**Année 2020-2021**

**Rapport de projet E2 – Développeur en Intelligence Artificielle**

Optimisation d’un programme d’I.A existant

« Campagne marketing par une institution bancaire portugaise »

*Etude réalisé par*

Audrey FOUGERE



[1. Introduction 3](#_Toc71480687)

[2. Le modèle 3](#_Toc71480688)

[2.1. Principe 3](#_Toc71480689)

[2.2. Notations 3](#_Toc71480690)

[2.3. Le modèle LOGIT 3](#_Toc71480691)

[3. Analyse du dataset « bank\_full.csv 3](#_Toc71480692)

[3.1. Analyse exploratoire des données 3](#_Toc71480693)

[3.2. Preprocessing de nos données 4](#_Toc71480694)

[4. Construction d’un modèle de classification 4](#_Toc71480695)

[4.1. Partitionnement des données d’entraînement 4](#_Toc71480696)

[4.2. Prédictions sur les échantillons de test 5](#_Toc71480697)

[4.3. Analyse des différentes métriques 5](#_Toc71480698)

[5. Eviter le Surapprentissage/ Overfitting 6](#_Toc71480699)

[5.1. Définition 6](#_Toc71480700)

[6. Réglages du modèle/ Optimisation des hyperparamètres 6](#_Toc71480701)

[6.1. Introduction 6](#_Toc71480702)

[6.2. Evaluer la robustesse de notre modèle par la validation croisée 6](#_Toc71480703)

[6.3. Mise en place d’une meilleure performance de mon modèle 7](#_Toc71480704)

[6.4. Analyse des métriques après Optimisation des Hyperparamètres 7](#_Toc71480705)

[6.5. Sauvegarde de mon modèle de classification 8](#_Toc71480706)

[7. Amélioration de l’application d’intelligence artificielle grâce à l’ajout d’une évolution fonctionnelle 8](#_Toc71480707)

[7.1. Estimation de charge au regard du besoin fonctionnelle 8](#_Toc71480708)

[7.2. Recherche d’une application Flask déjà existante 8](#_Toc71480709)

[7.3. Intégration de mon modèle d’IA dans une application existante 9](#_Toc71480710)

[7.4. Intégration d’une évolution fonctionnelle dans mon modèle 9](#_Toc71480711)

[7.4.1 Création d’une interface de mon application Flask 9](#_Toc71480712)

[7.5. Test de la non-régression de l’application pour donner suite à l’intégration de cette évolution 10](#_Toc71480713)

[8. Annexes 11](#_Toc71480714)

# Introduction

Concernant le projet E2, j’ai choisi le sujet de la campagne marketing en banque étudié dans le cadre du programme Data IA de l’école Simplon Microsoft Nantes. Cet ensemble de données concerne les campagnes de marketing par appel téléphonique direct, qui visent à promouvoir les dépôts à terme auprès des clients existants, par une institution bancaire portugaise de mai 2008 à novembre 2010. Il est accessible au public dans le référentiel UCI Machine Learning, qui peut être récupéré sur le site UCI : <https://archive.ics.uci.edu/ml/datasets/Bank+Marketing>

Le fichier plat utilisé pour faire de l’analyse de données est le suivant : **bank-full.csv**. Cet ensemble de données va être utilisé pour prédire si un nouveau client est susceptible de souscrire à un dépôt à long terme ou non.

Le modèle de Machine Learning utilisé sera un modèle de *classification*, qui est un exemple de technique d'apprentissage automatique supervisé, ce qui signifie qu'elle repose sur des données qui incluent des valeurs de caractéristiques connues ainsi que des valeurs d’étiquettes connues. Un algorithme de classification est utilisé pour adapter un sous-ensemble de données à une fonction qui peut calculer la probabilité pour chaque étiquette de classe à partir des valeurs d'entité. Les données restantes sont utilisées pour évaluer le modèle en comparant les prédictions qu'il génère à partir des entités aux étiquettes de classe connues.

# Le modèle

## Principe

Pour établir des prédictions sur de nouvelles observations, nous avons mis en place un modèle de régression logistique. C’est une méthode d'apprentissage supervisée très utile pour expliquer et prédire une variable discrète lorsque le jeu de données comporte de nombreuses variables catégorielles.

## Notations

Soit Y la variable à prédire / expliquée (dans notre cas, il s'agit de la souscription ou non à l'option de dépôt à long terme) et X = (X1, X2, ..., Xn) les variables explicatives

Dans le cadre de la régression logistique binaire, la variable Y prend deux modalités {0,1}. Les variables Xi sont exclusivement binaires ou continues.

- Soit Ω un ensemble de n échantillons (ici n = 45211) comportant n1 (respectivement n0)) observations correspondant à la modalité 1 (respectivement 0) de Y (dans notre cas, n1 = 5289 et n0 = 39922).

- P(Y=1) est la probabilité a priori pour que Y = 1. Pour simplifier on la notera p(1)

- P(X|1) est la distribution conditionnelle des X sachant la valeur prise par Y.

- La probabilité a posteriori d'obtenir la modalité 1 de Y (resp. 0) sachant la valeur prise par X est p(1|X) (resp. p(0|X))

## Le modèle LOGIT

On désigne par le terme LOGIT de p(1|X) l'expression suivante :

- il s'agit bien d'une régression car on cherche à montrer une relation de dépendance entre une variable à expliquer et une série de variables explicatives

- il s'agit d'une régression logistique car la loi de probabilité est modélisée par une loi logistique (<https://fr.wikipedia.org/wiki/Fonction_logistique_(Verhulst)>)

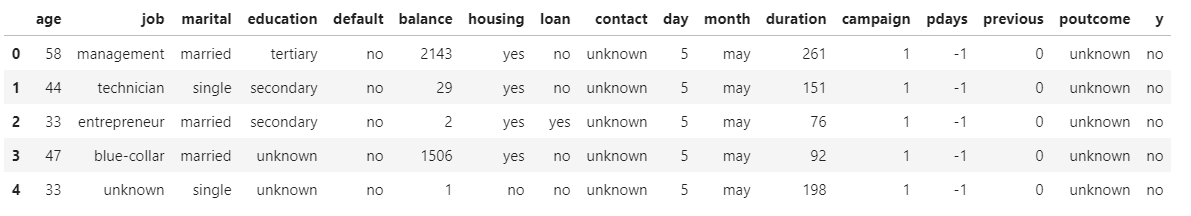
La fonction logistique est intéressante car elle est beaucoup moins sensible aux outliers dans le cas d’une classification binaire

Expliquer / prédire une variable binaire consiste alors à estimer au mieux les coefficients *bi* ci-dessus. Ensuite on peut retrouver la probabilité conditionnelle par la formule suivante :



## Analyse exploratoire des données

Après l’import des librairies : numpy et pandas, j’ai utilisé la méthode *read\_csv* pour convertir le fichier au format csv en dataframe.



On obtient 45211 observations pour 17 variables.

- La variable d'intérêt (cible) est "y", représentant la souscription à un dépôt à terme

- On observe 7 variables numériques : "age", "balance", "day", "duration", "campaign", "pdays" et "previous"

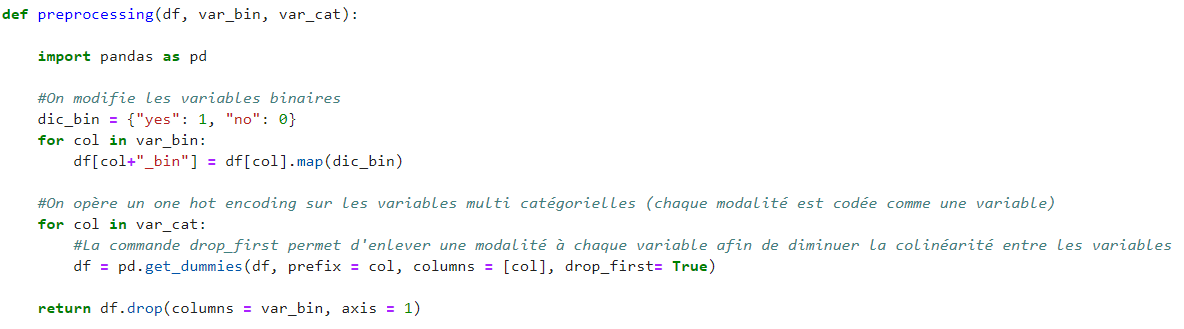
- On observe 10 variables catégorielles :

\* 4 variables binaires : "default", "housing", "loan" et "y"

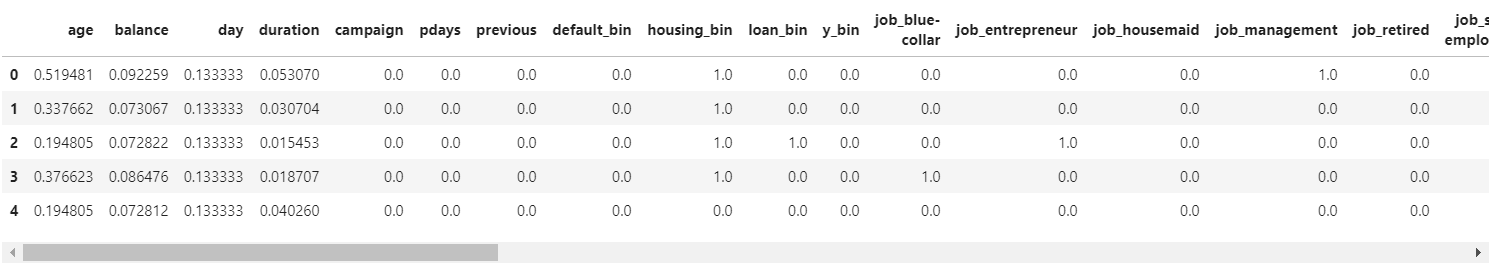
\* 6 variables avec strictement plus de 2 modalités : "job", "marital", "education", contact", "month" et "poutcome"

## Preprocessing de nos données

Ensuite, je suis passée par une étape de prétraitement des données. J’ai besoin de variables exclusivement binaires ou continues pour entrainer mon modèle de régression logistique ; je mets donc en place un one-hot encoding : c’est-à-dire que chaque modalité de variables multi catégorielles est transformée en variable binaire.

Je m’assure de la bonne mise en place de cet encoding avec le script python preprocessing ci-dessous :

J’ai aussi introduit une étape de normalisation de type min max afin d'avoir toutes les variables entre 0 et 1. Le but ici est d’éviter d’avoir un écart trop important entre les valeurs des différentes catégories.



# Construction d’un modèle de classification

## Partitionnement des données d’entraînement

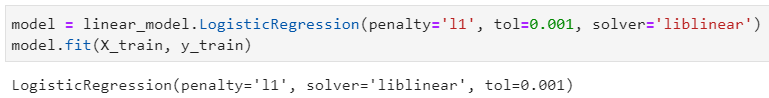
Lors de l’ajustement de mon modèle, je souhaite m’assurer de deux choses :

* Trouver le meilleur modèle (en termes de paramètre)
* Généraliser le modèle : c’est-à-dire le faire fonctionner correctement sur des données invisibles (sur lequel il ne s’est pas entraîné)

Une image contenant table

Description générée automatiquementPendant cette étape, je sépare les données en deux sous-ensembles : les données d’entrainement et les données de test. Mon modèle va s’entrainer sur les données d’entrainement afin de faire des prédictions sur les données de test (données sur lesquelles mon modèle ne s’est pas entraîné).

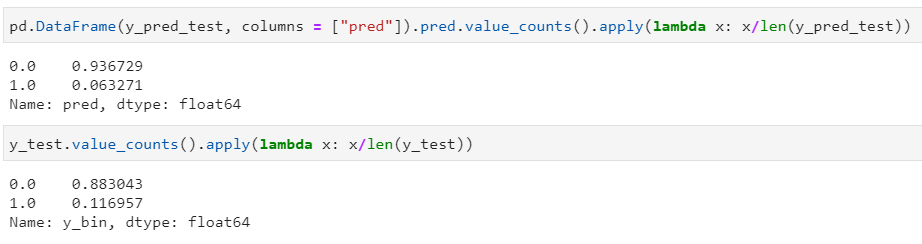
Tout d’abord, je mets en place une régression logistique de base : avec un partitionnement des données de type 70/30 (70% des données seront entraînés contre 30% testées).

Le dataset ne contient pas beaucoup d'observations (seulement quelques dizaines de milliers), on peut donc se dire qu'un solver « liblinear » serait approprié pour résoudre un tel problème.

## Prédictions sur les échantillons de test

En faisant des prédictions sur l'échantillon de test, on se rend compte que le modèle prédit deux fois moins de personnes susceptibles de souscrire qu'il y en a dans l'échantillon. Le modèle a tendance à être "frileux" (prédire moins de 1 que de 0) pour avoir une meilleur accuracy. Il est possible de changer ça en jouant sur l'hyperparamètre class\_weight





## Analyse des différentes métriques

Plusieurs paramètres d’un algorithme de classification nous permettent d’évaluer si un modèle est optimale :

* L’accuracy
* La precision (également appelée valeur predictive positive)
* Le recall (également appelé sensibilité)
* Le F1-score, calculé à partir de la precision et du rappel du test.
* La courbe ROC

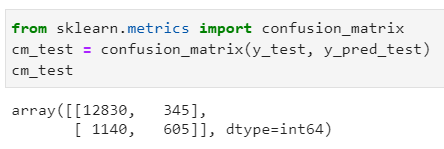
En apprentissage automatique supervisé, la matrice de confusion est une matrice qui mesure la qualité d’un système de classification. Chaque ligne correspond à une classe réelle, chaque colonne correspond à une classe estimée. Un des intérêts de la matrice de confusion est qu’elle montre rapidement si un système de classification parvient à classifier correctement

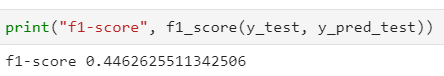
Une image contenant table

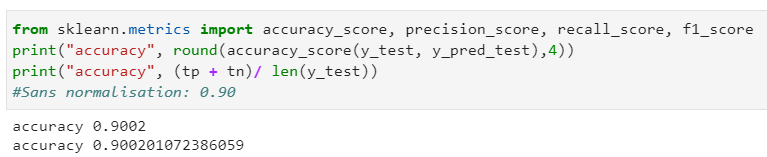
Description générée automatiquement

Regardons de plus près cette matrice

|  |  |
| --- | --- |
|  | Nombre |
| True Negative | 12830 |
| True Positive | 605 |
| False Positive | 345 |
| False Negative | 1140 |







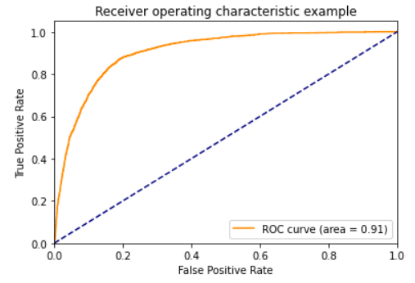
L’accuracy du modèle est de 0.90 et le f1-score de 0.44.

Après calcul, on obtient les résultats suivants :

precision = TP / TP + FP = 605 / (605 + 345) = 0.64

recall = TP / TP + FN = 605 / (605 + 1140) = 0.35

Un autre indicateur qui peut être utilisé pour évaluer notre modèle est la courbe ROC, également appel courbe de sensibilité/spécificité. C’est une mesure de la performance d’un classificateur binaire. Graphiquement, on représente la courbe ROC sous la forme d’une courbe qui donne le taux de vrais positifs en fonction du taux de faux positifs.



**AUC** signifie "aire sous la courbe ROC". Cette valeur mesure l'intégralité de l'aire à deux dimensions situées sous l'ensemble de la courbe ROC (par calculs d'intégrales) de (0) à (1).

Les valeurs d'AUC sont comprises dans une plage de 0 à 1. Un modèle dont 100 % des prédictions sont erronées, a un AUC de 0. Si toutes ses prédictions sont correctes, son AUC est égale à 1. Dans notre cas précis, notre AUC = 0.91

# Eviter le Surapprentissage/ Overfitting

## Définition

Lorsque le dataset est fractionné, il peut y avoir un surapprentissage ou alors un sous-ajustement de notre modèle. L’objectif est bien évidement d’éviter ces deux problèmes d’ajustement. Le surajustement signifie que notre modèle que nous avons formé s’est en quelque sorte « trop bien » entraîné et est maintenant trop ajusté à l’ensemble des données. Il a appris par cœur les données sans vraiment comprendre.

Cela peut se produire quand le modèle est trop complexe (c’est à dire trop de caractéristiques, variables par rapport au nombre d’observations). Ce modèle aura une grande précision pour les données de formation, mais probablement pas assez précis pour de nouvelles données/observations. Quant au sous-ajustement, cela signifie que le modèle ne s’ajuste pas aux données d’apprentissage et manque donc les tendances que peuvent traduire les données. Le modèle ne peut pas non plus être généralisé à de nouvelles données. C’est le résultat d’un modèle trop simple (c’est-à-dire pas assez de variables indépendantes).

Une image contenant texte, horloge

Description générée automatiquementVoici un schéma représentant un surajustement, un sous-ajustement et un modèle « parfait »

Dorénavant, pour pallier ce souci de surapprentissage, on va créer trois bases de données : une base d'apprentissage, une de test et une dernière de validation pour laquelle on est sûr que le modèle n'a jamais vu les données.

# Réglages du modèle/ Optimisation des hyperparamètres

## Introduction

Tout d’abord, Les hyper-paramètres sont en fait les paramètres d’ajustements des différents algorithmes de Machine Learning. Afin de les optimiser, je vais rechercher le "meilleur" set d'hyperparamètres pour mon modèle de régression logistique. J’ai essayé de chercher le meilleur arbitrage possible entre performance et robustesse du modèle. La performance consiste à avoir des scores élevés pour les différents métriques testées et la robustesse signifie avoir des métriques similaires au cours des différents apprentissages de la cross validation. Lors de l'optimisation des hyperparamètres, il est possible de spécifier le score que l'on cherche à maximiser. Il semblerait qu'il y ait deux stratégies possibles pour la banque :

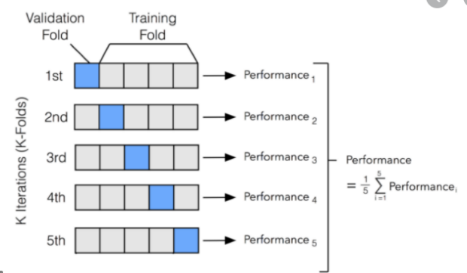
- contacter un maximum de personnes pour avoir le plus de souscriptions possibles (avoir le recall le plus grand possible)

- appeler un nombre limité de clients mais particulièrement bien ciblés (avoir la meilleure précision possible).

N'ayant pas d'indications concernant la stratégie que la banque cherche à adopter, j’ai choisi de maximiser le **f1-score**.

## Evaluer la robustesse de notre modèle par la validation croisée

Il est acquis qu’un modèle doit être évalué sur une base de test différente de celle utilisée pour l’apprentissage. Mais la performance est peut-être juste l’effet d’une aubaine et d’un découpage particulièrement avantageux. Pour être sûr que le modèle est robuste, on recommence plusieurs fois. On appelle cela la validation croisée ou cross validation. Concernant notre modèle, nous découpons la base de données en cinq segments de façon aléatoire. On en utilise quatre pour l’apprentissage et un pour le tester. On recommence cinq fois. Si le modèle est robuste, les cinq scores de test seront sensiblement égaux.



## Mise en place d’une meilleure performance de mon modèle

Afin de régler les hyperparamètres de mon algorithme de classification le plus performant, c’est-à-dire la régression logistique ; j’ai utilisé la librairie scikit-learn qui fournit la fonction de recherche par quadrillage : GridSearchCV. J’ai positionné une liste de possibilité pour chacun des hyperparamètres et j’ai entraîné mon modèle sur chacune des combinaisons pour calculer son score. A la fin, nous conservons uniquement le meilleur paramétrage.C’est une technique intéressante et performante mais qui a un très gros inconvénient : il faut être patient car mon modèle est entraîné sur toutes les combinaisons. Cependant, l’avantage, c’est que je n’ai à le faire qu’une seule fois.

J’ai consulté cette documentation qui liste tous les hyperparamètres que peut utiliser le modèle de régression logistique : <https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html>

J’ai joué sur quatre hyperparamètres qui me semblaient pertinents :

* le solver (algorithme à utiliser dans le problème d'optimisation)
* la pondération des classes (poids associés aux classes)
* la pénalité ou régularisation (utilisé pour spécifier la norme utilisée dans la pénalisation.)
* le C (il contrôle la force de pénalité)

Pour faire ce « Grid Search » avec Scikit-Learn, il faut dans un premier temps créer un dictionnaire Python (ici appelé « grid ») avec les hyperparamètres sélectionnés et dans un second temps paramétrer les valeurs que nous voulons tester. J’ai mis en place une validation croisée à cinq folds représenté par cv=5.

Voici le résultat de notre tuning d’hyperparamètres avec la fonction GridSearchCV



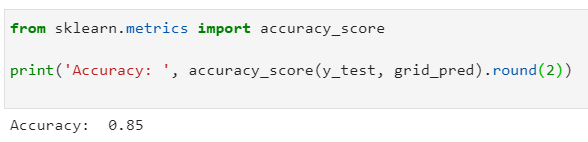
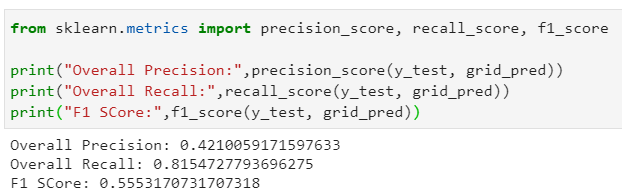
On remarque que notre cible « f1 » est passé de 0.45 à 0.55, on a gagné 0.10 en f1-score ce qui est conséquent.

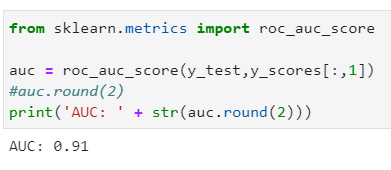
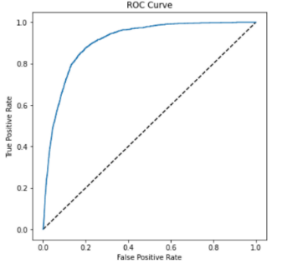
Notre meilleur set d’hyper paramètres se compose de :

C = 1.0, class\_weight = « balanced », penalty = « l1 » et solver = « liblinear »

## Analyse des métriques après Optimisation des Hyperparamètres

Voici une synthèse des métriques : on constate une légère diminution de notre performance du modèle avec une diminution de 0.05. En revanche notre f1-score a augmenté de 0.10 et notre AUC est équivalente.





.

**Tableau récapitulatif des valeurs des paramètres de notre modèle de régression logistique avant et après optimisation des hyperparamètres**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Résultat | Accuracy | Precision | Recall | F1-score | AUC |
| Avant optimisation des hyperparamètres | 0.90 | 0.64 | 0.35 | 0.45 | 0.91 |
| Après optimisation des hyperparamètres | 0.85 | 0.42 | 0.81 | 0.55 | 0.91 |

Après intégration des nouveaux hyperparamètres et test de notre algorithme de régression logistique.

L’accuracy a légèrement diminué certainement due à la précision qui est moins bonne qu’avant l’optimisation des hyperparamètres. Cependant, notre stratégie, qui était de maximiser le f1-score, a bel et bien fonctionné. Nous avons gagné 0.10 pour ce paramètre, ce qui est conséquent.

Les modifications intégrées ont effectivement amélioré le niveau de performance de l’intelligence artificielle.

## Sauvegarde de mon modèle de classification

On effectue une sauvegarde de notre modèle à l’aide de la librairie pickle ; cette librairie permet de sauvegarder des structures de données dans un fichier pour pouvoir les recharger plus tard. Dans notre exemple, notre modèle est sauvegardé au format pickle (Le module pickle permet la sérialisation des objets mémoire en chaînes (et réciproquement). Il est utile pour la persistance et le transfert de données sur un réseau. La manipulation des données utilise un fichier par sérialisation, où les données sont stockées sous forme de chaînes.)



# Amélioration de l’application d’intelligence artificielle grâce à l’ajout d’une évolution fonctionnelle

## Estimation de charge au regard du besoin fonctionnelle

Au regard du besoin fonctionnelle, j’ai constitué une estimation de charge avec l’outil Microsoft Excel et un modèle de Gantt prédéfinis que je vous ai mis en annexe (cf : diagramme\_gantt\_projet\_e2.xlsx)

J’ai comptabilisé sept user story, qui comprend :

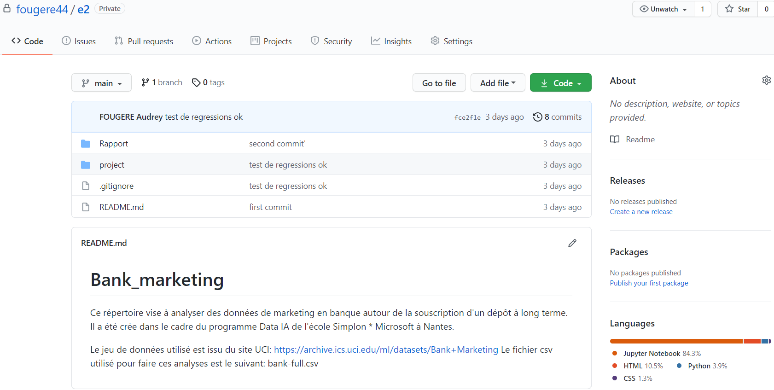
* Recherche d’une application Flask déjà existante (1 jour)
* Intégration de mon modèle de régression logistique dans l’application (1 jour)
* Conversion des variables binaires et catégorielle en 0 ou 1 (1 jour)
* Mise en place d’une interface de mon application Flask (1 jour)
* Mise en place de de nouvelles fonctionnalités de type boutons : « Reset », « Refresh », « Predict », « Carte result » et « % Prediction ». (1 jour)
* Phase de test (2 jours)

La charge totale comptabilisée est de sept jours.

## Recherche d’une application Flask déjà existante

J’ai repris une application existante disponible à cette adresse : <https://github.com/DheerajKumar97/Liver-Cancer-Prediction-With-Flask-Deployment.git> Celle-ci a été créée avec le micro Framework Flask.

J’ai créé un repository public sur mon compte GitHub que vous pouvez aller explorer en suivant ce lien 🡪 <https://github.com/fougere44/e2.git>



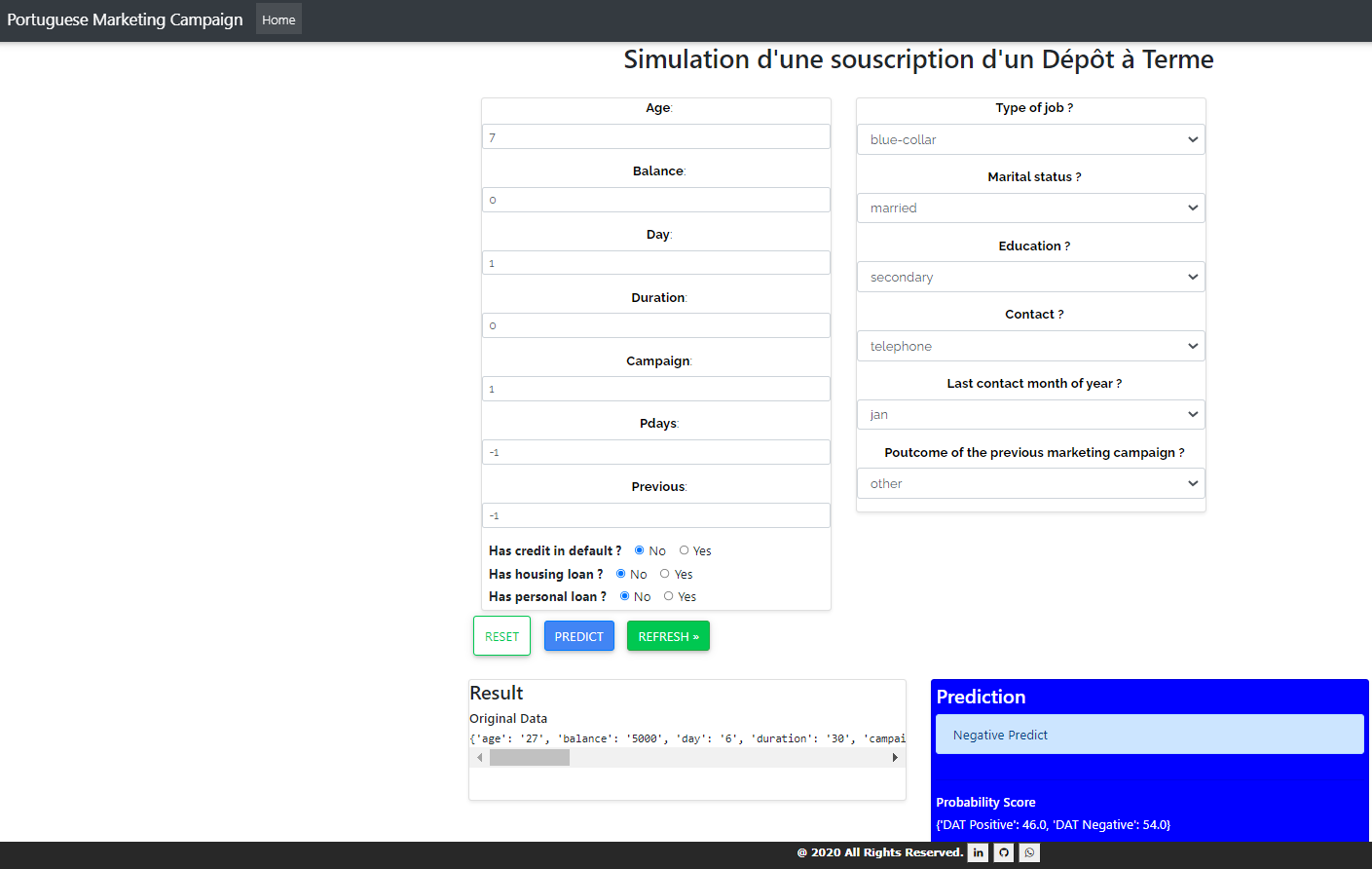
## Intégration de mon modèle d’IA dans une application existante

Comment ai-je intégré mon modèle de régression logistique ? Grâce à l’import des librairies os et pickle, j’ai développé une fonction load\_model()



## Intégration d’une évolution fonctionnelle dans mon modèle

### 7.4.1 Création d’une interface de mon application Flask



v

v

v

Ajout de nouvelles fonctionnalités

+

Création d’une interface web

TODO : Faire GIF de l’application.

La première évolution concernant la mise en place de l’interface de mon application Flask a été effectuée avec la méthode render\_template() de la librairie Flask. Cette méthode renvoie au fichier html « index.html » mise en annexe.

La deuxième évolution concerne les boutons Reset, Predict et Refresh :

Les éléments [<input>](https://developer.mozilla.org/fr/docs/Web/HTML/Element/Input) de type **"RESET"** sont affichés sous la forme de boutons permettant de réinitialiser l'ensemble des champs du formulaire avec leurs valeurs initiales.

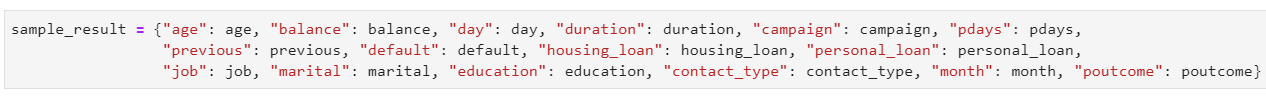


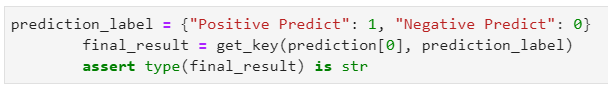
L’élément REFRESH a pour href = « / », ce qui signifie qui fois qu’on clique sur refresh, on revient à la page principale de notre application web

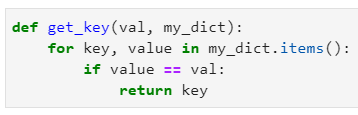


L’élément Predict est de type submit, ce qui veut dire qu’en ayant auparavant rempli les champs du formulaire et une fois cliqué sur le bouton « Predict », cela nous fait passer par la fonction predict() qui envoie une requête POST avec tous les éléments renseignés du formulaire : celle-ci est utilisée pour envoyer une entité vers la ressource indiquée. Cela entraîne généralement un changement d'état ou des effets de bord sur le serveur. Le serveur renvoie une requête GET qui permet de récupérer les données ; en l’occurrence, on cherche à obtenir une prédiction positive ou négative pour la souscription du dépôt à terme de la nouvelle observation renseignée.

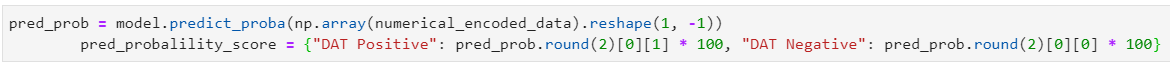
L’encart Result renvoie le {{ sample\_result }} ; le sample\_result est défini comme un dictionnaire { clé : valeur } ; ou la clé est le nom d’une catégorie et la valeur est la valeur associé à cette catégorie.



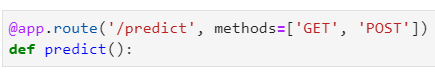
L’encart Prediction renvoie dans un premier temps la prédiction qui sera soit négative (souscription au dépôt à terme négative) ou alors positive (souscription au DAT positive)



Ce même encart renvoie dans un second temps les deux prédictions : l’une positive et l’autre négative retournée sous forme de pourcentage.



Pour résumer, la fonction predict renvoie ce render\_template :





## Test de la non-régression de l’application pour donner suite à l’intégration de cette évolution

Par suite de l’ajout de ces différentes fonctionnalités, et afin de tester la non-régression de l’application, j’ai installé le Framework pytest permettant de faire des tests et de vérifier si les différentes conditions sont juste ou fausse. Il permet de tester les éléments un à un mais on peut aussi lui demander de faire une série de tests. Ces méthodes de test dépendent de comment on implémente dans notre code.

De plus, j’ai importé la librairie unittest qui est une méthode de test logiciel par laquelle des unités individuelles de code source sont soumises à divers tests pour déterminer si elles sont aptes à être utilisées. Il détermine et vérifie la qualité de mon code.



Le premier test faisant partie de la classe **TestConfig**, est défini par une méthode donc le nom commence par les lettres test ; dans notre cas, elle s’appelle *test\_config\_loading()*. Cette convention de nommage signale au lanceur de tests quelles méthodes sont des tests.

Le cœur de chaque test est un appel à [assertEqual()](https://docs.python.org/fr/3/library/unittest.html#unittest.TestCase.assertEqual) pour vérifier un résultat attendu ; [assertTrue()](https://docs.python.org/fr/3/library/unittest.html#unittest.TestCase.assertTrue) ou [assertFalse()](https://docs.python.org/fr/3/library/unittest.html#unittest.TestCase.assertFalse) pour vérifier une condition. Ces méthodes sont utilisées à la place du mot-clé [assert](https://docs.python.org/fr/3/reference/simple_stmts.html#assert) pour que le lanceur de tests puisse récupérer les résultats de tous les tests et produire un rapport.

La méthode *test\_config\_loading()* vérifie dans ce cas présent que le DEBUG de mon application Flask est égale à True.



Le deuxième test faisant partie de la classe **FlaskTestCase**, est défini par deux méthodes : *test\_index()* et *test\_index\_data().*

La méthode *test\_index()* fait appel à un assertEqual (elle vérifie un résultat attendue) :

Lorsqu’en amont j’effectue les actions suivantes

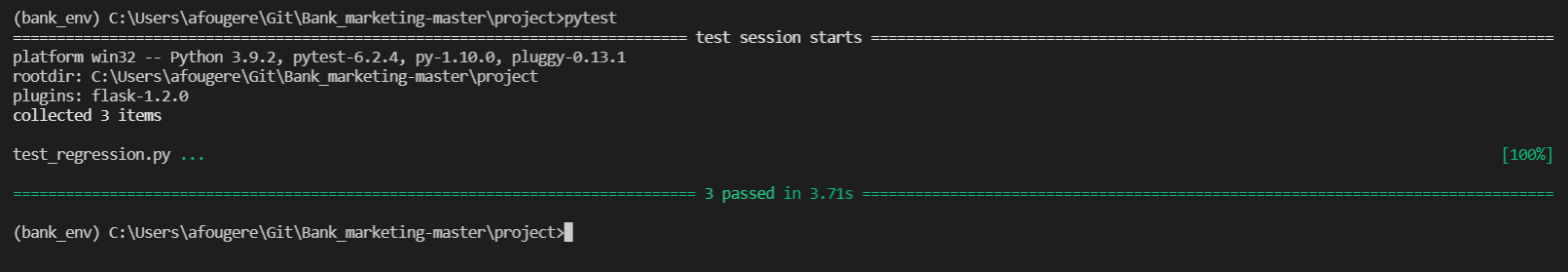
* Set FLASK\_APP=app
* Set FLASK\_ENV=development
* Flask run

Je clique ensuite sur <http://127.0.0.1:5000/> ; si j’arrive sur la page principale de mon application, alors mon application Flask renvoie un statut de réponse 200 ok, ceci indique la réussite de ma requête qui était d’accéder à mon application via le serveur localhost :5000/.

La méthode *test\_index\_data()* fait appel à un assertTrue (elle vérifie une condition) :

La condition en question est : s’il y a le mot « Predict » dans la réponse de la prédiction, alors la condition est vérifiée.

Afin de lancer tous les tests de mon projet d’analyse de marketing, je tape dans mon terminal la commande pytest.

Pytest va alors lancer tous les tests de mon projet

Tout est vert, mes tests sont bien validés !

# Annexes

diagramme\_gantt\_projet\_e2.xlsx

app.py

index.html

style.css