Rapport de projet de spécialité
Challenge Kaggle 4

Céline Duval Maxime Ollivier Julian Bustillos
Jean-Baptiste Le Noir de Carlan Loïc Masure

14 octobre 2016

Table des matières

1 Introduction

1.1 Apprentissage automatique - Machine learning

Le *machine learning* est une sous-catégorie de l'intelligence artificielle. Il peut être vu comme l'implémentation d'algorithmes apprenant à partir de données. Celles-ci vont permettre à l'algorithme d'établir certaines propriétés sur ces données et ensuite de pouvoir résoudre différents problèmes, essentiellement de la classification, sur d'autres données. Les algorithmes de *machine learning* sont souvent divisés en deux catégories : les algorithmes d'apprentissage supervisé et ceux d'apprentissage non supervisé. On parle d'apprentissage supervisé lorsque les classes d'appartenance des données sont connues à l'avance, et d'apprentissage non supervisé lorsque ni leur nombre ni leur nature ne sont connus et vont être déterminés par l'algorithme lui-même. Ce projet concerne l'apprentissage supervisé.

	Caractère 0(int)	Caractère 1(string)	Caractère 2(date)	...	Caractère d-1 (boolean)
Occurrence 0	12	"Pierre"	01/01/1970		1
Occurrence 1	31	"Paul"	14/07/2000		0
Occurrence 2	17361	"Jacques"	25/12/0000		0
...					
Occurrence n-1	2	"Marcel"	29/02/2008		1

FIGURE 1 – Formulation d'un tableau de données

Ce tableau de données d'entraînement est complet. Dans les données de test, il manque une caractéristique par rapport aux données d'entraînement qu'il faudra deviner pour chaque ligne.

1.2 Kaggle

Nous avons décidé de choisir notre sujet sur Kaggle. C'est un site internet américain créé en 2010 qui permet aux amateurs comme aux professionnels de s'exercer seuls ou par équipes à des challenges de *machine learning* proposés par des entreprises ou par le site Kaggle lui-même. Pour chaque challenge, un échantillon d'entraînement et un échantillon de test sont proposés. L'échantillon d'entraînement est une liste d'individus pour lesquels sont fournis un certain nombre de critères ainsi que la donnée sur laquelle se feront les prédictions. Il permet de construire et de perfectionner notre algorithme. Il est aussi fourni un échantillon de test contenant les individus pour lesquels la prédiction doit être faite à partir de notre algorithme. La performance de cet algorithme est ensuite évaluée par Kaggle en fonction du taux de prédictions correctes pour ce jeu de données.

Pour chaque challenge, un classement en temps réel est disponible. Lorsque les challenges sont proposés par les entreprises, une récompense financière est attribuée aux quelques premiers du classement au terme de la période de compétition en échange de la propriété intellectuelle de leur algorithme. Une interview est aussi réalisée lors de laquelle les gagnants expliquent leur cheminement et la manière dont fonctionnent leurs algorithmes, appelé *pipeline*. Un forum relativement actif est disponible sur le site, les utilisateurs peuvent ainsi partager des informations et poser des questions.

1.3 Python

Nous avons décidé d'utiliser le langage Python pour implémenter nos algorithmes. Celui-ci est doté de deux avantages importants pour ce projet. Le premier est qu'il est exécutable à la volée et très léger dans sa syntaxe, ce qui permet de coder et de debugger très rapidement. Le second est qu'en tant que langage de programmation fonctionnelle, on peut très facilement faire appel à des bibliothèques implémentant déjà les algorithmes utilisés. En l'occurrence, nous avons employé la bibliothèque **scikit-learn** qui fournit la plupart des algorithmes de *machine learning*.

Nous avons opté pour la version **Python 2.7**. En effet, la version **Python 3** présente certains bugs pour des packages.
Nous avons utilisé l'ensemble de packages fournis par **Anaconda**, une distribution de Python : ceux-ci comprennent Python.2.7, toute la bibliothèque **scikit-learn** ainsi que l'éditeur de texte Spyder adapté au langage Python, Jupyter, etc ...
Scikit-learn est un site open-source qui propose sa propre bibliothèque et de nombreux codes illustrant l'utilisation de ses packages. On y retrouve aussi une liste exhaustive des algorithmes proposés pour le *machine learning*. Celle-ci nous a donné de nombreuses pistes pendant le projet.
Python n'est pas un langage enseigné à l'ENSIMAG. En 2015, l'IEEE (Institut des ingénieurs électriciens et électroniciens) publie une étude qui montre que Python est le troisième langage le plus demandé par les entreprises (étude réalisée aux États-Unis). Il nous a donc semblé judicieux d'apprendre ce langage afin d'étoffer notre CV. D'un point de vue fonctionnalité, certains packages Python, dont **xgboost** développé plus bas, sont codés en C et sont donc plus rapides en exécution qu'un package en python, et nous permettent de gagner un temps non-négligeable pour certains calculs.

2 Titanic

2.1 Description des données

L'échantillon d'entraînement contient des données sur 891 passagers : Nom, Sexe, Age, Nombre de frères/sœurs/conjoints, Nombre de parents/enfants, Numéro de ticket, Prix payé, Cabine, Numéro de classe, Port d'embarquement. Il est également indiqué si chaque voyageur de cet échantillon a survécu ou non (1 s'il a survécu, 0 sinon). Le but est alors de prévoir la survie ou non de 417 voyageurs d'un second échantillon contenant les mêmes donnés.
Après avoir étudié les données, nous pouvons rapidement mettre un place un modèle naïf basé sur le sexe des individus : on prédit que les femmes survivront et les hommes non. En effet, comme indiqué sur le graphe ci-dessous, les femmes ont une tendance plus élevée à survivre (74% contre 19% pour les hommes).

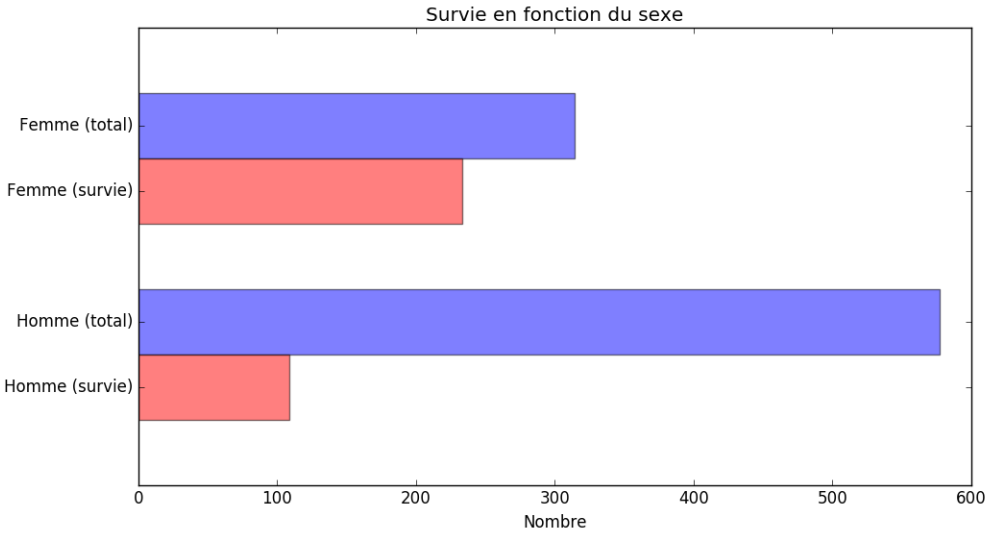

FIGURE 2 – Survie en fonction du sexe

Ce premier modèle basique nous a donné un score de 0.76555.

2.2 Traitement des données

La première étape logiquement parlant est le traitement des données, aussi appelé *feature engineering*. Le but est d'adapter les variables pour qu'elles apportent le plus d'information possible. Cette étape se réalise par tâtonnement car on ne peut pas prévoir à l'avance les conséquences d'une certaine adaptation. Cette phase comporte surtout de l'intuition et de la recherche, il faut tester chaque idée et la conserver si elle augmente notre score de bonnes prédictions.

Nous avons choisi d'abandonner le ticket car nous avions trop peu de données et de nombreuses autres catégories possibles.

Pour ce sujet, nous avons créé et testé les variables suivantes :

- Calcul de la taille des familles à partir de leurs nombres de frères/sœurs/épouses et parents/enfants
- Détermination de catégories à partir du titre des voyageurs contenu dans leurs noms (Mrs, Mr, Master, Miss, Major, ...)
- Les cabines ont étés identifiées par leur lettre (A-G,T) ou bien par la mention 'inconnu' si la variable ne contenait aucune valeur.
- Certains produits de variables ont été effectués avec le sexe (0 ou 1) ou bien le numéro de classe. Cela se justifie par le fait que le sexe permet de filtrer les hommes étant donné l'influence du sexe vue dans le premier modèle, de même pour la classe.

D'autres adaptations ont été mises en place, comme le codage *one-hot* pour le port d'embarquement qui est une variable catégorielle. Cela permet de faire disparaître la distance induite par un encodage avec des entiers. De plus, nous avons centré et réduit des variables et séparé les prix des billets en catégories choisies à partir des quartiles.

Enfin, nous avons aussi dû compléter les données lorsqu'elles n'étaient pas présentes pour certains utilisateurs. Nous avons essayé de substituer ce manque par la moyenne sur tous les individus et l'extrapolation par régression linéaire à partir des mêmes données présentes pour les autres passagers.

2.3 Validation des modèles

Le nombre de soumissions sur le site Kaggle étant limité à 10 par jour pour ce challenge, nous devions trouver un moyen pour évaluer nous-même la qualité de nos prédictions afin de ne soumettre que les meilleures. Pour cela, nous avons mis en place un script de validation croisée *10-fold*. Celui-ci sépare les données de l'échantillon d'entraînement en 10 ensembles de taille égale ou presque. Un apprentissage est alors effectué sur 9/10 de l'échantillon et une prédiction sur le 1/10 restant, cela pour chaque combinaison possible (au nombre de 10). Nous regardons ensuite la moyenne des scores afin de valider ou non notre modèle.
Évidemment, pour un algorithme donné, ce score de prédiction ne sera pas le même que celui obtenu sur Kaggle car les données sur lesquelles il est calculé ne sont pas les mêmes. Ce score est toutefois un bon indicateur car il est la plupart du temps un peu supérieur à celui de Kaggle mais reste très proche.

2.4 Modèles de prédiction

2.4.1 Algorithmes

Une fois le travail sur les données effectué, nous avons prédit la survie ou non des voyageurs de l'échantillon de test à l'aide de différents algorithmes présents dans la bibliothèque *scikit-learn* de Python. Le détail du fonctionnement de ces algorithmes est fourni en annexe afin de mieux nous concentrer dans cette partie sur l'aspect recherche et amélioration des prédictions.
Nous avons commencé par tester différents algorithmes en faisant varier les paramètres. Cela nous a ainsi permis une première élimination des algorithmes ne semblant pas adaptés au problème fourni. Cependant, les résultats obtenus étaient presque tous en-deçà du score donné par le modèle basique évoqué antérieurement.

Nous nous sommes alors penchés sur la sélection de variables. En effet, nos temps de calculs étant acceptables, il nous était possible de les itérer de nombreuses fois. Nous avons donc tenté d'améliorer chacun des algorithmes "efficaces" par la méthode suivante : on énumère chaque combinaison de variables possible et on regarde le score obtenu. On récupère pour chaque algorithme le score le plus élevé obtenu par notre validation. Un des avantage de cette technique est que chaque opération est indépendante, on peut donc les traiter en parallèle et ainsi obtenir un gain de temps considérable.

2.4.2 Combinaison d'algorithmes

Nous avons essayé de combiner les différents résultats obtenus afin d'améliorer notre score. Nous avons pris les résultats des algorithmes les plus intéressants et avons attribué 1 si la majorité des algorithmes le voyait survivre et 0 sinon. Nous avons aussi pondéré chaque algorithme suivant son résultat Kaggle. Mais cela ne nous a pas donné de résultat intéressant.

Nous avons alors tenté une autre méthode en utilisant nos trois meilleurs algorithmes : SVM, KNN et random forest. Nous avions les résultats pour le test avec SVM que l'on notera **res_svm**, de même pour **res_knn** et **res_rf**.
Nous voulions alors savoir comment déterminer le meilleur résultat avec ceux-ci. Nous avons alors divisé le train en deux (**train_train** et **train_test**) afin de faire l'apprentissage pour choisir entre ces trois algorithmes.
On applique alors SVM, KNN et RF à **train_train** afin de prédire **train_test**. Nous avons alors **predict_svm**, **predict_knn** et **predict_rf** qui prévoient **train_test** avec **train_train** avec ces trois algorithmes.
Nous faisons alors de l'apprentissage en utilisant un algorithme sur **predict_svm**, **predict_knn** et **predict_rf**. Nous avons en effet le vrai résultat de **train_test** (puisqu'il appartient au train). Par exemple le résultat de **train_test** est 1, la prédiction de **predict_svm** est 0, la prédiction de **predict_knn** est 1, la prédiction de **predict_rf** est 1. L'algorithme va donc apprendre que lorsque SVM et RF sont à 1 et KNN est à 0, il faudra mettre 1 comme résultat final.
Pour faire cet apprentissage nous avons utilisé les algorithmes SVM, KNN, RF, et la régression logistique.
En fait nous apprenons sur le fait de choisir 1 ou 0 en fonction du résultat des 3 différents algorithmes.
Nous appliquons alors cet apprentissage sur **res_svm**, **res_knn** et **res_rf** pour déterminer les résultats du test.
Malheureusement cette méthode n'a pas donné de résultat satisfaisant.

2.5 Pipeline proposée

La pipeline est l'ensemble des démarches effectuées pour atteindre notre meilleur score.
Nous commençons d'abord par obtenir les âges manquants par régression linéaire, et remplacer les prix de billets manquant par la moyenne de ces derniers. On construit ensuite les variables décrites antérieurement.
La construction de notre solution optimale est basée sur l'algorithme Support vector machine avec un noyau gaussien. Nous avons commencé par effectuer une sélection de variables sur toutes les variables créées antérieurement. Cette étape nous a permis de réduire les 12 variables de départ de moitié. Les variables restantes sont les suivantes : classe du passager, son nombre de frère et sœur à bord, la taille de sa famille à bord, son sexe, le titre devant le nom et si le voyageur a embarqué ou non au port de Southampton.

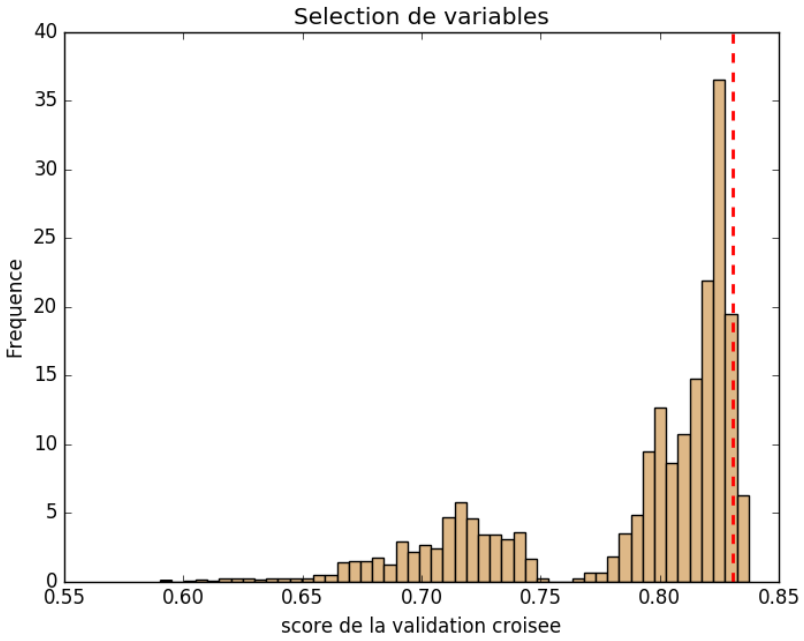

FIGURE 3 – Sélection de variables

Ci-dessus les résultats de la sélection de variables. La ligne rouge en pointillés correspond au résultat sans sélection de variables. On remarque que les résultats obtenus sont en grande partie proche de ce dernier, mais la meilleure solution est supérieure. On passe ainsi de 0.830 à 0.837 pour notre validation croisée.
Il nous reste ensuite à lancer la prédiction sur les données de test et à soumettre le fichier sur Kaggle.

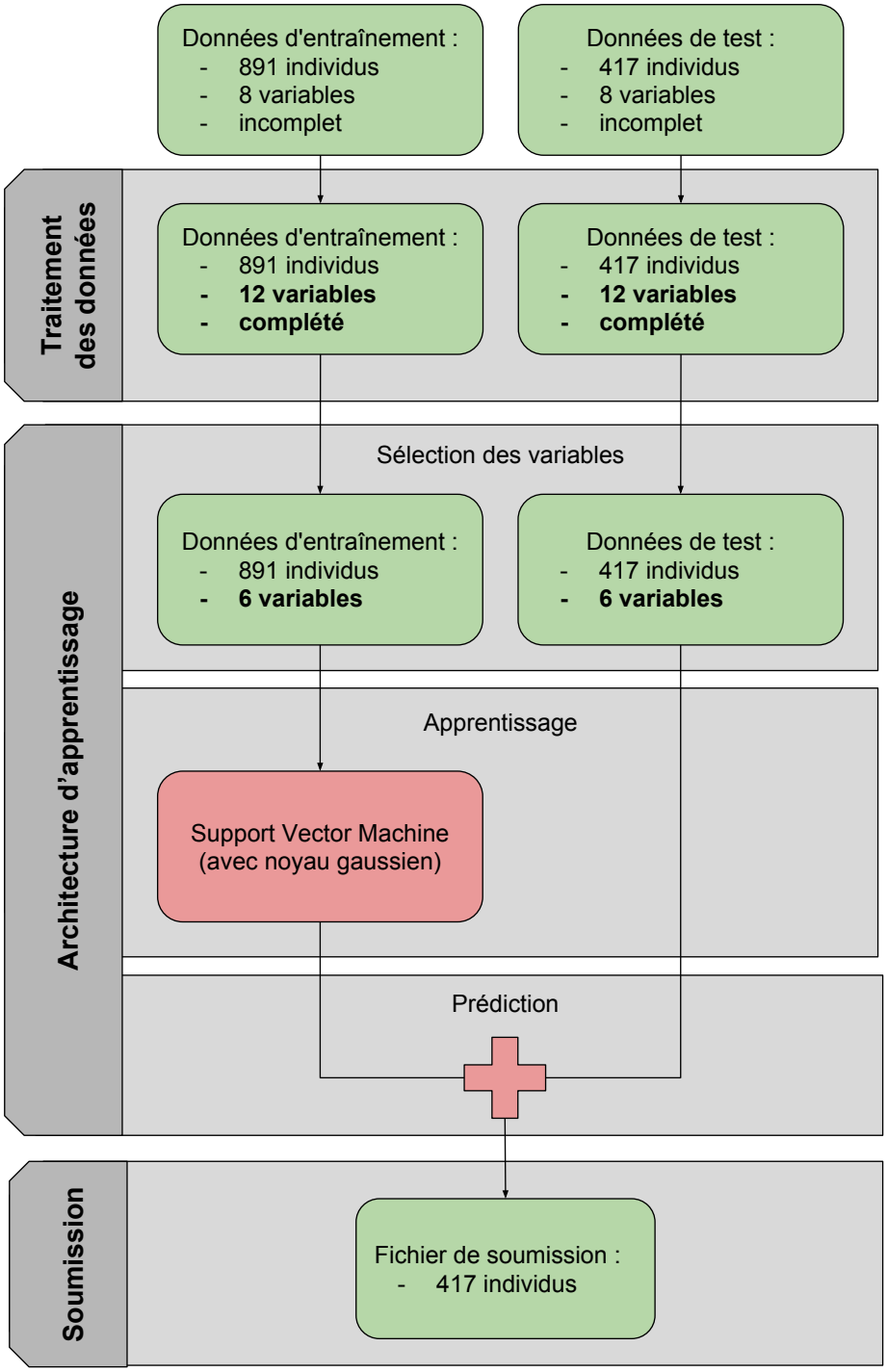

FIGURE 4 – Pipeline Titanic

Notre score final est de 0.80383, soit une progression d'environ 4% par rapport au modèle basique proposé.

2.6 Résultats

Les scores suivants sont ceux donnés par le site Kaggle lors de nos soumissions :

- Notre première soumission se basant seulement sur l'affirmation "les femmes survivent et les hommes meurent" a réalisé un score de 0.76555.
- Une simple régression logistique donnait un score de 0.75120
- L'algorithme de SVM avec un noyau linéaire donnait un score de 0.76555
- L'algorithme des K nearest neighbors nous a permis d'améliorer notre score à 0.77990.
- Les algorithmes de gradient boosting et Label propagation nous ont donné des résultats identiques.
- L' encodage en *one hot* du port d'embarquement nous donne dans le meilleur cas un score de 0.76555 pour l'algorithme du gradient boosting.
- Prendre en compte le titre du passager (Mrs, Lord, Master ...) et si le passager voyage seul ou en famille nombreuse nous a permis d'améliorer notre score en Random Forest de 0.78469.
- Nous avons ensuite réalisé un score de 0.78947 pour l'algorithme SVM et 0.78469 pour Label Propagation.
- En ajoutant au *feature engineering* l'encodage en *one hot* du port d'embarquement et la prise en compte du numéro de cabine nous a permis d'obtenir notre meilleur score de 0.80383 en utilisant l'algorithme du SVM avec un noyau gaussien.
- Ceci n'a cependant pas donnée d'amélioration pour les algorithmes de Label Propagation (0.76077), gradient boosting (0.75598) ou régression logistique (0.74641).
- Notre dernière technique de*feature engineering* de multiplier les caractères d'Age, Sexe et de classe, nous a donné un score de 0.79904 avec l'algorithme de SVM avec noyau gaussien.

2.7 Discussion

Une des premières étapes a été de réaliser un algorithme naïf. Cependant dans l'exemple du Titanic, celui-ci donne de particulièrement bons résultats et il nous a fallu utiliser des algorithmes plus poussés qu'une simple régression linéaire, dans notre cas l'algorithme des k nearest neighbors.
Nous avons ensuite tenté d'adapter les paramètres de la fonction de prédiction : en premier lieu coder en *one hot* uniquement le port d'embarquement. Mais le score sur Kaggle a baissé plutôt qu'augmenté.
En revanche, travailler sur le titre du passager et s'il voyage seul ou non nous a permis d'augmenter notre score. Nous avons donc continué à travailler sur les variables : dans le cas du numéro de cabine, ceci a été bénéfique. En revanche, multiplier les coefficients des caractères a eu un impact négatif sur nos algorithmes.
On peut remarquer que suivant le travail sur les variables effectué, l'algorithme qui donne le meilleur score n'est pas toujours le même. À chaque nouveau traitement des variables il nous fallait donc réitérer le processus.

2.8 Conclusion

Ce sujet d'introduction proposé par Kaggle a été une très bonne entrée en matière. Il nous a permis de nous former rapidement au langage Python et à ses bibliothèques dédiées au calcul ainsi qu'au machine learning. De plus, les problématiques d'analyse prédictive sont abordées sur un exemple aux difficultés limitées : nombre de passagers et de variables assez faibles, prédiction binaire (il faut prévoir si les passager meurent ou non). Cela nous a permis d'essayer de nombreux algorithmes sans être restreints par la puissance de nos machines.

3 Expedia

3.1 Introduction du sujet

Expedia est un site de réservation en ligne. Celui-ci voudrait proposer les hôtels les plus à même d'intéresser leurs clients en fonction de leurs profils. L'idée est donc d'adresser la meilleure recommandation possible pour chaque client. A partir des fichiers d'entraînement donnés par Kaggle, nous devons remplir dans un fichier **test.csv** les numéros d'hôtels conseillés pour chaque utilisateur. Kaggle nous laisse proposer jusqu'à 5 hôtels par utilisateur.

3.2 Description des données

Le site Kaggle met à notre disposition deux jeux de données : **train.csv** et **destinations.csv**. **train.csv** contient 37 millions de lignes qui correspondent chacune à la visite d'un utilisateur sur le site, qu'il ait fait une réservation ou non. À chaque ligne sont associées 24 colonnes qui présentent les caractéristiques de chaque visite, dont l'hôtel regardé et l'identifiant de l'utilisateur qui a visité le site.
La première chose que nous pouvons observer est qu'il sera difficile d'exploiter ces données telles quelles. Les algorithmes prendront un temps extrêmement long, et nous pouvons supposer que plusieurs caractères et visites produiront un bruit sur la prédiction.
Un des caractères est *is_booking*, il indique si la visite sur le site a entraîné une réservation de l'hôtel. Celui-ci est codé en tinyint avec 1 en cas de réservation et 0 sinon. Un premier écrémage des données a donc été de conserver uniquement les lignes pour lesquelles une réservation a été faite.
Ceci nous permet alors d'avoir une base de données d'entraînement de 3.7 millions de lignes.

On cherche maintenant à éliminer des colonnes. On remarque que *cnt* est une colonne présente dans **train.csv** mais pas dans **test.csv**. On ferait alors la fonction de prédiction à partir de **train** avec cette colonne mais on ne pourrait pas la ré-appliquer à **test** puisque cette colonne n'existe pas. Cette colonne est donc supprimée.

On s'intéresse maintenant à **destinations**. Le fichier **train** contient une colonne *srch_destination_id*. On retrouve cette colonne dans **destinations** associée à 149 autres caractères codés en double qui décrivent cette destination. Nous avons donc appliqué une ACP (Analyse en composantes principales) et gardé les trois premiers termes. Nous les avons ensuite rajoutés au fichier **train** et **test** pour les utiliser dans la fonction de prédiction.

Les colonnes des date de réservation, date de départ et date de retour sont des String, et donc inutilisables sous cette forme. Nous les avons traduites en différents caractères plus représentatifs : l'année, le mois, le jour, le numéro du jour dans la semaine, et même pour la date de réservation l'heure et la minute.

Les colonnes des dates de départ et de retour comportent de nombreux NaN dans les bases de données test. Or les dates d'arrivée et de départ pour un hôtel semblent avoir un impact sur le type d'un hôtel donc le choix de l'utilisateur.
Tout comme les données manquantes de l' *Age* dans Titanic, nous avons prédit ces NaN à l'aide d'une régression linéaire sur les dates déjà existantes : c'est-à-dire que pour chaque colonne créée plus haut à partir des caractères dates de réservation, de départ et de retour nous avons prédit chaque valeur qui était un NaN. Cela nous permet d'utiliser ensuite les colonnes de temps comme caractère de prédiction pour le choix de l'hôtel.

Il faut également remplir les valeurs manquantes de la colonne *orig.destination.distance* qui quantifie la distance entre un utilisateur et l'hôtel qu'il réserve. Pour ce faire, nous avons tenté de déduire cette information des colonnes *hotel.country, hotel.continent, user.country*. La différence avec la régression effectuée sur la date de réservation, c'est que ces colonnes sont des caractères discrets, non ordonnés (i.e. on ne peut pas définir d'ordre logique sur les continents ou les pays pour deviner la distance).

Nous effectuons donc une technique de **data imputation** qui consiste, pour chaque valeur manquante de *orig.destination.distance* à considérer toutes les réservations dont l'utilisateur vient du même pays que l'utilisateur courant, et dont le pays de l'hôtel réservé est le même que le pays de l'hôtel dans lequel l'utilisateur courant réserve son hôtel. En Python, la bibliothèque **Pandas** est dotée d'instructions similaires à des requêtes SQL pour extraire les données voulues. Nous faisons alors appel à une fonction d'aggrégation (moyenne ou médiane de la colonne *orig.destination.distance*) qui servira de valeur pour remplacer la valeur manquante à deviner.

La popularité d'un hôtel dépend de la période de l'année : par exemple, un hôtel à la plage peut être populaire en été seulement et un hôtel à la montagne en hiver seulement. De même le choix d'hôtel peut être différent suivant la durée du séjour. Nous avons donc implémenté les fonctions *saison* et *weekend* :
Une colonne est rajoutée *weekend*, en tinyint, telle que si le départ à lieu pendant le week-end, le paramètre vaut 1 et 0 s'il est en semaine. L'idée est la même pour *saison* : on a rajouté 4 colonnes : été, printemps, hiver, automne. Chacune est codée en tinyint de la même manière que *weekend*. Nous avons réutilisé la technique de codage binaire du Titanic pour la colonne contenant le port d'embarquement. Nous avions alors obtenu de meilleurs résultats. De même, la colonne *posa_continent* a été divisée en 5 pour chaque continent. En supprimant l'identification par 1,2,3,4, ou 5, l'encodage *one hot* enlève l'idée de distance.

Nous faisons de même pour la colonne *srch_destination_type_id*, nous utilisons le codage binaire pour différencier les destinations possibles (montagne, plage, campagne, ville...).

Ces caractères rajoutés et enlevés nous permettent une utilisation différente des données et peut possiblement aider à obtenir une meilleure fonction de prédiction, c'est-à-dire avec moins de bruit et des variables qui n'impliquent pas une idée de distance.

3.3 Validation des modèles

Le site Kaggle nous propose jusqu'à 5 soumissions par jour pour nous renvoyer un pourcentage de réussite sur notre fonction de prédiction. Pour chaque visite d'un utilisateur, inscrite dans **test.csv** nous devons prédire l'hôtel que l'utilisateur va réserver. Nous pouvons proposer jusqu'à 5 hôtels possibles par visite. Nous avons de plus implémenté une cross-validation comme dans Titanic, afin d'estimer notre fonction de prédiction à l'aide de la même formule que celle utilisée par Kaggle :

L'équation est la suivante, appelée MAP, Mean Average Precision, calculée ici pour 5 valeurs :

$$MAP@5 = \frac{1}{N}\sum_{i=1}^{N} AP@min(5, n)$$

avec

$$AP@n_i = \sum_{k=1}^{n_i} \frac{P(k)}{min(n_i, m_i)}$$

$$P(k) = \frac{nombre\ de\ bonnes\ predictions\ pour\ les\ k\ premiers\ elements}{k}$$

avec N = nombre de lignes
n_i = nombre de prédictions pour la i-eme ligne
m_i=nombre de résultats pour la i-eme ligne

Après plusieurs soumissions, il nous a semblé que la base de test avec laquel Expédia comparait nos résultats ne contenait qu'un hôtel par individu. D'après la formule de calcul d'erreur, soumettre 5 hôtels par individu nous permettra donc d'obtenir un meilleur score.

Nous avons cependant remarqué que la cross-validation prenait un temps extrêmement long et n'était donc pas indiquée pour calculer notre pourcentage de réussite.

Nous avons alors procédé à une validation temporelle : nous avons divisé notre base d'entraînement en deux, la première, **fichier1**, correspond aux visites faites en 2013 et 2014 avant juillet, la deuxième, **fichier2**, les visites après juillet 2014.

Nous tentons d'estimer les visites après juillet 2014 à partir de **fichier1**. On rappelle que nous n'avons gardé que les lignes telles que *is_booking* = 1. Nous avons ensuite comparé nos résultats à ceux de **fichier2** à l'aide la fonction MAP vue plus haut.
Une estimation de notre taux de réussite était ainsi obtenue. Celui-ci étant calculé de la même manière que sur Kaggle, nous pouvions donc savoir si notre score pouvait s'améliorer sur le site, et ainsi éviter les soumissions inutiles.

3.4 Modèles de prédiction

3.4.1 Algorithmes

Le forum Kaggle a rendu public un problème de *data leak* dans la base de données proposée : en regroupant les caractères *user_location_country*, *user_location_region*, *user_location_city*, *hotel_market* and *orig_destination_distance*, on trouve des similitudes entre le fichier **train** et **test** au niveau de cet ensemble de caractères. On peut alors pour chacune de ces similitudes récupérer dans **train** les *hotel_cluster* et les ajouter dans le fichier **test**.
L'administrateur Kaggle estimait que ce problème touchait environ $\frac{1}{3}$ des données. Nous avons implémenté cette astuce surtout par besoin d'estimation de notre niveau : en effet, ce problème de *data leak* a été rendu publique et est donc utilisé par une grande partie des utilisateurs participant à ce concours et faisant du *machine learning*. En l'utilisant nous aussi, nous pouvons plus facilement nous comparer à nos concurrents. Nous soumettons ainsi dans le cas d'un hôtel correspondant : en premier cet hôtel suivi des 4 autres meilleurs hôtels prédits par notre algorithme.

Comme dans Titanic, certaines colonnes n'apportent que du "bruit" et font perdre de la précision aux algorithmes. L'idée est donc de garder les caractères pour lesquels le taux de réussite de la fonction de prédiction est le plus élevé. Encore une fois, vu le nombre important de données et de caractères, il est impératif de mettre en place un

algorithme rapide. Son fonctionnement est le suivant :

- On calcule d'abord le taux de réussite pour un algorithme avec tous les caractères. On l'appelle *max*. On calcule ensuite le taux de réussite pour chacun des cas où on enlève un paramètre.

- On a alors deux cas possibles : soit *max* est supérieur à ces nouveaux taux de réussite, et l'algorithme s'arrête et renvoie *max* et les caractères utilisés. Sinon on récupère les nouveaux caractères avec le taux de réussite maximal et on remplace l'ancien *max*. L'algorithme continue ensuite en enlevant à nouveau chacun des caractères possibles, jusqu'à l'arrêt de l'algorithme (alors que Titanic testait tout les cas possibles de un au maximum de caractères).

L'énumération de l'algorithme n'étant pas complète, elle propose cependant l'avantage de pouvoir être arrêtée à chaque étape en ayant un résultat meilleur (non strictement). Ainsi, certains algorithmes ont dû être arrêtés avant la fin par manque de temps tout en fournissant de meilleurs résultats.

3.4.2 Combinaison d'algorithmes

Pour améliorer notre score Kaggle nous avons essayé de combiner nos meilleurs résultats afin d'en obtenir un supérieur.
Pour ce faire, nous avons réuni nos 8 meilleures prédictions. Chaque prédiction affiche 5 hôtels par *id*(numéro de ligne). Pour chaque *id* nous avons pris les 5 hôtels prédits par chacune des 8 classifications. Pour chaque hôtel nous avons attribué un certain nombre de "points".
Par exemple on attribue 5 points au 1^{er} hôtel, 4 au $2^{\text{ème}}$, 3 au $3^{\text{ème}}$, 2 au $4^{\text{ème}}$, 1 au $5^{\text{ème}}$ et 0 aux autres. Les points sont additionnés pour les 8 séries de 5 hôtels. Nous avons aussi pondéré les différentes prédictions : par exemple, le 1^{er} hôtel provenant d'une prédiction affectée d'un coefficient 2 aura $2 \times 5 = 10$ points.
Pour un *id* donné la prédiction finale sera composée des 5 hôtels ayant obtenu le plus de points.
Ainsi nous avons amélioré notre score Kaggle de $0,756\%$, avec des algorithmes de *k nearest neighbors* et *grad boosting*.

Nous avons essayé une méthode d'agrégation appelée *bagging* qui réalise plusieurs apprentissages sur des sous-ensembles de l'ensemble d'apprentissage et pondère ces derniers afin de donner la prédiction.

Nous avons aussi mis en place un système de pondération de plusieurs algorithmes : un poids est affecté à chacun des algorithmes, les probabilité renvoyés par chacun de ces algorithme pour chaque hôtel lors d'une prédiction sont ainsi pondéré avant d'être sommés et renvoyer la probabilité finale. Nous avons de plus tenté de trouver des meilleurs poids en définissant une grille de poids et en essayant d'obtenir la validation avec le meilleur score. Enfin, nous avons préféré utiliser un algorithme de minimisation de l'erreur en sortie pour obtenir des poids optimaux, donnant de meilleurs résultats en un temps bien plus court.

3.5 Pipeline proposée

Nous avons commencé par traiter les donnés comme décrit dans la partie éponyme. Ensuite nous avons réalisé une sélection des données les plus pertinentes pour entraîner l'algorithme knn. Cela a permis d'obtenir une première amélioration significative de notre score, comme on peut le voir sur la Figure 5 ci-dessous :

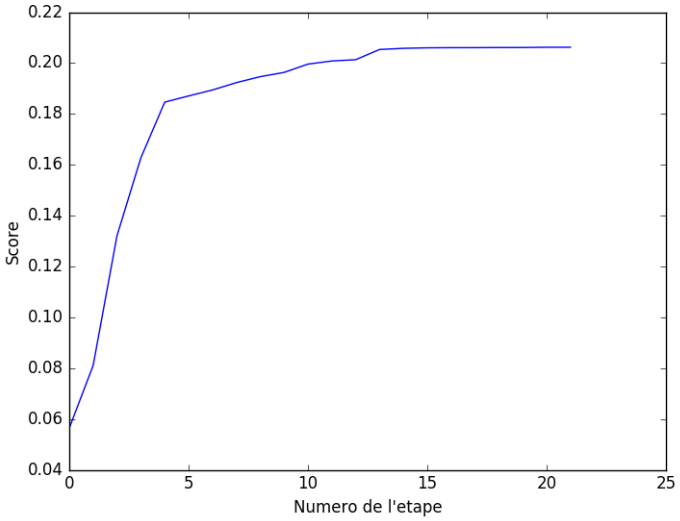

FIGURE 5 – Score en fonction du nombre d'étapes de l'algorithme

Enfin, nous avons essayé de combiner knn avec une régression logistique et un *gradient boosting*, ces derniers donnant aussi de bons résultats. Pour cela, nous avons sélectionné, en partant des variables utilisées pour knn, les variables et les poids permettant de minimiser l'erreur sur notre validation.
La prédiction obtenue utilisée conjointement avec le *data leak* nous a ainsi permis d'obtenir notre meilleur score : 0.3615.

Un schéma explicitant la *pipeline* est présent à la page suivante.

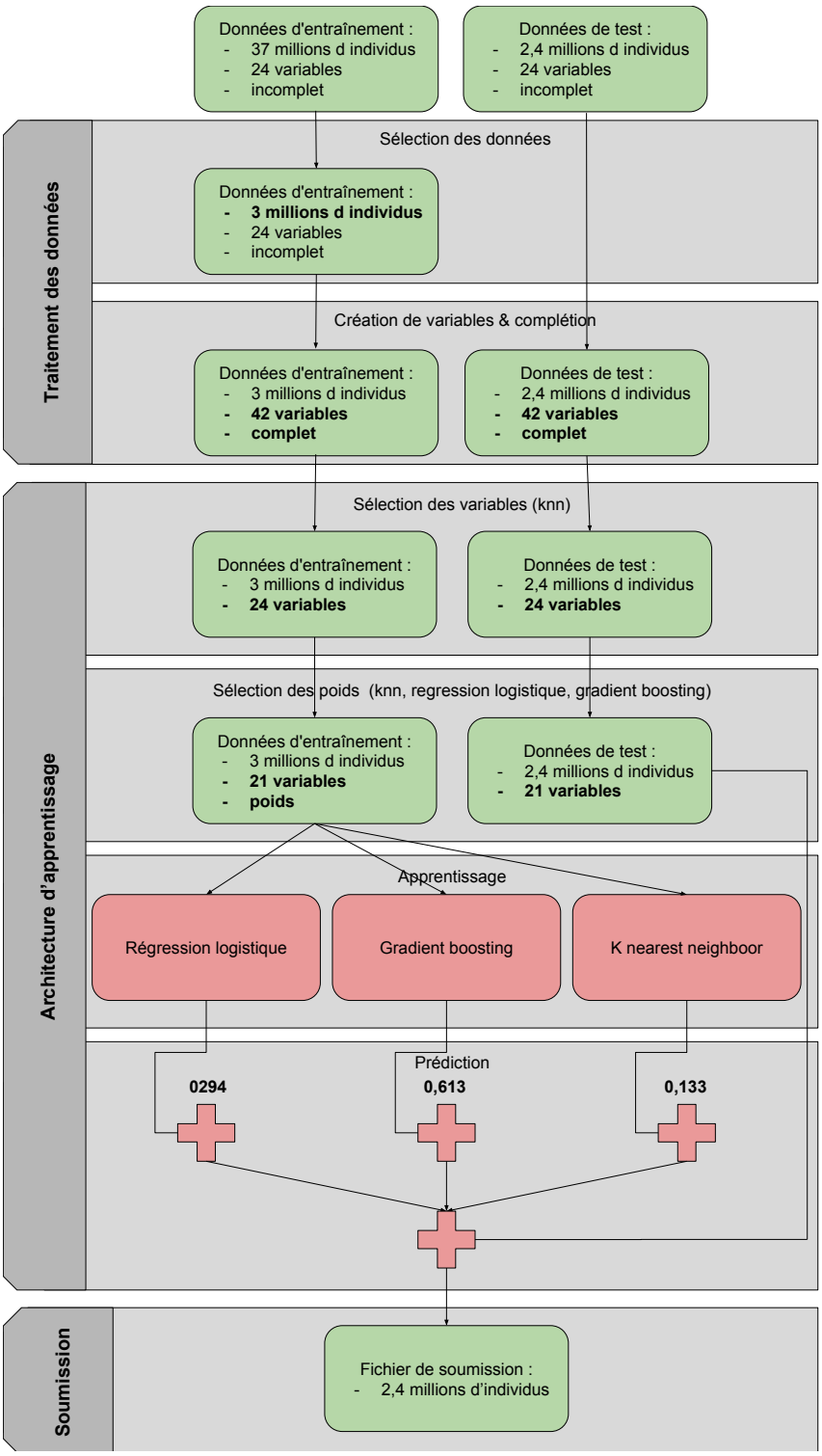

FIGURE 6 – Pipeline Expedia

3.6 Résultats

- Pour commencer, seuls 10000 événements ont été choisis aléatoirement pour constituer l'ensemble d'entraînement afin d'obtenir des temps de calcul raisonnables. La premier essai que nous avons réalisé a été de prédire pour chaque événement les 5 hôtels les plus représentés parmi les données d'entraînement. Nous avons alors obtenu un score de 0.06986.
- La régression logistique a donné 0.08812.
- Des méthodes telles que *random forest* et *gradient boosting* ont augmenté nos scores jusqu'à environ 0.13962.
- Nous avons ensuite stagné jusqu'à la prise en compte du *data leak* expliquée précédemment et l'utilisation de 100000 individus qui nous a fait atteindre un score 0.28401.
- En remplaçant les NaN par un régression linéaire, nous avons réussi à obtenir un score de 0.28752.
- L'avancée suivante a été réalisée avec l'utilisation de la ACP sur les variables de **destinations.csv**, du feature engineering (encodage *one hot*, sélection de variables et création de nouvelles variables), et la prise en compte de tous les individus qui nous a donné notre score actuel de 0.35398 avec l'utilisation de l'algorithme des *k nearest neighbors*.
- En pondérant différents algorithmes par des poids obtenus avec une minimisation de l'erreur, nous avons obtenu 0.3615.
- Nous avons aussi tenté d'utiliser l'encodage *one hot* sur le type de destination (plage, montagne...) mais le résultat était seulement de 0.35026.

3.7 Discussion

Comme dans Titanic, nous avons commencé par utiliser un algorithme naïf qui nous a donné des résultats plutôt faibles. Des algorithmes plus poussés prenant en compte les caractères de chaque visite nous ont permis de donner de meilleures prédictions.
En prenant en compte 10 fois plus d'individus, l'algorithme avait 10 fois plus d'information et a donc pu produire une fonction de prédiction beaucoup plus précise que précédemment. On le voit de même quand nous passons de 1000000 à 3.7 millions de visites. Par faute de temps et de matériel adapté, nous n'avons pas pu tester si l'utilisation de l'ensemble des données (37 millions de visites) aurait pu nous permettre d'augmenter notre score.
Remplacer les données non renseignées (NaN) à l'aide d'une régression linéaire nous a permis d'utiliser trois nouveaux caractères (jour de réservation, jour de départ et jour d'arrivée) qui peuvent avoir un impact sur l'hôtel choisi. Et donc avoir une fonction de prédiction plus précise.
Utiliser l'encodage *one hot* nous a permis d'annuler la notion de distance et ainsi d'améliorer notre score, mais seulement pour certains caractères : les continents, les saisons, si le voyage est pendant un *weekend*... alors que l'encodage du type de destination n'a pas donné d'amélioration.
Quand on applique la ACP, nous améliorons nos scores d'algorithmes. Rajouter la sélection de variables augmente encore plus notre score. Cependant la sélection de variables seule donne les meilleurs résultats, d'où le fait que la ACP n'est pas utilisée dans notre pipeline.
Nous notons aussi l'impact de la combinaison de plusieurs algorithmes entre eux (ceux ayant parmi les meilleurs résultats) pour augmenter la précision de près de 1%.

3.8 Conclusion

À travers ce sujet, nous avons été exposés aux problématiques concrètes qui se posent régulièrement en *machine learning*, telles que la taille exubérante des données, la puissance limitée de nos outils de travail ou encore le nombre important de classes pour le critère à prédire. Ce fut difficilement gérable au commencement du projet mais nous avons ensuite réussi à contourner habilement ces problèmes, notamment en exerçant notre algorithme sur un sous-ensemble des données et non sur son intégralité. Évidemment, cela réduisait l'information disponible et diminuait donc la qualité possible de nos algorithmes, cependant cela nous a permis un premier travail de base sur les données et la manière de les utiliser. L'utilisation de packages spécialisés de python nous a ensuite permis de travailler sur un ensemble de données plus conséquent et de finalement aboutir à notre meilleur score sur Kaggle de 36%.

4 Annexe - Principaux algorithmes utilisés

4.1 Régression linéaire

Le modèle de régression linéaire est un modèle de régression d'une variable expliquée sur une ou plusieurs variables explicatives dans lequel on fait l'hypothèse que la fonction qui relie les variables explicatives à la variable expliquée est linéaire dans ses caractères. Soient les vecteurs d'entrées $x_i = (x_{i1}, ..., x_{ip})$, et n-couples $((x_1, y_1), ..., (x_n, y_n))$. Le modèle est de la forme :

$$Y_i = \beta_0 + \sum_{j=1}^{p} x_{ij}\beta_j + \epsilon_i$$

où ϵ_i est un terme résiduel avec $E[\epsilon_i] = 0$ et $var(\epsilon_i) = \sigma^2$.
Ce modèle est estimé par la méthode des moindres carrés. On va chercher les coefficients β pour minimiser :

$$RSS(\beta) = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2$$

4.2 Régression logistique

La régression logistique est un modèle de classification. Les variables sont catégorielles.
Le but est de modéliser la probabilité d'appartenance d'un individu à une catégorie « k ».
On considère $Y(\omega)$ la modalité de Y prise par un individu ω. $(X_1(\omega), ..., X_j(\omega))$ est la description d'un individu ω dans l'espace des variables explicatives.
$\Pi_k(\omega) = P[Y(\omega) = y_k | X(\omega)]$ avec $\sum_k \Pi_k(\omega) = 1$.
On a $(K-1)$ équations LOGIT :

$$LOGIT_k(\omega) = ln\left(\frac{\Pi_k(\omega)}{\Pi_K(\omega)}\right) = a_{0,k} + a_{1,k}X_1(\omega) + ... + a_{j,k}X_j(\omega)$$

On en déduit les $(K-1)$ probabilités d'affectation :

$$\Pi_k(\omega) = \frac{e^{LOGIT_k(\omega)}}{1 + \sum_{k=1}^{K-1} e^{LOGIT_k(\omega)}}$$

et

$$\Pi_K(\omega) = 1 - \sum_{k=1}^{K-1} \Pi_k(\omega)$$

Puis pour affecter à la catégorie :

$$Y(\omega) = y_{k*} \Leftrightarrow y_{k*} = argmax_k[\Pi_k(\omega)]$$

.

4.3 Arbre de décison - Decision tree

Dans cet algorithme il faut que toutes les variables soient catégorielles et il faut donc créer plusieurs catégories pour les variables quantitatives. L'arbre de décision est alors un arbre au sens algorithmique avec des nœuds et des branches. Partant d'un noeud, chaque branche correspond au choix d'une catégorie pour une certaine variable. Chaque nœud terminal correspond à une situation pour laquelle toutes les catégories pour toutes les variables ont été déterminées et indique la prédiction correspondante. L'algorithme consiste, étant donné une entrée, à se balader dans l'arbre depuis la racine en considérant les catégories d'appartenance aux variables de cette entrée jusqu'à arriver à un nœud terminal qui indique alors la prédiction pour cette entrée.

4.4 Forêt d'arbres décisionnels - Random forest

Cet algorithme est une version améliorée de celui des arbres de décision. Il effectue un apprentissage sur de multiples arbres de décision entraînés sur des sous-ensembles de données légèrement différents.

4.5 Méthode des k plus proches voisins - K nearest neighbor

Dans le modèle des K nearest neighbor, K étant choisi et fixé, l'algorithme va déterminer la classe d'appartenance d'une entrée comme étant la classe la plus représentée parmi ses k plus proches voisins selon une distance qu'il faut définir.

4.6 Machine à vecteurs de support - Support vector machine

Le Support Vector Machine est un algorithme géométrique. Étant donnés des points répartis en 2 classes (ou plus) dans un espace vectoriel de dimension au moins 2, il faut trouver un hyperplan affine de telle sorte que tous les points d'une même classe soient d'un même côté de l'hyperplan. Concrètement, il s'agit de maximiser $\min\{l_i h_i d(x_i, H)\}$ où l_i vaut 1 ou -1 selon la classe du point, $d(x_i, H)$ est la distance du point x_i à l'hyperplan H, et h_i qui vaut aussi 1 ou -1 selon que le produit scalaire $< x_i, n_H >$ est positif ou non.

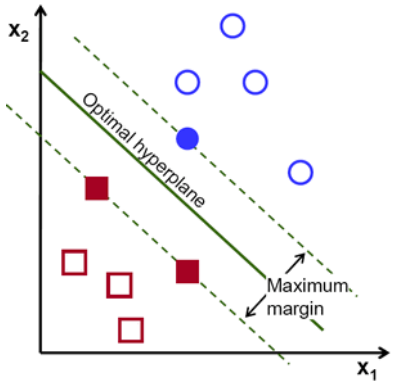

FIGURE 7 – Exemple de recherche de marge optimale entre le point le plus proche du plan et le plan lui-même.

Cet algorithme a une complexité comprise entre $d.n^2$ et $d.n^3$ où d est le nombre de classes à distinguer et n est le nombre de points à considérer. On peut souligner par la même que cet algorithme ne passe donc pas très bien à l'échelle puisque que pour

un grand nombre de données, le temps de calcul explose. On a pu le vérifier durant le projet puisque l'application du SVM aux données d'Expedia a été bien plus longue que les autres algorithmes utilisés.

Il est aussi possible de choisir un noyau (linéaire par défaut) qui permet d'appliquer une transformation à l'espace d'origine. C'est ensuite dans ce nouvel espace que l'on va recherche l'hyperplan optimal.

4.7 Propagation de classe - Label propagation

L'algorithme de *Label Propagation* est une technique de classification semi-supervisée. Il s'agit, étant donné un ensemble de points d'un espace, dont on connait pour une partie seulement la classe à laquelle ils appartiennent, de deviner les classes inconnues des autres points. Pour cela, il utilise un graphe complet dont les noeuds représentent les classes à deviner et dont les arcs sont pondérés par une fonction de distance issue de la similarité entre deux points(selon les autres critères connus). Ces distances représentent alors les probabilités qui serviront pour tirer au sort le label du noeud courant. Puis, on met à jour les distances et on "devine" de proche en proche les labels manquants. Cet algorithme est similaire à celui des plus proches voisins.

4.8 Adaboost

Le nom vient en fait de la contraction de Adaptative Boosting. Cet algorithme utilise beaucoup de prédicteurs faibles, c'est à dire des prédicteurs qui n'obtiennent pas un bon score de prédiction lorsqu'ils sont utilisés seuls, et qui utilisent peu de variables. Adaboost attribue alors un poids à chacune des prédictions : $F(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$ avec h_t les prédicteurs et α_t les poids qui leurs sont attribués. Cet algorithme mise sur le fait qu'avec un grand nombre de prédicteurs les erreurs vont se compenser pour obtenir un score de prédiction correct.

4.9 Gradient boosting

Cette technique de *boosting* est majoritairement employée avec des arbres de décision. L'idée principale est là encore d'agréger plusieurs classificateurs ensembles mais en les créant itérativement. Ces classificateurs faibles sont généralement des fonctions simples et paramétrées, le plus souvent des arbres de décision. Le super-classificateur final est une pondération (par un vecteur w) de ces classificateurs faibles.

Une approche pour construire ce super-classifier est de :

- Prendre une pondération quelconque (poids w_i) de classificateurs faibles (paramètres a_i) et former son super-classificateur
- Calculer l'erreur induite par ce super-classificateur, et chercher le classificateur faible qui s'approche le plus de cette erreur (ce qui revient à le chercher dans l'espace des caractères)
- Retrancher le classificateur faible au super-classificateur tout en optimisant son poids par rapport à une fonction de perte
- Répéter le procédé itérativement

Le classificateur du gradient boosting est donc au final paramétré par les poids de pondération des différents classificateurs faibles, ainsi que par les paramètres des fonctions utilisées. Il s'agit donc d'explorer un espace de fonctions simples par une descente de gradient sur l'erreur.