

Université de Tunis

**Institut supérieure de gestion de Tunis**



Matière  
**PML**

Professeur  
**Dr. Mariem Jamel**

Rapport de Sujet de  
**Prévision d'approbation de carte de credit**

Elaboré par:  
**FREDJ Imen**



## Demandes de cartes de crédit :

Les banques commerciales reçoivent de nombreuses demandes de cartes de crédit. Beaucoup d'entre elles sont rejetées pour diverses raisons, telles que des soldes de prêts élevés, des niveaux de revenus faibles, ou trop d'interrogations sur le rapport de crédit d'un individu, par exemple. Analyser manuellement ces demandes est fastidieux, sujet à des erreurs et prend beaucoup de temps. Heureusement, cette tâche peut être automatisée grâce à la puissance du Machine Learning, et pratiquement toutes les banques commerciales le font de nos jours. Dans notre projet, nous allons construire un prédicteur automatique d'approbation de carte de crédit en utilisant des techniques d'apprentissage automatique, tout comme le font les vraies banques.

Nous utiliserons l'ensemble de données d'approbation de cartes de crédit provenant du référentiel UCI Machine Learning. La structure de ce notebook est la suivante :

- Tout d'abord, nous commencerons par charger et visualiser l'ensemble de données.
- Nous verrons que l'ensemble de données comporte un mélange de caractéristiques numériques et non numériques, qu'il contient des valeurs de différentes plages, et qu'il contient un certain nombre d'entrées manquantes.
- Nous devons prétraiter l'ensemble de données pour nous assurer que le modèle d'apprentissage automatique que nous choisissons puisse faire de bonnes prédictions.
- Une fois nos données en bon état, nous effectuerons une analyse exploratoire des données pour construire nos intuitions.
- Enfin, nous construirons un modèle d'apprentissage automatique capable de prédire si la demande de carte de crédit d'un individu sera acceptée.

## Chargement et visualisation de l'ensemble de données :

```
# Import pandas
import pandas as pd

# Load dataset
cc_apps = pd.read_csv("datasets/Application_Data.csv", header=None)

# Get the shape of the dataset
dimensions = cc_apps.shape

print("Dimensions of the dataset:", dimensions)

# Inspect data
cc_apps.head()
```

Dimensions of the dataset: (25129, 21)

Comme indiqué dans la capture on peut distinguer que notre ensemble de données comprend 25 129 lignes et 21 colonnes, fournissant une ampleur significative pour notre analyse.

Notre dataset comporte les caractéristiques suivantes :

- Pièce d'identité du demandeur
- Sexe du demandeur
- Voiture détenue
- Propriété immobilière
- Total des enfants
- Revenu total
- Type de revenu
- Type de formation
- Situation de famille
- Type de logement
- Titre du poste
- Téléphone mobile détenu
- Téléphone de travail personnel
- Téléphone personnel
- Courriel de la propriété

- Total des membres de la famille
- Âge du demandeur
- Années de travail
- Total des mauvaises créances
- Total de la bonne dette

Ces caractéristiques sont les probables dans une demande de carte de crédit typique. Cela nous donne un point de départ assez solide, et nous pouvons faire correspondre ces caractéristiques par rapport aux colonnes de la sortie.

	0	1	2	3	4	5	6	7	8	9	...
0	Applicant_ID	Applicant_Gender	Owned_Car	Owned_Realty	Total_Children	Total_Income	Income_Type	Education_Type	Family_Status	Housing_Type	...
1	5008806	M	1	1	0	112500	Working ...	Secondary / secondary special ...	Married ...	House / apartment ...	...
2	5008808	F	0	1	?	270000	Commercial associate ...	Secondary / secondary special ...	Single / not married ...	House / apartment ...	...
3	5008809	F	0	1	0	270000	Commercial associate ...	Secondary / secondary special ...	Single / not married ...	House / apartment ...	...
4	5008810	F	?	1	0	270000	Commercial associate ...	Secondary / secondary special ...	Single / not married ...	House / apartment ...	...

Comme nous pouvons le voir à première vue des données, l'ensemble de données comporte un mélange de caractéristiques numériques et non numériques. Cela peut être corrigé avec un prétraitement, mais avant de le faire, apprenons un peu plus sur l'ensemble de données pour voir s'il existe d'autres problèmes à résoudre.

## Inspection des applications

```
# Print summary statistics
cc_apps_description = cc_apps.describe()
print(cc_apps_description)

print("\n")

# Print DataFrame information
cc_apps_info = cc_apps.info()
print(cc_apps_info)
```

```

count      0      1      2      3      4      5  \<class 'pandas.core.frame.DataFrame'>
unique      25129   25129  25129  25129  25129  25129  RangeIndex: 25128 entries, 0 to 25127
top      Applicant_ID  F      0      1      0  135000  Data columns (total 21 columns):
freq      1    15625  14611  16448  15901    3006  #   Column              Non-Null Count  Dtype
                                     ---  ---
count      6  \
unique      25129
top      Working      ...
freq      15616

count      7  \
unique      25129
top      Secondary / secondary special      ...
freq      16802

count      8  \
unique      25129
top      Married      ...
freq      17507

                                     0   Applicant_ID  25128 non-null  int64
                                     1   Applicant_Gender  25128 non-null  object
                                     2   Owned_Car  25128 non-null  int64
                                     3   Owned_Realty  25128 non-null  int64
                                     4   Total_Children  25128 non-null  int64
                                     5   Total_Income  25128 non-null  int64
                                     6   Income_Type  25128 non-null  object
                                     7   Education_Type  25128 non-null  object
                                     8   Family_Status  25128 non-null  object
                                     9   Housing_Type  25128 non-null  object
                                    10   Owned_Mobile_Phone  25128 non-null  int64
                                    11   Owned_Work_Phone  25128 non-null  int64
                                    12   Owned_Phone  25128 non-null  int64
                                    13   Owned_Email  25128 non-null  int64
                                    14   Job_Title  25128 non-null  object
                                    15   Total_Family_Members  25128 non-null  int64
                                    16   Applicant_Age  25128 non-null  int64
                                    17   Years_of_Working  25128 non-null  int64
                                    18   Total_Bad_Debt  25128 non-null  int64
                                    19   Total_Good_Debt  25128 non-null  int64
                                    20   Status  25128 non-null  int64
dtypes: int64(15), object(6)

```

Nous avons identifié quelques problèmes qui affecteront les performances de notre modèle(s) d'apprentissage automatique s'ils ne sont pas résolus :

1. Notre ensemble de données contient à la fois des données numériques et non numériques (en particulier des données de types float64, int64 et object). Plus précisément, les caractéristiques 1, 6, 7, 8, 9 et 10 contiennent des valeurs non numériques (de types float64, float64, int64 et int64 respectivement), et toutes les autres caractéristiques contiennent des valeurs numériques.
2. L'ensemble de données contient également des valeurs provenant de plusieurs plages. Certaines caractéristiques ont une plage de valeurs de 0 et 1, d'autres ont une plage de 45000 à 1350000, et certaines ont une plage de 21 à 61. En plus de cela, nous pouvons obtenir des informations statistiques utiles (comme la moyenne, le maximum et le minimum) sur les caractéristiques qui ont des valeurs numériques.

3. Enfin, l'ensemble de données comporte des valeurs manquantes, que nous allons prendre en charge dans cette tâche. Les valeurs manquantes dans l'ensemble de données sont étiquetées par '?', comme on peut le voir dans la sortie de la dernière cellule.

	2	3	4
Car	Owned_Realty	Total_Children	
1	1	0	
0	1	?	
0	1	0	
?	1	0	

## Gestion des valeurs manquantes (partie I)

Maintenant, remplaçons temporairement ces points d'interrogation représentant des valeurs manquantes par NaN.

```
# Import numpy
import numpy as np

# Inspect missing values in the dataset
print(cc_apps.isnull().values.sum())

# Replace the '?'s with NaN
cc_apps = cc_apps.replace('?', np.nan)

# Inspect the missing values again
cc_apps.tail(17)
```

0

Married ...	House / apartment ...	...	1	0	0	0	3	30	5	0	18	1
Married ...	House / apartment ...	...	NaN	0	0	0	3	30	5	0	NaN	1
Married ...	House / apartment ...	...	1	0	0	0	3	30	5	0	13	1
Married ...	House / apartment ...	...	1	0	0	0	3	30	5	0	2	1
Married ...	House / apartment ...	...	1	0	0	0	2	54	6	0	30	1

Nous avons remplacé tous les points d'interrogation par NaN. Cela va nous aider dans le prochain traitement des valeurs manquantes que nous allons effectuer.

## Gestion des valeurs manquantes (partie II)

Une question importante qui se pose ici est pourquoi accordons-nous autant d'importance aux valeurs manquantes ? Ne peuvent-elles pas simplement être ignorées ? Ignorer les valeurs manquantes peut fortement affecter les performances d'un modèle d'apprentissage automatique. En ignorant les valeurs manquantes, notre modèle peut passer à côté d'informations utiles pour son entraînement.

Pour éviter ce problème, nous allons imputer les valeurs manquantes des valeurs numériques avec une stratégie appelée imputation par la moyenne.

```
# Separate Numeric and Non-Numeric Columns
numeric_cols = cc_apps.select_dtypes(include='number')
non_numeric_cols = cc_apps.select_dtypes(exclude='number')

cc_apps[numeric_cols.columns] = cc_apps[numeric_cols.columns].fillna(cc_apps[numeric_cols.columns].mean())

# Count the number of NaNs in the dataset to verify
print(cc_apps.isnull().values.sum())
```

173

Tout d'abord, on a séparé les valeurs numériques des valeurs catégoriques. Puis, nous avons pris en charge avec succès les valeurs manquantes présentes dans les colonnes numériques.

Il reste encore des valeurs manquantes à imputer pour les colonnes contenant des données non numériques. La stratégie d'imputation par la moyenne ne fonctionnerait pas ici. Cela nécessite un traitement différent.

## Gestion des valeurs manquantes (partie III)

Nous allons imputer ces valeurs manquantes avec les valeurs les plus fréquentes présentes dans les colonnes respectives. C'est une bonne pratique lorsqu'il s'agit d'imputer des valeurs manquantes pour des données catégorielles en général.

```
# Iterate over each column of cc_apps
for col in cc_apps.columns:
    # Check if the column is of object type
    if cc_apps[col].dtypes == 'object':
        # Impute with the most frequent value
        cc_apps = cc_apps.fillna(cc_apps[col].value_counts().index[0])

# Count the number of NaNs in the dataset and print the counts to verify
print(cc_apps.isnull().values.sum())
```

0

Les valeurs manquantes sont maintenant correctement gérées.

Il reste encore un prétraitement mineur mais essentiel des données avant de passer à la construction de notre modèle. Nous allons diviser ces étapes de prétraitement restantes en trois tâches principales :

1. Convertir les données non numériques en données numériques.
2. Diviser les données en ensembles d'entraînement et de test.
3. Mettre à l'échelle les valeurs des caractéristiques dans une plage uniforme.

## **Conversion des données non numériques en données numériques :**

Tout d'abord, nous allons convertir toutes les valeurs non numériques en valeurs numériques. Nous le faisons car non seulement cela résulte en un calcul plus rapide, mais de nombreux modèles d'apprentissage automatique (surtout ceux développés avec scikit-learn) exigent que les données soient dans un format strictement numérique. Nous ferons cela en utilisant une technique appelée encodage des libellés (label encoding).





## Mis à l'échelle des valeurs des caractéristiques dans une plage uniforme (Scaling) :

Les données sont maintenant divisées en deux ensembles distincts - les ensembles d'entraînement et de test respectivement. Il ne nous reste plus qu'une dernière étape de prétraitement, l'échelonnage, avant de pouvoir ajuster un modèle d'apprentissage automatique aux données.

```
# Import MinMaxScaler
from sklearn.preprocessing import MinMaxScaler
# Instantiate MinMaxScaler and use it to rescale X_train and X_test
scaler = MinMaxScaler(feature_range=(0,1))
rescaledX_train = scaler.fit_transform(X_train)
rescaledX_test = scaler.fit_transform(X_test)
```

## Ajustement d'un modèle de régression logistique sur l'ensemble d'entraînement

L'objectif central de notre projet est de prédire si une demande de carte de crédit sera approuvée ou non, une tâche qui relève de la classification. Dans cette perspective, un bon modèle d'apprentissage automatique devrait être capable de prédire avec précision le statut des demandes par rapport à ces statistiques.

Pour débiter notre modélisation d'apprentissage automatique, nous optons pour un modèle de régression logistique, un choix courant pour les tâches de classification, étant un modèle linéaire généralisé.

```
# Import LogisticRegression
from sklearn.linear_model import LogisticRegression

# Instantiate a LogisticRegression classifier with default parameter values
logreg = LogisticRegression(solver='sag',max_iter=1000)

# Fit logreg to the train set
logreg.fit(rescaledX_train, y_train)
```

```
▼ LogisticRegression
LogisticRegression(max_iter=1000, solver='sag')
```

## Effectuer des prédictions et évaluer les performances

Nous allons maintenant évaluer notre modèle sur l'ensemble de test en ce qui concerne la précision de la classification. Mais nous examinerons également la matrice de confusion du modèle. Dans le cas de la prédiction des demandes de carte de crédit, il est tout aussi important de voir si notre modèle d'apprentissage automatique est capable de prédire le statut d'approbation des demandes refusées qui ont été initialement refusées. Si notre modèle ne performe pas bien à cet égard, il pourrait finir par approuver une demande qui aurait dû être refusée. La matrice de confusion nous aide à voir la performance de notre modèle sous ces aspects.

```
# Import confusion_matrix
from sklearn.metrics import confusion_matrix

# Use Logreg to predict instances from the test set and store it
y_pred = logreg.predict(rescaledX_test)

print(y_pred)

# Get the accuracy score of Logreg model and print it
print("Accuracy of logistic regression classifier: ", logreg.score(rescaledX_test, y_test))

# Print the confusion matrix of the Logreg model
print(confusion_matrix(y_test, y_pred))

[1 1 1 ... 1 1 1]
Accuracy of logistic regression classifier:  0.9429639454962017
[[ 0 473]
 [ 0 7820]]
```

Notre modèle était plutôt bon ! Il a réussi à obtenir un score de précision de presque 95 %.

Pour la matrice de confusion, le premier élément de la première ligne de la matrice de confusion représente les vrais négatifs, c'est-à-dire le nombre d'instances négatives (demandes refusées) prédites correctement par le modèle. Et le dernier élément de la deuxième ligne de la matrice de confusion représente les vrais positifs, c'est-à-dire le nombre d'instances positives (demandes approuvées) prédites correctement par le modèle.

## Grid Searching et amélioration des performances du modèle

Voyons si nous pouvons faire mieux. Nous pouvons effectuer une recherche en grille des paramètres du modèle pour améliorer la capacité du modèle à prédire les approbations de cartes de crédit.

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Define the grid of values for tol and max_iter
tol = [0.01, 0.001, 0.0001]
max_iter = [100, 150, 200]

# Create a dictionary where tol and max_iter are key
param_grid = dict(tol=tol, max_iter=max_iter)
```

### Recherche du meilleur modèle performant

Nous avons défini la grille des valeurs des hyperparamètres et les avons converties en un seul format de dictionnaire, comme le souhaite GridSearchCV() en tant que l'un de ses paramètres. Maintenant, nous allons commencer la recherche en grille pour voir quelles valeurs sont les meilleures.

```
# Instantiate GridSearchCV with the required parameters
grid_model = GridSearchCV(estimator=logreg, param_grid=param_grid, cv=5)

# Use scaler to rescale X and assign it to rescaledX
rescaledX = scaler.fit_transform(X)

# Fit data to grid_model
grid_model_result = grid_model.fit(rescaledX, y)

# Summarize results
best_score, best_params = [ grid_model_result.best_score_,
                             grid_model_result.best_params_ ]
print("Best: %f using %s" % (best_score, best_params))
```

Best: 0.948068 using {'max\_iter': 100, 'tol': 0.01}

Nous allons instancier GridSearchCV() avec notre modèle logreg précédent en utilisant toutes les données dont nous disposons. Au lieu de fournir séparément les ensembles

d'entraînement et de test, nous fournirons X (version échelonnée) et y. Nous indiquerons également à GridSearchCV() d'effectuer une validation croisée sur cinq plis.

Nous terminerons en enregistrant le meilleur score obtenu et les meilleurs paramètres respectifs.

## Ajustement d'un modèle Random Forest sur l'ensemble d'entraînement

Commençons par entraîner notre modèle Random Forest.

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators=501)
rf.fit(X_train, y_train)
```

```
▼      RandomForestClassifier
RandomForestClassifier(n_estimators=501)
```

## Effectuer des prédictions et évaluer les performances

Nous allons maintenant évaluer notre modèle sur l'ensemble de test, en particulier en termes de précision de la classification.

```
from sklearn.metrics import confusion_matrix

# Use logreg to predict instances from the test set and store it
Y_pred_rf = rf.predict(rescaledX_test)
print(Y_pred_rf)

# Get the accuracy score of Random fores model and print it
print("Accuracy of Random forest classifier: ", rf.score(rescaledX_test, y_test))

# Get the accuracy score of logreg model and print it
print("Confusion matrix rf \n", confusion_matrix(y_test, Y_pred_rf))
```

```
[1 1 1 ... 1 1 1]
Accuracy of Random forest classifier:  0.9429639454962017
Confusion matrix rf
[[ 0 473]
 [ 0 7820]]
```

Notre modèle s'est également révélé performant ! Il a réussi à atteindre un score de précision presque similaire à celui du modèle de régression linéaire.

## Ajustement d'un modèle d'arbre de décision sur l'ensemble d'entraînement

Commençons par entraîner le modèle :

```
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier()
dt.fit(X_train,y_train)
```

```
▼ DecisionTreeClassifier
DecisionTreeClassifier()
```

## Effectuer des prédictions et évaluer les performances

Nous allons maintenant évaluer notre modèle sur l'ensemble de test.

```
Y_pred_dt = dt.predict(X_test)
from sklearn.metrics import recall_score, precision_score, accuracy_score, confusion_matrix, classification_report
print("Confusion matrix Decision Tree \n", confusion_matrix(y_test, Y_pred_dt))
print('Accuracy DT',accuracy_score(y_test,Y_pred_dt))
print('recall DT',recall_score(y_test,Y_pred_dt))
print('Precision DT',precision_score(y_test,Y_pred_dt))
```

```
Confusion matrix Decision Tree
[[ 273  200]
 [ 171 7649]]
Accuracy DT 0.9552634752200652
recall DT 0.9781329923273657
Precision DT 0.9745190470123583
```

Cette fois-ci, notre modèle a présenté des résultats encore meilleurs !

## Choix du Modèle pour le Déploiement :

Tous les modèles que nous avons testés ont affiché de bons résultats. Cependant, pour la phase de déploiement, nous opterons pour le modèle d'arbre de décision en raison de ses performances supérieures.

## Liaison entre l'Interface Graphique et le Modèle :

Cette phase revêt une importance critique dans notre projet, et son établissement est indispensable. La liaison entre ces deux composants est réalisée de la manière suivante :

1. Sauvegarder le modèle d'arbre de décision dans le fichier 'Decision\_tree\_model.pkl' situé dans le même répertoire que le fichier de l'interface.

```
import joblib
joblib.dump(dt, 'Decision_tree_model.pkl')

['Decision_tree_model.pkl']
```

2. Mise en place d'une fonction pour prétraiter les données entrées par l'utilisateur, les préparant ainsi à être utilisées ultérieurement par le modèle.

```
#function to process the user data
def preprocess_input(user_input):

    # Create a DataFrame from the user input
    user_df = pd.DataFrame([user_input])

    # Handle missing values
    user_df = user_df.replace('?', np.nan)

    # Separate Numeric and Non-Numeric Columns
    numeric_cols = user_df.select_dtypes(include='number')
    non_numeric_cols = user_df.select_dtypes(exclude='number')

    user_df[numeric_cols.columns] = user_df[numeric_cols.columns].fillna(u

    # Impute non-numeric missing values with the most frequent value
    for col in user_df.columns:
        if user_df[col].dtypes == 'object':
            user_df = user_df.fillna(user_df[col].value_counts().index[0])

    # Use LabelEncoder to transform non-numeric columns
    for col in user_df.columns:
        if user_df[col].dtypes == 'object':
            user_df[col] = le.fit_transform(user_df[col])

    # Use MinMaxScaler to scale the features
    rescaled_user_input = scaler.fit_transform(user_df)

    return rescaled_user_input
```

3. Développement d'une fonction qui permet de prédire l'acceptation ou le refus de la demande de l'utilisateur.

```
def predict_credit_approval(user_input):  
  
    # Preprocess the user input  
    preprocessed_input = preprocess_input(user_input)  
  
    # Make predictions using the loaded model  
    prediction = dt.predict(preprocessed_input)  
  
    return prediction
```

4. Chargement du modèle depuis le fichier 'Decision\_tree\_model.pkl' dans le fichier de l'interface :

```
# Load the pre-trained model  
model = joblib.load('Decision_tree_model.pkl')
```

## Le déploiement :

Le modèle a été déployé en utilisant Streamlit, comme illustré dans les captures ci-dessous:

The application is titled "Credit Card Approval Predictor". It prompts the user to "Fill in the details below to predict credit approval."

**Identity:**

- ID: 5000000
- Gender: ☒ Male, ☐ Female

**Demographic infos:**

- Age: 25 (range 18 to 100)
- Marital Status: Single / not married
- working years: 20

**Model Details:**

- Model: Decision Tree Classifier
- Accuracy: 0.93

**Financial Info:**

- Owned Car: 0
- Total Children: 0
- Total Family Members: 1
- Housing Type: House / apartment
- Income Type: Working

**Total Income:** 926705



Demographic infos :

Age:

25

18100

Marital Status:

Single / not married

working years

20

Model Details:

Model: Decision Tree Classifier

Accuracy: 0.93

Dataset Information:

Number of Instances: 25000

Features: 21

Target: Approval Status (1: Approv

Total Income:

926705

02000000

Personal and professional context:

Education Level:

Higher education

Job:

Accountants

Owned Realty

0

Contact Info:

Owned Mobile Phone

0

Owned Work Phone

0

Gender

☒ Male
 ☐ Female

Demographic infos :

Age:

37

18100

Marital Status:

Single / not married

working years

20

Model Details:

Model: Decision Tree Classifier

Accuracy: 0.93

Dataset Information:

Number of Instances: 25000

Features: 21

Target: Approval Status (1: Approv

Owned Mobile Phone

0

Owned Work Phone

0

Owned Phone

0

Owned Email

0

Credit history:

Total Bad Debt

0

Total Good Debt

0

Demand

Les caractéristiques ont été divisé selon leur type pour l'utilisateurs. Voici un Aperçu des Groupes de Caractéristiques :

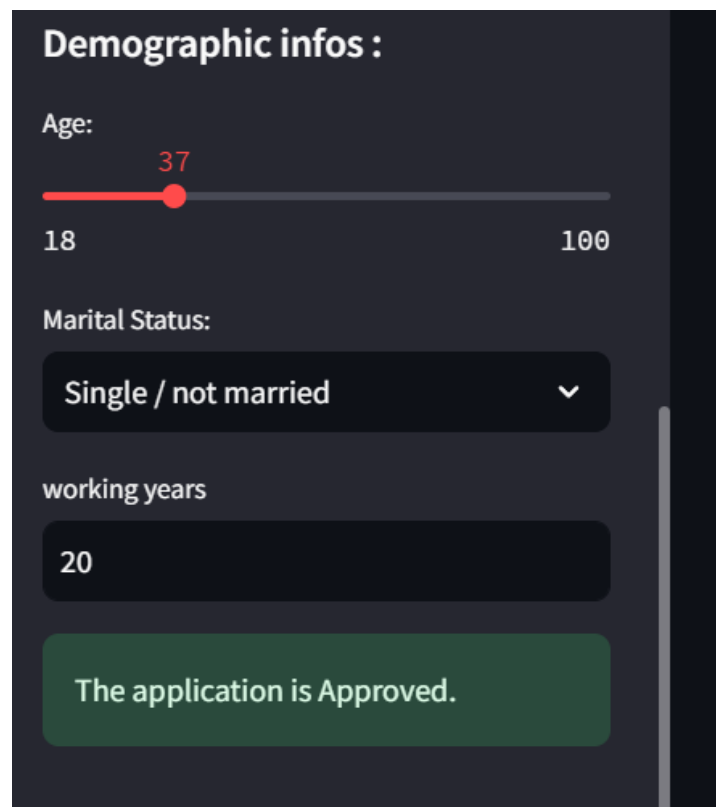
- **Identité du Demandeur** : Pièce d'identité du demandeur, Sexe du demandeur.
- **Informations Financières** : Voiture détenue, Propriété immobilière, Total des enfants, Revenu total, Type de revenu.

- **Contexte Personnel et Professionnel** : Type de formation, Situation de famille, Type de logement, Titre du poste
- **Communications et Contacts** : Téléphone mobile détenu, Téléphone de travail personnel, Téléphone personnel, Courriel de la propriété
- **Informations Démographiques** : Total des membres de la famille, Âge du demandeur, Années de travail
- **Historique de Crédit** : Total des mauvaises créances, Total de la bonne dette

## Résultat des demandes :

L'utilisateur reçoit une acceptation ou un refus de demande selon ces données.

Exemple d'acceptation :



**Demographic infos :**

Age:

37

18 100

Marital Status:

Single / not married ▼

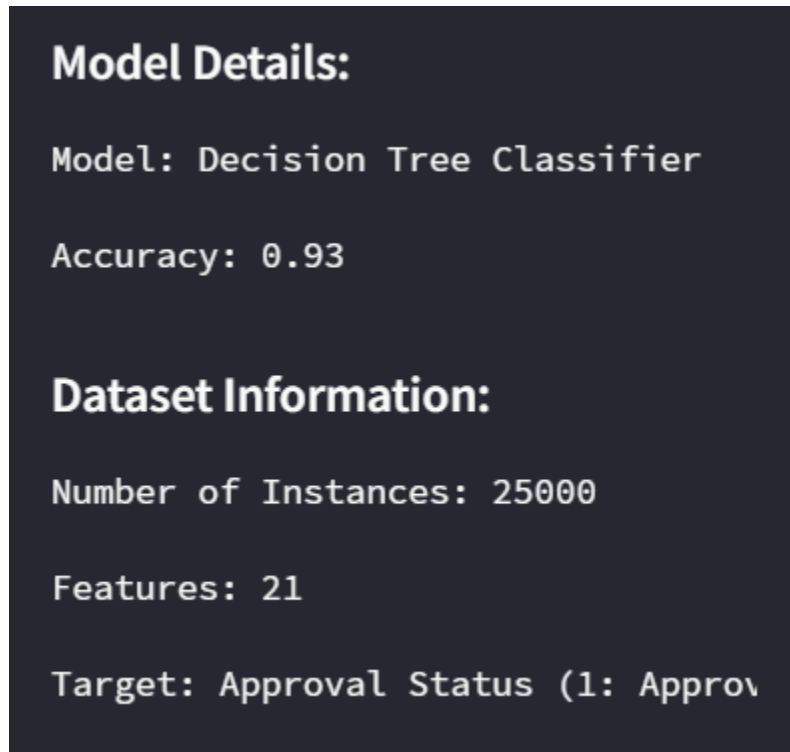
working years

20

The application is Approved.

## Informations sur le modèle :

L'interface contient aussi quelque détails spécifiques sur les informations du modèle utilisé :



## Conclusion :

En construisant ces prédicteurs de carte de crédit, nous avons abordé certaines des étapes de prétraitement les plus largement connues, telles que la mise à l'échelle, l'encodage des libellés et l'imputation des valeurs manquantes. Notre parcours s'est conclu par l'application d'apprentissage automatique pour prédire l'approbation d'une demande de carte de crédit en fonction des informations fournies. Enfin, nous avons clôturé le processus en déployant le modèle que nous avons sélectionné.