent. Cependant, ici on ne voit pas de trace de sur-apprentissage, on pense donc que ce faible taux de prédiction est dû à un manque de données.

### 

### **6. Application**

L’application web est visible à l’adresse suivante : <http://interface.saveyourfood.fr/index.php>



Le but est de ce site est de lui donner un fichier audio (.mp3 ou .wav) et un type de modèle sur lequel exécuter la prédiction. Le site nous dit alors si le morceau contient une guitare, un piano ou un saxophone.

Cette application fonctionne avec du php pour le fonctionnement du site, du python pour adapter le fichier reçu et prédire à l’aide de la librairie en Rust.

Lorsque le fichier est donné à l’application, il est automatiquement transformé en fichier .wav puis téléchargé sur le serveur. Un script python est ensuite appelé avec en arguments, le chemin du fichier à traiter, le modèle à exécuter et son type (linéaire, mlp).

Le script va alors couper le fichier en plusieurs spectrogrammes (chacun équivalent à 2 secondes du morceau), et prédire le résultat de l’ensemble de ces spectrogrammes.

Le script python va ensuite retourner :

nombre de prédictions / total des prédictions (et ce pour chaque instrument)

### **7. Axes d’amélioration**

Partie Dataset :

Les données récoltés étaient très variées et on pense que cela a rendu la prédiction compliqué. Il aurait été plus sage de se concentrer sur une sous-catégorie par instrument et dans un seul milieu.

Partie Rust :

Il aurait été intéressant de voir le SVM fonctionner ainsi que ses résultats sur notre dataset.

Partie applicative :

Etant donné que l’utilisateur peut mettre n’importe quel fichier audio, il aurait été intéressant de rajouter une prédiction pour un instrument non reconnu. En effet, aujourd’hui si un utilisateur rentre un extrait de 8 secondes avec 6 secondes où il parle et 2 secondes où il joue de la guitare le modèle va être induit en erreur et va essayer de prédire une guitare, un piano ou un saxophone sur les 6 secondes ou l’utilisateur parle.

Imaginons maintenant que nous avons un modèle fiable à 100%, mais 4 secondes de l’extrait ou l’utilisateur parle ont été détectés comme étant du piano. Alors le résultat renvoyé va être le piano, ce qui ne correspond pas au résultat souhaité.