1 Ingénierie des features

L'ingénierie des features est l'étape du pipeline qui va permettre de se ramener à un ensemble de features optimale pour l'algorithme sélectionné. En effet, la performance d'un algorithme est aussi bien dépendante de la valeurs de ses hyperparamètres que de l'ensemble des features sur lequel il est évalué. L'ingénierie des features comprends différentes étapes selon la natures des features.
Un cycle typique d'ingénierie de features va comprendre une itération des étapes suivantes : Définition d'un ensemble de features, expérimentation et analyse de résultats sur un ensemble de validation, modification de l'ensemble de features. La définition d'un ensemble de features va passer par différentes étapes d'agrémentation de l'ensemble. Ainsi formulé, une approche simple est de scinder le problème en deux parties. La première consiste à générer des features, on peut alors passer par un processus itératif simple sur l'ensemble des features en utilisant un ensemble d'opérateurs. La deuxième partie du problème consiste, quant à elle, à choisir le sous ensemble de features le plus performant en fonction de l'algorithme choisi.

Formellement, le problème revient à partir d'un ensemble de features de départ $F_{initial}$ à générer un autre ensemble de features $F_{generated}$ et à partir de l'union de ces deux ensembles, générer deux autres ensembles avec les valeurs normalisées $F_{normalized}$ et standardisées $F_{standardized}$ (on a donc $|F_{initial} \cup F_{generated}| = |F_{normalized}| = |F_{standardized}|$).
Selon le budget, on pourra choisir de générer et considérer $F_{normalized}$ et $F_{standardized}$, ou uniquement l'un des deux ensembles.
On note l'ensemble des features possibles $F_{space} = F_{initial} \cup F_{generated} \cup F_{normalized} \cup F_{standardized}$

On considère ensuite un algorithme A avec un vecteur d'hyperparamètres fixé v et une métrique L, le problème revient à trouver $F^* \subset F_{space}$ tel qu'il minimise L évaluée avec l'algorithme A et la configuration v. On doit ainsi déterminer :

$$F^* = \underset{\mathrm{F} \in \mathrm{F}_{\mathrm{space}}}{argmin} L(A_v, F_{train}, F_{test}) \tag{1}$$

Les étapes de prétraitement présentées précédemment serviront à agrémenter l'ensemble $F_{generated}$. Des approches basées sur le representation learning ont également été étudiées mais elles ne permettent pas de conserver l'explicabilité du modèle

1.1 Génération de features

La génération de features consiste à découler des features qui sont initialement présentes dans le jeu de données d'autres features plus adaptées pour l'algorithme utilisé. Ces étapes permettront donc d'ajouter des feature à l'ensemble $F_{generated}$. On va dans un premier temps définir les opérations que l'on propose et qui seront appliquées à chaques features de l'ensemble de départ.

1.1.1 Les opérateurs

Les opérateurs choisis reprennent des techniques de transformation de features présentes dans la librairie MLLib et des implémentations d'opérations simples. On peut diviser les opérateur en 3 genres : les opérateurs unaires, les opérateurs binaires et les opérateurs group-by-then.

- Les opérateurs unaires reprennent les transformations suivantes :
  - ◇ La discrétisation : cette transformation va prendre une feature numérique continue et la convertir en une feature discrète.
    Plusieurs algorithmes d'apprentissage automatique fonctionnent mieux quand les features numériques avec des distributions de probabilité non-standards sont rendues discrètes. La discrétisation appliquée est la discrétisation par quantile. Elle va construire des classes

ayant le même nombres d'observations. L'étendue des groupes est choisi en utilisant un algorithme d'approximation des quantiles.

◇ L'expansion polynomiale : cette transformation va prendre une feature numérique et l'élever à une puissance n. On fixe dans l'outil n à 2.

◇ Les transformations de prétraitement : On pourra aussi considérer comme opérations unaires, les transformations appliquées dans les étapes de prétrataitement qu'on a présenté précédemment.

- Les opérateurs binaires reprennent les opérateurs arithmétiques classiques que sont l'addition, la soustraction, la multiplication et la division. L'addition et la multiplication étant commutative, on évite les doublons en ne générant qu'une fois une feature pour une paire rencontrée deux fois.

- Les opérateurs Group-by-then sont des opérateurs qui sont appliqués sur une paire de features. On a implémenté 4 opérateurs de ce genre, le GroupByThenMin, le GroupByThenMax, le GroupByThenAvg et le GroupByThenCount. Ces opérateurs vont grouper des exemples sur des features catégorielles ou numériques puis appliquer une opération sur une feature numérique. Leur utilisation est illustrée dans le tableau 1.

Airline	Distance	GroupedDistance	GroupByThenAvg
AF	3000	[3000,7000]	5000
AF	7000	[3000,7000]	5000
BX	2222	[2222]	2222
UJ	4500	[4500,500]	2500
LU	545	[545]	545
UJ	500	[4500,500]	2500

Table 1: Un exemple de GroupByThenAvg sur une feature.

Pour conserver un espace de features d'une taille raisonnable, on appliquera toutes les transformations unaires, cependant, en fonction du budget alloué par l'utilisateur on appliquera soit une transformation binaire, soit une transformation GroupByThen sur des paires de features sélectionnées au hasard.

De même, pour ne pas générer une taille d'espace de features trop important. On considère que l'on n'appliquera que les opérateurs aux features de l'ensemble de départ $F_{initial}$

Une fois ces transformations appliquées, on obtient l'espace de features $F_{generated}$, on peut alors appliquer les opérations de feature scaling (normalisation et standardisation) pour obtenir les espaces $F_{normalized}$ et $F_{standardized}$.

En pseudo-code, la procédure pour la génération des features est ainsi :

Algorithm 1 Génération de features

Input : Un jeu de données $D_{F_{initial}}$ avec ses features dans l'espace $F_{initial}$.
Output :

- ◇ Un jeu de donnée avec des features ajoutées : $F_{generated}$.
- ◇ Un jeu de donnée avec les features ajoutées, puis normalisées : $F_{normalized}$.
- ◇ Un jeu de donnée avec les features ajoutées, puis standardisées : $F_{standardized}$.

procedure GENERATEFEATURESSET($D_{F_{initial}}$)
  $F_{generated} \leftarrow F_{initial}$
  for f in $F_{initial}$ **do**
    if f *hasMissingValue* **then**
      $F_{generated} \leftarrow F_{generated} \cup$ *impute*(f) ▷ *impute*, as the other functions, return f with all strategies
      Withdraw f from $F_{generated}$
    end if
    $F_{generated} \leftarrow F_{generated} \cup$ *discretize*(f)
    $F_{generated} \leftarrow F_{generated} \cup$ *polynomiallyExpand*(f)
  end for
  while *generationBudget* ≥ 0 **do**
    Randomly pick f_1 and f_2 from $F_{initial}$
    Randomly apply *groupByThen*(f_1,f_2) or *applyBinary*(f_1,f_2)
    Decrease *generationBudget*
  end while
  $F_{normalized} \leftarrow normalize(F_{generated})$
  $F_{standardized} \leftarrow standardize(F_{generated})$
end procedure

1.2 Sélection de features

Une fois l'espace de features F_{space} généré, il faut à présent passer par une étape de sélection. Le postulat à la base de la sélection de feature est le fait que l'on puisse identifier, à partir des features que l'on a, celles qui sont redondantes ou non porteuses d'informations. D'une manière générale, le problème est généralement subdivivisé en 2 composantes : une composante de recherche qui génère un sous-ensemble de features $F_{subset} \subset F_{space}$ et une composante d'évaluation qui évalue le sous-ensemble généré.

Différentes méthodes d'évaluation existent dont plusieurs basées sur une analyse statistique. Bien que ces méthodes puissent être utilisées, il a été prouvé qu'elles ne sont pas les plus efficaces. Ces méthodes d'évaluation basées sur des analyses statistiques sont indépendantes des algorithmes d'apprentissage utilisées par la suite. Ce sont des méthodes dites à base de filtres. Une autre catégorie de méthodes d'évaluation existent, les méthodes ditent d'emballage. Ces méthodes sont dépendantes de l'algorithme d'apprentissage qui sera utilisé par la suite. Il existe également des méthodes hybrides qui combinent le filtrage et l'emballage pour d'abord attribuer un classement aux features puis ensuite l'utiliser pour évaluer la sélection sur l'algorithme d'apprentissage. La composante de sélection peut être classée en 3 catégories : exhaustive, avec des heuristiques ou avec des métaheuristiques. Une approche exhaustive est très couteuse car pour un ensemble de n features, elle revient à essayer 2^n sous-ensembles de features. Les approches par heuristiques classiques sont souvent les algorithmes de recherches best-first et les méthodes d'escalades. Cependant, ces approches rencontrent des défauts comme respectivement le temps de calcul et les optima locaux. Les approches par métaheuristiques combinent l'exploitation de solution avec l'exploration d'autres solutions pour permettre d'atteindre des optima globaux. Des métaheuristiques généralement cités pour le problème de la sélection de features sont les algorithmes génétiques. Ces approches requièrent cependant un nombre important d'hyperparamètres (la taille de

la population, le nombre de générations, etc) rendant la recherche de configuration complexe en plus d'avoir un résolution couteuse. Des approches par des méthodes de Monte-Carlo ont été également développée pour répondre au problème de sélection de features. Les méthodes de Monte-Carlo sont des méthodes qui reposent sur des simulations aléatoires pour la sélection. Des approches par recherche arborescente Monte-Carlo ont ainsi été développées [CL18] faisant usage de méthodes hybrides.
Nous allons consacrer notre approche aux méthodes de Monte-Carlo pour résoudre le problème de sélection de features.

Le problème de la sélection de features peut être formalisé ainsi :
On considère tour à tour chaque feature de l'espace $F_{generated}$, l'action à réaliser est alors le fait de garder ou non une feature. On se retrouve ainsi avec un arbre binaire dans lequel chaque niveau après la racine correspond, en considérant l'espace des features normalisées, à une décision sur une feature $f_i \in F_{normalized}$ $(i \in (1, .., n))$. Un exemple d'un tel arbre est présenté dans 1

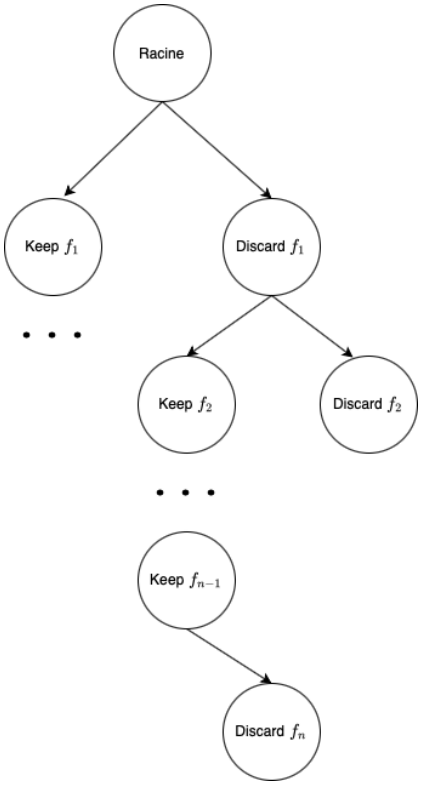

Figure 1: Un arbre pour le problème de la sélection de features

Une version plus évoluée consiste à avoir à la base de l'arbre un choix sur l'un ou l'autre des 3 espaces générés. Par simplicité, on évaluera automatiquement les features de l'espace $F_{normalized}$.

1.2.1 Méthodes de Monte-Carlo imbriquées pour la sélection de features

Les méthodes de Monte-Carlo imbriquées sont des variantes des méthodes de Monte-Carlo qui utilisent des appels imbriqués pour biaiser les playouts. Nested Monte-Carlo Search [Caz09] est un algorithme de Monte-Carlo imbriquée qui a obtenu de bons résultats pour plusieurs problèmes d'optimisation.

Une variante des méthodes de Monte-Carlo imbriquées à été développée par Chris Rozin et porte le nom de Nested Rollout Policy Adaptation (NRPA) [Ros11]. Cette variation permet d'apprendre en ligne une politique de playout. L'algorithme repose sur deux composantes majeures : une structure imbriquée introduite dans [Caz09] et une politique adaptative de playout. La politique adaptative de

rollout est alors une politique qui est paramétrisée par un poids sur chaque action. Les actions sont ainsi choisies pendant les playouts en utilisant cette politique. L'échantillonage de Gibbs est utilisé pour la mise à jour de la politique. Chaque action est codée par un index sur la politique qui retourne son poids.

NRPA possède cependant un problème d'évaluation. En effet, la meilleure séquence renvoyée par l'algorithme peut être biaisé vers la meilleure séquence du niveau 1 dans le cas où une bonne politique n'obtiendrait pas de bons résultats au niveau 0. Or, cette politique peut être meilleure ce qui souligne une restriction dans l'exploration au niveau 0. Pour résoudre ce problème une autre variante de l'algorithme a été introduite qui va générer P séquences au lieu d'une : Stabilized NRPA [CST20]. L'idée derrière l'algorithme est que la génération de ces plusieurs séquences puisse diversifier l'exploration au niveau 1.
Les auteurs ont ainsi noté que cela améliorait la stabilité de la convergence de NRPA.

Afin de réaliser la sélection de nos features, on passe par une application de l'algorithme NRPA stabilisée qui a obtenu de meilleurs résultats de manière générale pour notre problème. On considérera comme un *état*, un triplet avec : une liste d'indices à ajouter, une liste d'indice ajoutés et une liste d'indice non ajoutés.

Le pseudo code pour la version stabilisée de NRPA est le suivant :

Algorithm 2 The Stabilized NRPA algorithm.

```
1: StabilizedNRPA (level, policy)
2: if level == 0 then
3:     return playout(root, policy)
4: else if level == 1 then
5:     bestScore ← −∞
6:     for 1,..,P do
7:         (result, new) ← StabilizedNRPA(level-1, policy)
8:         if result ≥ bestScore then
9:             bestScore ← result
10:            seq ← new
11:        end if
12:    end for
13:    return (bestScore, seq)
14: else
15:    bestScore ← ∞
16:    for 1,..,N do
17:        (result, new) ← StabilizedNRPA(level-1, policy)
18:        if result ≥ bestScore then
19:            bestScore ← result
20:            seq ← new
21:        end if
22:        policy ← Adapt(policy, seq)
23:    end for
24:    return (bestScore, seq)
25: end if
```

- Dans notre cas, la valeur **bestScore** correspondra ainsi à une accuracy et **seq** sera une liste d'indice de features sélectionnées.

- Dans notre cas, l'état **root** sera un triplet avec une liste d'indices à ajouter qui correspond à tous les indices de features de l'espace utilisé, une liste d'indices ajoutés qui est vide et une liste d'indices non ajoutés qui est également vide.

Le pseudo-code de la fonction playout qui joue une partie aléatoire en utilisant la politique associée

est le suivant :

Algorithm 3 L'algorithme de playout.

```
Playout (state, policy)
sequence ← []
while True do
    if state is terminal then
        return (score(state), sequence)
    end if
    z ← 0.0
    for m in possible moves for state do
        z ← z + exp(policy[code(m)])
    end for
    choose a move with probability exp(policy[code(m)]) / z
    state ← play(state, move)
    sequence ← sequence + move
end while
```

- Dans notre cas, la fonction **score** sera calculée en effectuant une validation croisée à 3 blocs avec les features sélectionnées et en renvoyant la métrique définie par l'utilisateur pour le pipeline.

- Dans notre cas, la fonction **code** renverra un indice pour l'action m sur un vecteur de réel de taille $k = 2 * |F_{normalized}|$ correspondant aux poids de la politique.

Enfin, le pseudo code de la fonction *adapt* qui adapte la politique est :

Algorithm 4 L'algorithme Adapt.

```
Adapt (policy, sequence)
polp ← policy
state ← root
for move in sequence do
    polp[code(move)] ← polp[code[move)] + α
    z ← 0.0
    for m in possible moves for state do
        z ← z + exp(policy[code(m)])
    end for
    for m in possible moves for state do
        polp[code(move)] ← polp[code(move)] - α × exp(policy[code(m)]) / z
    end for
    state ← play(state, move)
end for
policy ← polp
```

Finalement, on obtient comme pseudo-code pour l'automatisation complète de la sélection des features l'algorithme suivant :

Algorithm 5 Automatisation de l'ingénierie des features

Input

- ◇ Un jeu de données $D_{F_{initial}}$ avec ses features dans l'espace $F_{initial}$.
- ◇ Une liste de k algorithmes $A = (A_1, ..., A_k)$ pour lesquels générer les meilleurs sous-ensembles de features.

Output

- ◇ Une liste de k sous-ensembles de features donnant les meilleurs performances pour les algorithmes dans A.

```
1: procedure GENERATEBESTFEATURESSETS
2:     F_generated, F_normalized, F_standardized ← generateFeaturesSet(D_F_initial).
3:     couplesAlgoBestFeatures ← {}
4:     for algorithm in A do
5:         Select one of the feature sets and generate the root state.
6:         bestScore, seq ← StabilizedNRPA(level, policy)
7:         couplesAlgoBestFeatures ← couplesAlgoBestFeatures ∪ [algorithm, seq]
8:     end for
9:     return couplesAlgoBestFeatures
10: end procedure
```

L'application de cet algorithme permet ainsi de se retrouver, pour les algorithmes fournis dans l'outil, avec un ensemble de features qui permettra de fournir de meilleurs résultats pour la sélection du modèle et de restreindre l'espace de recherche pour l'optimisation des hyperparamètres. Cependant, on se rend compte que la boucle for va tourner sur tous les algorithmes pour déterminer le meilleur ensemble de features. Cela montre à nouveau l'apport que pourrait donner une distribution des calculs par algorithme.