Imports

In [1]:
```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score
from sklearn.utils import resample
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
```

Import de notre dataframe

`On observe de nombreux NAN qui risque de géner l'analyse`

In [2]:
```python
df = pd.read_csv('data.csv')
print(df.isna().sum())
df.describe()
```

```
SeriousDlqin2yrs                        14465
RevolvingUtilizationOfUnsecuredLines    14465
age                                     14465
NumberOfTime30_59DaysPastDueNotWorse    14465
DebtRatio                               14465
MonthlyIncome                           14465
NumberOfOpenCreditLinesAndLoans         14465
NumberOfTimes90DaysLate                 14465
NumberRealEstateLoansOrLines            14465
NumberOfTime60_89DaysPastDueNotWorse    14465
NumberOfDependents                      14465
dtype: int64
```

Out[2]:

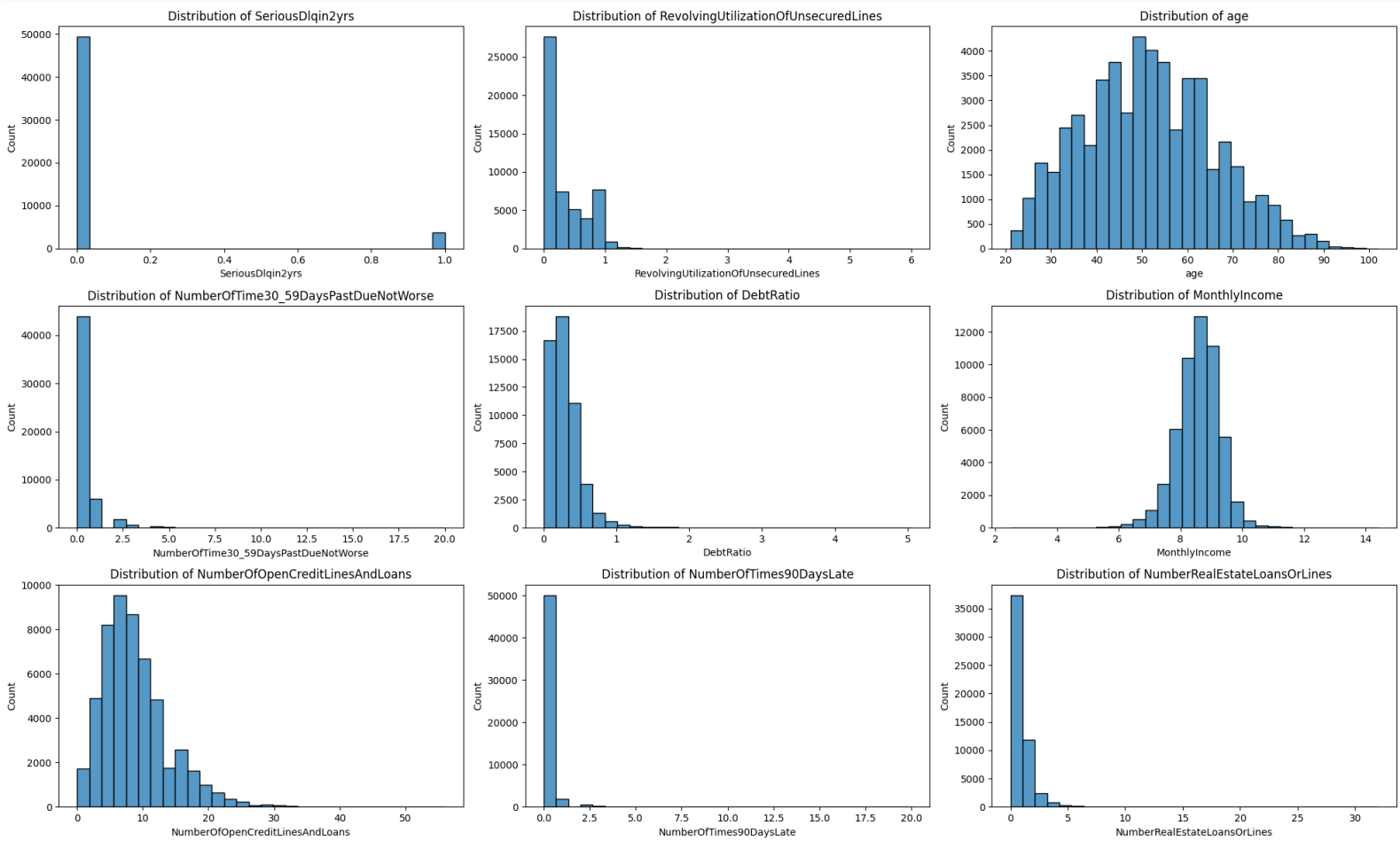
	SeriousDlqin2yrs	RevolvingUtilizationOfUnsecuredLines	age	NumberOfTime30_59DaysPastDueNotWorse	DebtRatio	MonthlyIncome	NumberOfOpenCreditLinesAndLoans	NumberOfTimes90DaysLate	NumberRealEstateLoansOrLines	NumberOfTime60_89DaysPastDueNotWorse
count	53037.000000	53037.000000	53037.000000	53037.000000	53037.000000	53037.000000	53037.000000	53037.000000	53037.000000	53037.000000
mean	0.070253	0.335496	51.359692	0.285555	0.287466	8.569434	8.805928	0.115240	1.064351	0.090503
std	0.255575	0.363052	14.364651	0.990302	0.246446	0.700666	5.186279	0.844985	1.153268	0.766177
min	0.000000	0.000000	21.000000	0.000000	0.000000	2.484907	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.035839	41.000000	0.000000	0.132507	8.160804	5.000000	0.000000	0.000000	0.000000
50%	0.000000	0.179483	51.000000	0.000000	0.255561	8.612685	8.000000	0.000000	1.000000	0.000000
75%	0.000000	0.582780	61.000000	0.000000	0.385230	9.026898	12.000000	0.000000	2.000000	0.000000
max	1.000000	6.000000	102.000000	20.000000	5.052257	14.399992	56.000000	20.000000	32.000000	20.000000

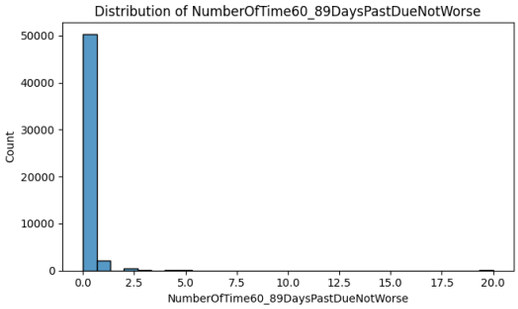
In [3]:
```python
df = df.dropna()
```

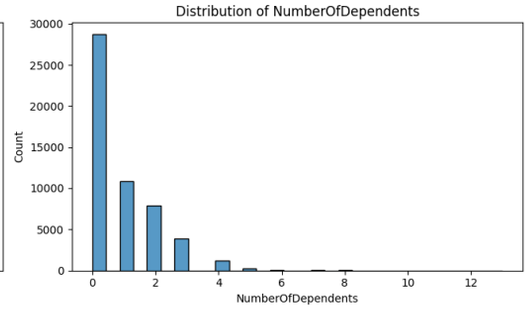
Visualisation des données

`On observe que la distribution est biaisé car il y a énormément plus de 0 que de 1 dans notre target.`
`Notre model ML risque donc d'être biaisé lui aussi si l'on utilise nos données ainsi`

In [4]:
```python
fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(20, 16))

axes = axes.flatten()

for i in range(len(df.columns), len(axes)):
    fig.delaxes(axes[i])

for i, var in enumerate(df.columns):
    sns.histplot(df[var], bins=30, kde=False, ax=axes[i])
    axes[i].set_title(f'Distribution of {var}')

plt.tight_layout()
plt.show()
```





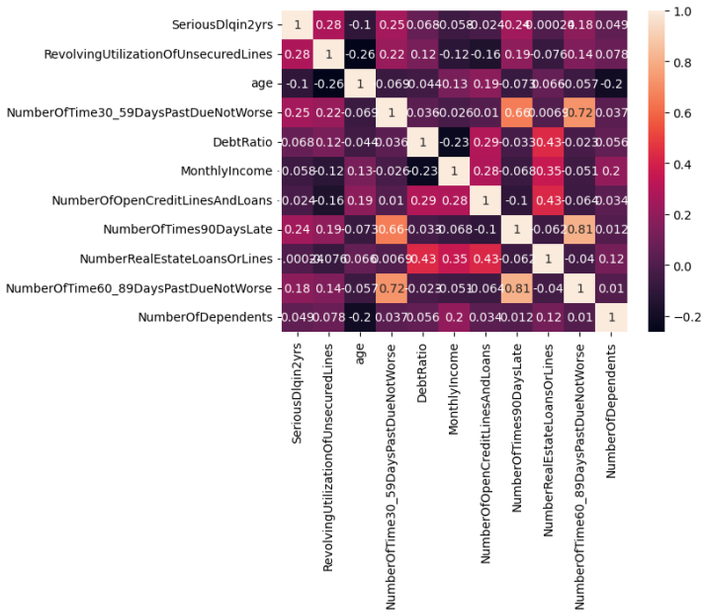


`On observe que les correlation sont toujours très faible et étant donné que nous utilisons un model de type tree il n'est peut pas nécessaire de chercher de relation non linéaire entre variable car l'algorithme est plutot bon pour les trouver`

Les models utilisé sont majoritairement les trees ainsi que les Neural network car nous pensons qu'en absence de grande corrélation et donc de potentiel relation linéaire entre les attribus, ceux-ci peuvent etre utiles

In [5]:
```
correlation_matrix = df.corr()
sns.heatmap(correlation_matrix, annot=True)
```

Out[5]:
```
<Axes: >
```



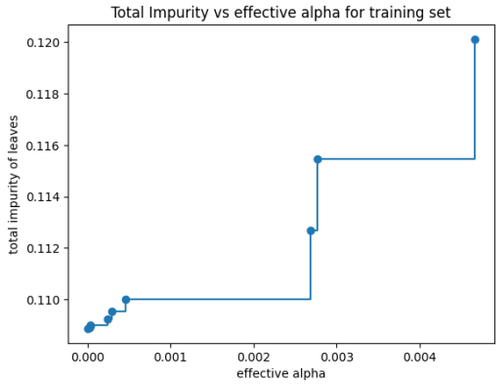
Trees Partie 1

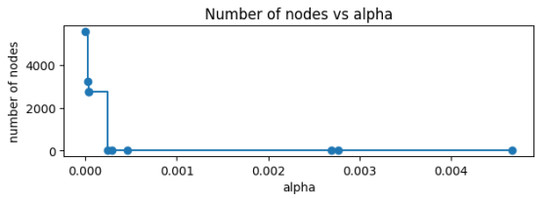
Sans essaie de retrait du biais

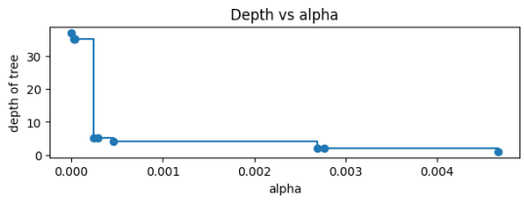
Ici nous utilisons nos graph afin d'affiner la valeur d'alpha afin d'avoir une valeur qui permette de maximiser notre scoreF1

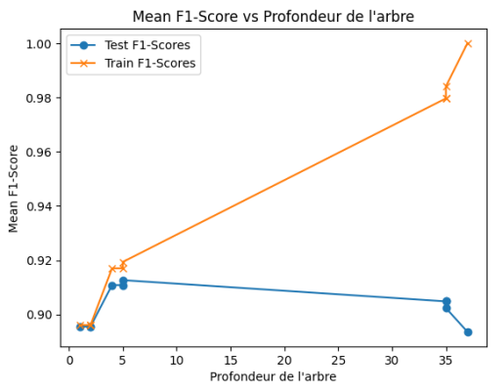
`On observe une valeur optmimal en 2.92505002e-04 car on ne s'écarte pas trop de notre train et que notre valeur est haute`

In [6]:
```
X,y=df.iloc[:, 1:],df.SeriousDlqin2yrs
X_train, X_test, y_train, y_test=train_test_split(X,y,test_size=0.33, random_state=42)
clf = DecisionTreeClassifier(max_depth=4, random_state=0,splitter='random')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

plt.figure()
plt.plot(ccp_alphas[:-1], impurities[:-1], marker='o', drawstyle="steps-post")
plt.xlabel("effective alpha")
plt.ylabel("total impurity of leaves")
plt.title("Total Impurity vs effective alpha for training set")
plt.show()

clfs = []
for ccp_alpha in ccp_alphas:
        clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
        clf.fit(X_train, y_train)
        clfs.append(clf)


clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]


node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]

    # Plotting number of nodes and depth against alpha
fig, ax = plt.subplots(2, 1)
ax[0].plot(ccp_alphas, node_counts, marker='o', drawstyle="steps-post")
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker='o', drawstyle="steps-post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
plt.show()

f1_scores_train = []
f1_scores_test = []

for clf in clfs:

        y_pred_test = clf.predict(X_test)
        y_pred_train = clf.predict(X_train)

        f1_test = f1_score(y_test, y_pred_test, average='weighted')
        f1_train = f1_score(y_train, y_pred_train, average='weighted')

        f1_scores_test.append(f1_test)
        f1_scores_train.append(f1_train)


plt.figure()
plt.plot(depth, f1_scores_test, marker='o', label='Test F1-Scores')
plt.plot(depth, f1_scores_train, marker='x', label='Train F1-Scores')
plt.xlabel('Profondeur de l\'arbre')
plt.ylabel('Mean F1-Score')
plt.title('Mean F1-Score vs Profondeur de l\'arbre')
plt.legend()
plt.show()
ccp_alphas
```









Out[6]:
```
array([0.00000000e+00, 2.82125926e-05, 3.23562034e-05, 3.28091467e-05,
       2.40608816e-04, 2.92505002e-04, 4.59422734e-04, 2.68510722e-03,
       2.76791719e-03, 4.66714926e-03])
```

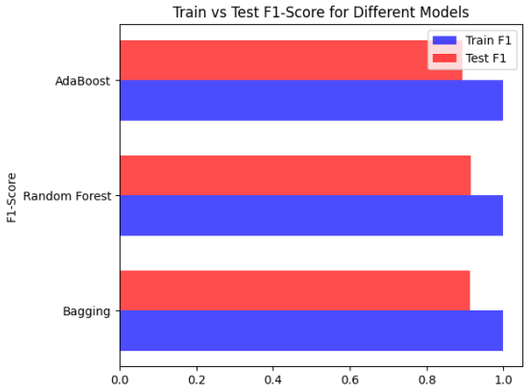
Test de la valeur 2.92505002e-04

In [7]:
```
clf = DecisionTreeClassifier(random_state=0, ccp_alpha= 2.92505002e-04)
clf.fit(X_train, y_train)
y_pred_test = clf.predict(X_test)
y_pred_train = clf.predict(X_train)
f1_test = f1_score(y_test, y_pred_test, average='weighted')
f1_train = f1_score(y_train, y_pred_train, average='weighted')
print('train ={}'.format(f1_train))
print('test ={}'.format(f1_test))
```

```
train =0.917008563021587
test =0.910824806943426
```

Maintenant essayons les autres models tel que BaggingClassifier, RandomForestClassifier et AdaBoostClassifier `On observe des résultats environ égaux mais la random forest est tout de même meilleur.` `Par ailleurs il y a des écarts importants entre le train et le test nous allons donc utiliser la métode ROC pour affiner notre analyse`

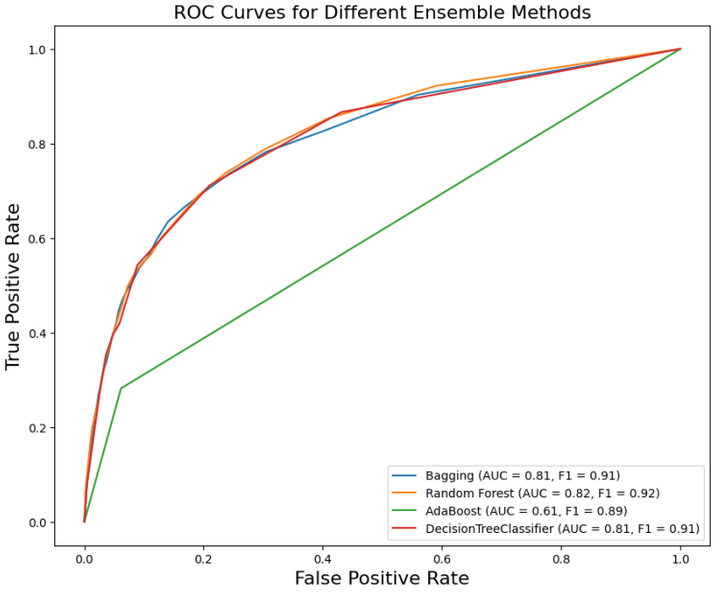
In [8]:
```
bagging_clf = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=50, random_state=42)
random_forest_clf = RandomForestClassifier(n_estimators=50, random_state=42)
adaboost_clf = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=50, random_state=42)

# Fit and predict for Bagging
bagging_clf.fit(X_train, y_train)
y_pred_train_bagging = bagging_clf.predict(X_train)
y_pred_test_bagging = bagging_clf.predict(X_test)

# Fit and predict for Random Forest
random_forest_clf.fit(X_train, y_train)
y_pred_train_rf = random_forest_clf.predict(X_train)
y_pred_test_rf = random_forest_clf.predict(X_test)

# Fit and predict for AdaBoost
adaboost_clf.fit(X_train, y_train)
y_pred_train_adaboost = adaboost_clf.predict(X_train)
y_pred_test_adaboost = adaboost_clf.predict(X_test)

# Calculate F1-Scores
f1_bagging_train = f1_score(y_train, y_pred_train_bagging, average='weighted')
f1_bagging_test = f1_score(y_test, y_pred_test_bagging, average='weighted')

f1_rf_train = f1_score(y_train, y_pred_train_rf, average='weighted')
f1_rf_test = f1_score(y_test, y_pred_test_rf, average='weighted')

f1_adaboost_train = f1_score(y_train, y_pred_train_adaboost, average='weighted')
f1_adaboost_test = f1_score(y_test, y_pred_test_adaboost, average='weighted')

# Plotting
models = ['Bagging', 'Random Forest', 'AdaBoost']
train_f1_scores = [f1_bagging_train, f1_rf_train, f1_adaboost_train]
test_f1_scores = [f1_bagging_test, f1_rf_test, f1_adaboost_test]

x = np.arange(len(models))
width = 0.35

fig, ax = plt.subplots()

rects1 = ax.barh(x - width/2, train_f1_scores, width, label='Train F1', alpha=0.7, color='blue')
rects2 = ax.barh(x + width/2, test_f1_scores, width, label='Test F1', alpha=0.7, color='red')

ax.set_ylabel('F1-Score')
ax.set_title('Train vs Test F1-Score for Different Models')
ax.set_yticks(x)
ax.set_yticklabels(models)
ax.legend()

fig.tight_layout()
plt.show()
```



On peut finalement observer que les courbes sont bien au dessus de X=Y ce qui veut dire que tous nos model est meilleur que du 50% de chance de dire la vérité (ce qui est déjà une bonne nouvelle en soit)
On peut surtout voit que l'ada boost ne semble pas intéressant compéré aux autres et que les autres semblent se valoir environ nous les soumettrons donc

In [9]:
```
from sklearn import metrics
import matplotlib.pyplot as plt

# Initialize lists to hold false positive rates, true positive rates, AUC values, and F1-Scores for different ensemble methods
fpr_ensembles = []
tpr_ensembles = []
auc_ensembles = []
f1_ensembles = []
labels = []

# List of models
models = [
    ("Bagging", BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=50, random_state=42)),
    ("Random Forest", RandomForestClassifier(n_estimators=50, random_state=42)),
    ("AdaBoost", AdaBoostClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=50, random_state=42)),
    ("DecisionTreeClassifier", DecisionTreeClassifier(ccp_alpha=2.92505002e-04, random_state=0, splitter='random'))
]

# Loop over different ensemble methods
for label, model in models:
    # Train the model
    model.fit(X_train, y_train)

    # Predict probabilities
    y_pred_proba = model.predict_proba(X_test)[:, 1]

    # Predict labels
    y_pred = model.predict(X_test)

    # Compute F1-Score and store it
    f1 = f1_score(y_test, y_pred, average='weighted')
    f1_ensembles.append(f1)

    # Compute the ROC curve
    fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_proba)
    fpr_ensembles.append(fpr)
    tpr_ensembles.append(tpr)

    # Compute the area under the ROC curve
    auc = metrics.auc(fpr, tpr)
    auc_ensembles.append(auc)
    labels.append(label)

# Display F1-Scores
for label, f1 in zip(labels, f1_ensembles):
    print(f"F1-Score for {label}: {f1:.2f}")

# Plotting the ROC curves
plt.figure(figsize=(10, 8))

# Loop through and plot each ROC curve
for i, label in enumerate(labels):
    plt.plot(fpr_ensembles[i], tpr_ensembles[i], label=f'{label} (AUC = {auc_ensembles[i]:.2f}, F1 = {f1_ensembles[i]:.2f})')

plt.xlabel('False Positive Rate', fontsize=16)
plt.ylabel('True Positive Rate', fontsize=16)
plt.title('ROC Curves for Different Ensemble Methods', fontsize=16)
plt.legend(loc="lower right")
plt.show()
```

```
F1-Score for Bagging: 0.91
F1-Score for Random Forest: 0.92
F1-Score for AdaBoost: 0.89
F1-Score for DecisionTreeClassifier: 0.91
```



Trees Partie 2

En applicant une normalisation

`Nous avons tenté de normaliser nos données mais les résultats étaient strictement égaux`

In [10]:
```
scaler = MinMaxScaler()
df = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
df
```

Out[10]:

	SeriousDlqin2yrs	RevolvingUtilizationOfUnsecuredLines	age	NumberOfTime30_59DaysPastDueNotWorse	DebtRatio	MonthlyIncome	NumberOfOpenCreditLinesAndLoans	NumberOfTimes90DaysLate	NumberRealEstateLoansOrLines	NumberOfTime60_89DaysPastDueNotWorse	Number
0	0.0	0.159525	0.234568	0.00	0.022763	0.451423	0.071429	0.00	0.00000	0.0	
1	0.0	0.109697	0.209877	0.05	0.016168	0.464595	0.035714	0.05	0.00000	0.0	
2	0.0	0.151207	0.345679	0.05	0.004873	0.719700	0.125000	0.00	0.03125	0.0	
3	0.0	0.035530	0.654321	0.00	0.063119	0.476362	0.053571	0.00	0.03125	0.0	
4	0.0	0.125744	0.222222	0.00	0.037720	0.476362	0.142857	0.00	0.00000	0.0	

...	...	...	...	...	...	...	...	...	...	...
53032	0.0	0.017352	0.469136	0.00	0.077284	0.567066	0.178571	0.00	0.06250	0.0
53033	0.0	0.064290	0.358025	0.00	0.067204	0.473930	0.125000	0.00	0.00000	0.0
53034	0.0	0.049958	0.283951	0.00	0.106947	0.515559	0.071429	0.00	0.03125	0.0
53035	0.0	0.000000	0.111111	0.00	0.000000	0.517520	0.071429	0.00	0.00000	0.0
53036	0.0	0.141714	0.530864	0.00	0.044153	0.547371	0.142857	0.00	0.06250	0.0

53037 rows × 11 columns

Trees Partie 3

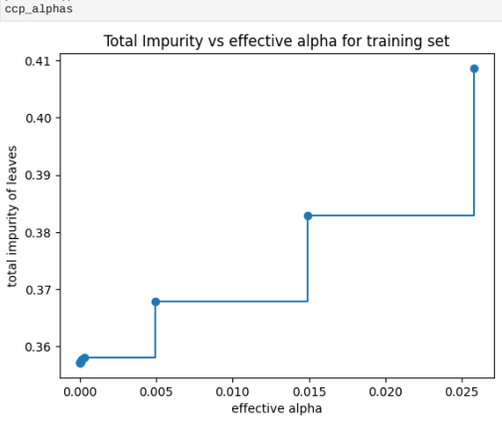
En tentant de contrebalancer le biais de surepréservation

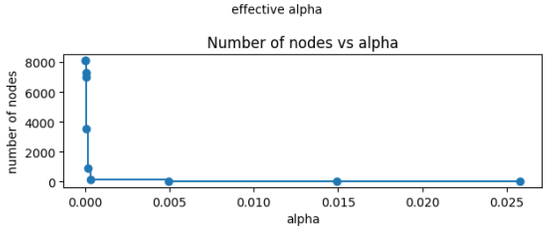
`Pour tenter de contrebalancer la surepréservation nous allons tenté de sur-échantillonner les autres valeurs` `Nous allons détailler notre approche dans les commentaires`

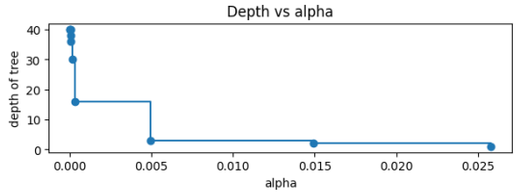
In [11]:
```
df = pd.read_csv('data.csv')
df = df.dropna()
df.isna().sum()#Import et nettoyage des données
#df[df["SeriousDlqin2yrs"]== 1].count() # Nous comptons tous les clients qui sont sérieux dans notre df de train (c'est principalement ici pour se rendre compte du déséquilibre de notre dataset)

df_majority = df[df.SeriousDlqin2yrs==0]# catégorie majoritaire
df_minority = df[df.SeriousDlqin2yrs==1]# catégorie minoritaire

df_minority_resampled = resample(df_minority,                    # Sur-échantillonnage de la classe minoritaire en la portant à la même taille que la classe majoritaire
                                 replace=True,                   # En utilisant un échantillonnage aléatoire avec remplacement
                                 n_samples=df_majority.shape[0],
                                 random_state=42)


df_wo_bias = pd.concat([df_majority, df_minority_resampled])


df_wo_bias.SeriousDlqin2yrs.value_counts()

# Prepare data for training
X, y = df_wo_bias.iloc[:, 1:], df_wo_bias.SeriousDlqin2yrs
```

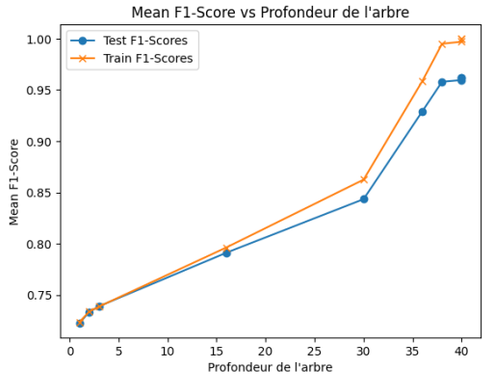
Test avec le sur-échantillonnage

In [12]:
```
X_train, X_test, y_train, y_test=train_test_split(X,y,test_size=0.33, random_state=42)
clf = DecisionTreeClassifier(max_depth=4, random_state=0,splitter='random')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

plt.figure()
plt.plot(ccp_alphas[:-1], impurities[:-1], marker='o', drawstyle="steps-post")
plt.xlabel("effective alpha")
plt.ylabel("total impurity of leaves")
plt.title("Total Impurity vs effective alpha for training set")
plt.show()

clfs = []
for ccp_alpha in ccp_alphas:
        clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
        clf.fit(X_train, y_train)
        clfs.append(clf)


clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]

node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]

    # Plotting number of nodes and depth against alpha
fig, ax = plt.subplots(2, 1)
ax[0].plot(ccp_alphas, node_counts, marker='o', drawstyle="steps-post")
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker='o', drawstyle="steps-post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
plt.show()

f1_scores_train = []
f1_scores_test = []


for clf in clfs:

        y_pred_test = clf.predict(X_test)
        y_pred_train = clf.predict(X_train)

        f1_test = f1_score(y_test, y_pred_test, average='weighted')
        f1_train = f1_score(y_train, y_pred_train, average='weighted')

        f1_scores_test.append(f1_test)
        f1_scores_train.append(f1_train)

plt.figure()
plt.plot(depth, f1_scores_test, marker='o', label='Test F1-Scores')
plt.plot(depth, f1_scores_train, marker='x', label='Train F1-Scores')
plt.xlabel('Profondeur de l\'arbre')
plt.ylabel('Mean F1-Score')
plt.title('Mean F1-Score vs Profondeur de l\'arbre')
plt.legend()
plt.show()
ccp_alphas
```







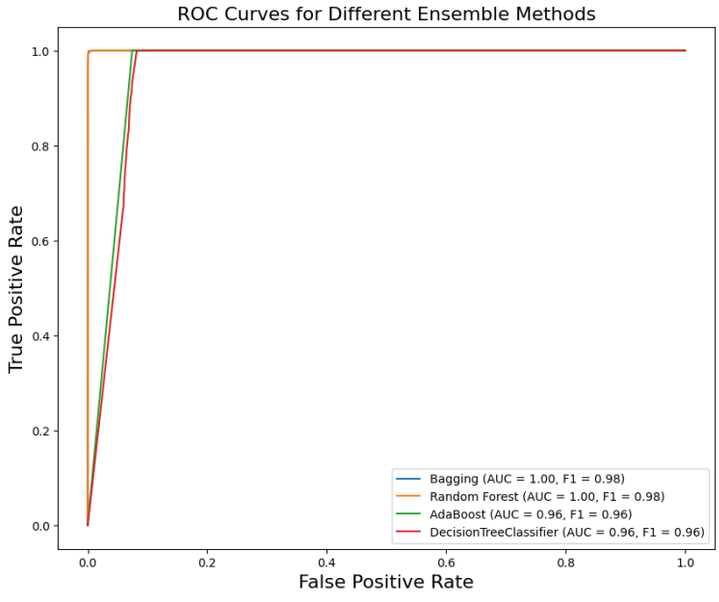


Out[12]:
```
array([0.00000000e+00, 6.67331305e-06, 1.71016553e-05, 2.51769851e-05,
       6.12230547e-05, 1.63727702e-04, 2.90703537e-04, 4.92659703e-03,
       1.49340218e-02, 2.58114663e-02])
```

On voit que le fait d'appliquer notre sur-échantillonnage semble fonctionner spécialement avec la random forest et le bagging

In [13]:
```python
from sklearn import metrics
import matplotlib.pyplot as plt

# Initialize lists to hold false positive rates, true positive rates, AUC values, and F1-Scores for different ensemble methods
fpr_ensembles = []
tpr_ensembles = []
auc_ensembles = []
f1_ensembles = []
labels = []

# List of models
models = [
    ("Bagging", BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=50, random_state=42)),
    ("Random Forest", RandomForestClassifier(n_estimators=50, random_state=42)),
    ("AdaBoost", AdaBoostClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=50, random_state=42)),
    ("DecisionTreeClassifier", DecisionTreeClassifier(ccp_alpha=1.71016553e-05, random_state=0, splitter='random'))
]

# Loop over different ensemble methods
for label, model in models:
    # Train the model
    model.fit(X_train, y_train)

    # Predict probabilities
    y_pred_proba = model.predict_proba(X_test)[:, 1]

    # Predict labels
    y_pred = model.predict(X_test)

    # Compute F1-Score and store it
    f1 = f1_score(y_test, y_pred, average='weighted')
    f1_ensembles.append(f1)

    # Compute the ROC curve
    fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_proba)
    fpr_ensembles.append(fpr)
    tpr_ensembles.append(tpr)

    # Compute the area under the ROC curve
    auc = metrics.auc(fpr, tpr)
    auc_ensembles.append(auc)
    labels.append(label)

# Display F1-Scores
for label, f1 in zip(labels, f1_ensembles):
    print(f"F1-Score for {label}: {f1:.2f}")

# Plotting the ROC curves
plt.figure(figsize=(10, 8))

# Loop through and plot each ROC curve
for i, label in enumerate(labels):
    plt.plot(fpr_ensembles[i], tpr_ensembles[i], label=f'{label} (AUC = {auc_ensembles[i]:.2f}, F1 = {f1_ensembles[i]:.2f})')

plt.xlabel('False Positive Rate', fontsize=16)
plt.ylabel('True Positive Rate', fontsize=16)
plt.title('ROC Curves for Different Ensemble Methods', fontsize=16)
plt.legend(loc="lower right")
plt.show()
```

```
F1-Score for Bagging: 0.98
F1-Score for Random Forest: 0.98
F1-Score for AdaBoost: 0.96
F1-Score for DecisionTreeClassifier: 0.96
```



Neural network

`Ici on sur-échantillonne notre df car les réseaux de neurones on tendance à sur apprendre si non`

Avec sur échantillonnage:

In [26]:
```
import tensorflow as tf
import tensorflow_addons as tfa
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [15]:
```
df = pd.read_csv('data.csv')
df = df.dropna()
```

In [16]:
```
df_majority = df[df.SeriousDlqin2yrs==0]# catégorie majoritaire
df_minority = df[df.SeriousDlqin2yrs==1]# catégorie minoritaire

df_minority_resampled = resample(df_minority,                    # Sur-échantillonnage de la classe minoritaire en la portant à la même taille que la classe majoritaire
                                 replace=True,                   # En utilisant un échantillonnage aléatoire avec remplacement
                                 n_samples=df_majority.shape[0],
                                 random_state=42)


df_wo_bias = pd.concat([df_majority, df_minority_resampled])


X, y = df_wo_bias.iloc[:, 1:], df_wo_bias.SeriousDlqin2yrs
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)


# Prepare data for training
X, y = df_wo_bias.iloc[:, 1:], df_wo_bias.SeriousDlqin2yrs
scaler = StandardScaler()
X_train, X_test = scaler.fit_transform(X_train), scaler.transform(X_test)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='relu', input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1, activation='sigmoid')
])


model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy', tfa.metrics.F1Score(num_classes=1, average='weighted', threshold=0.5)])


model.fit(X_train, y_train, epochs=10, batch_size=20, validation_split=0.2)

test_loss, test_accuracy, test_f1_score = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_accuracy}")
print(f"Test F1-Score: {test_f1_score}")
```

```
Epoch 1/10
2643/2643 [==============================] - 4s 1ms/step - loss: 0.5338 - accuracy: 0.7444 - f1_score: 0.7427 - val_loss: 0.4807 - val_accuracy: 0.7741 - val_f1_score: 0.7733
Epoch 2/10
2643/2643 [==============================] - 3s 981us/step - loss: 0.5014 - accuracy: 0.7723 - f1_score: 0.7736 - val_loss: 0.4766 - val_accuracy: 0.7781 - val_f1_score: 0.7739
Epoch 3/10
2643/2643 [==============================] - 3s 979us/step - loss: 0.4922 - accuracy: 0.7749 - f1_score: 0.7761 - val_loss: 0.4726 - val_accuracy: 0.7791 - val_f1_score: 0.7793
Epoch 4/10
2643/2643 [==============================] - 3s 970us/step - loss: 0.4891 - accuracy: 0.7751 - f1_score: 0.7750 - val_loss: 0.4701 - val_accuracy: 0.7779 - val_f1_score: 0.7740
Epoch 5/10
2643/2643 [==============================] - 3s 1ms/step - loss: 0.4849 - accuracy: 0.7771 - f1_score: 0.7748 - val_loss: 0.4704 - val_accuracy: 0.7781 - val_f1_score: 0.7770
Epoch 6/10
2643/2643 [==============================] - 3s 978us/step - loss: 0.4845 - accuracy: 0.7770 - f1_score: 0.7754 - val_loss: 0.4691 - val_accuracy: 0.7769 - val_f1_score: 0.7701
Epoch 7/10
2643/2643 [==============================] - 2s 925us/step - loss: 0.4830 - accuracy: 0.7768 - f1_score: 0.7745 - val_loss: 0.4687 - val_accuracy: 0.7807 - val_f1_score: 0.7767
Epoch 8/10
2643/2643 [==============================] - 2s 920us/step - loss: 0.4836 - accuracy: 0.7771 - f1_score: 0.7740 - val_loss: 0.4659 - val_accuracy: 0.7784 - val_f1_score: 0.7768
Epoch 9/10
2643/2643 [==============================] - 2s 923us/step - loss: 0.4815 - accuracy: 0.7767 - f1_score: 0.7742 - val_loss: 0.4667 - val_accuracy: 0.7782 - val_f1_score: 0.7747
Epoch 10/10
2643/2643 [==============================] - 2s 920us/step - loss: 0.4811 - accuracy: 0.7776 - f1_score: 0.7755 - val_loss: 0.4652 - val_accuracy: 0.7829 - val_f1_score: 0.7799
1018/1018 [==============================] - 1s 648us/step - loss: 0.4673 - accuracy: 0.7834 - f1_score: 0.7800
Test accuracy: 0.783352792263031
Test F1-Score: 0.7800205945968628
```

Sans sur-échantillonnage :

In [17]:
```
df = pd.read_csv('data.csv')
df = df.dropna()
X, y = df.iloc[:, 1:], df.SeriousDlqin2yrs
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
scaler = StandardScaler()
X_train, X_test = scaler.fit_transform(X_train), scaler.transform(X_test)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='relu', input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1, activation='sigmoid')
])


model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy', tfa.metrics.F1Score(num_classes=1, average='weighted', threshold=0.5)])


model.fit(X_train, y_train, epochs=10, batch_size=20, validation_split=0.2)

test_loss, test_accuracy, test_f1_score = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_accuracy}")
print(f"Test F1-Score: {test_f1_score}")
```

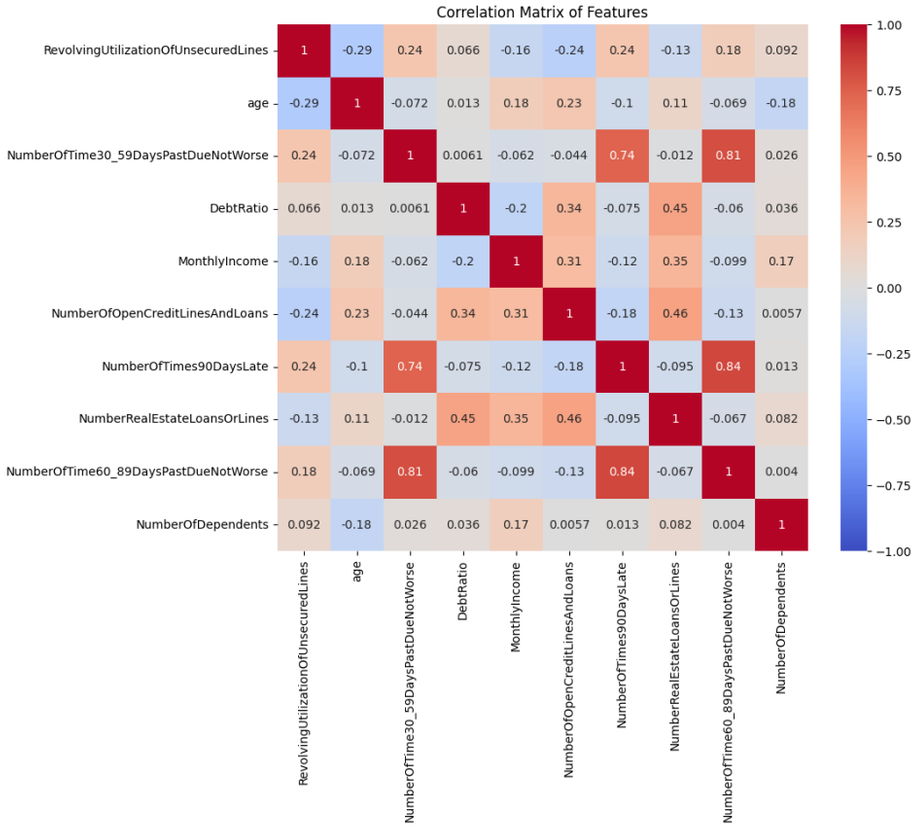
```
Epoch 1/10
1422/1422 [==============================] - 2s 1ms/step - loss: 0.2365 - accuracy: 0.9309 - f1_score: 0.1776 - val_loss: 0.1913 - val_accuracy: 0.9322 - val_f1_score: 0.1603
Epoch 2/10
1422/1422 [==============================] - 2s 1ms/step - loss: 0.2038 - accuracy: 0.9326 - f1_score: 0.1778 - val_loss: 0.1860 - val_accuracy: 0.9319 - val_f1_score: 0.1357
Epoch 3/10
1422/1422 [==============================] - 1s 1ms/step - loss: 0.1994 - accuracy: 0.9329 - f1_score: 0.1662 - val_loss: 0.1846 - val_accuracy: 0.9323 - val_f1_score: 0.1693
Epoch 4/10
1422/1422 [==============================] - 1s 1ms/step - loss: 0.1958 - accuracy: 0.9331 - f1_score: 0.1577 - val_loss: 0.1843 - val_accuracy: 0.9323 - val_f1_score: 0.1333
Epoch 5/10
1422/1422 [==============================] - 1s 1ms/step - loss: 0.1959 - accuracy: 0.9322 - f1_score: 0.1316 - val_loss: 0.1847 - val_accuracy: 0.9327 - val_f1_score: 0.1871
Epoch 6/10
1422/1422 [==============================] - 1s 1ms/step - loss: 0.1950 - accuracy: 0.9330 - f1_score: 0.1634 - val_loss: 0.1848 - val_accuracy: 0.9323 - val_f1_score: 0.1806
Epoch 7/10
1422/1422 [==============================] - 1s 997us/step - loss: 0.1929 - accuracy: 0.9339 - f1_score: 0.2175 - val_loss: 0.1836 - val_accuracy: 0.9323 - val_f1_score: 0.1750
Epoch 8/10
1422/1422 [==============================] - 1s 974us/step - loss: 0.1925 - accuracy: 0.9335 - f1_score: 0.1938 - val_loss: 0.1856 - val_accuracy: 0.9319 - val_f1_score: 0.1136
Epoch 9/10
1422/1422 [==============================] - 1s 1ms/step - loss: 0.1932 - accuracy: 0.9339 - f1_score: 0.1967 - val_loss: 0.1849 - val_accuracy: 0.9327 - val_f1_score: 0.1464
Epoch 10/10
1422/1422 [==============================] - 1s 975us/step - loss: 0.1925 - accuracy: 0.9331 - f1_score: 0.1765 - val_loss: 0.1848 - val_accuracy: 0.9332 - val_f1_score: 0.2070
547/547 [==============================] - 0s 639us/step - loss: 0.1924 - accuracy: 0.9322 - f1_score: 0.1705
Test accuracy: 0.932183027267456
Test F1-Score: 0.17051014304161072
```

Logistic classification

I want to make sure that the features i chose earlier are not highly correlated

- Checking the Correlation Matrix:

In [18]:
```
import seaborn as sns
import matplotlib.pyplot as plt

df_majority = df[df.SeriousDlqin2yrs==0]# catégorie majoritaire
df_minority = df[df.SeriousDlqin2yrs==1]# catégorie minoritaire

df_minority_resampled = resample(df_minority,                    # Sur-échantillonnage de la classe minoritaire en la portant à la même taille que la classe majoritaire
                                 replace=True,                   # En utilisant un échantillonnage aléatoire avec remplacement
                                 n_samples=df_majority.shape[0],
                                 random_state=42)


df_wo_bias = pd.concat([df_majority, df_minority_resampled])


X, y = df_wo_bias.iloc[:, 1:], df_wo_bias.SeriousDlqin2yrs
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
# Compute the correlation matrix
corr = X_train.corr()

# Draw the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title("Correlation Matrix of Features")
plt.show()
```



- Checking the Variance Inflation Factor

In [19]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor

# Calculate VIF for each feature
vif_data = pd.DataFrame()
vif_data["Feature"] = X_train.columns
vif_data["VIF"] = [variance_inflation_factor(X_train.values, i) for i in range(X_train.shape[1])]

print(vif_data)
```

```
                                Feature        VIF
0  RevolvingUtilizationOfUnsecuredLines   3.016605
1                                   age  15.685631
2  NumberOfTime30_59DaysPastDueNotWorse   3.685732
3                            DebtRatio   3.118425
4                        MonthlyIncome  23.129401
5       NumberOfOpenCreditLinesAndLoans   5.294920
6                NumberOfTimes90DaysLate   3.876593
7            NumberRealEstateLoansOrLines   2.443091
8  NumberOfTime60_89DaysPastDueNotWorse   4.998565
9                     NumberOfDependents   1.750183
```

Generally, a VIF value above 10 indicates high multicollinearity. Here, age and MonthlyIncome have VIF values significantly higher than 10

- I'm gonna start by checking their correlations with other variabless so i can decide whether to remove or combine features.

In [20]:
```
print(corr['age'])
print(corr['MonthlyIncome'])
```

```
RevolvingUtilizationOfUnsecuredLines   -0.291766
age                                     1.000000
NumberOfTime30_59DaysPastDueNotWorse   -0.071917
DebtRatio                               0.013235
MonthlyIncome                           0.177498
NumberOfOpenCreditLinesAndLoans         0.228896
NumberOfTimes90DaysLate                -0.099958
NumberRealEstateLoansOrLines            0.114241
NumberOfTime60_89DaysPastDueNotWorse   -0.068708
NumberOfDependents                     -0.175515
Name: age, dtype: float64
RevolvingUtilizationOfUnsecuredLines   -0.162314
age                                     0.177498
NumberOfTime30_59DaysPastDueNotWorse   -0.062285
DebtRatio                              -0.198250
MonthlyIncome                           1.000000
NumberOfOpenCreditLinesAndLoans         0.310929
NumberOfTimes90DaysLate                -0.117346
NumberRealEstateLoansOrLines            0.346710
NumberOfTime60_89DaysPastDueNotWorse   -0.099408
NumberOfDependents                      0.167039
Name: MonthlyIncome, dtype: float64
```

I dropped the 'age' column from the DataFrame because i believe that the correlations are strong enough and that including both age and MonthlyIncome in the model may lead to multicollinearity issues. So i chose age necause i think it's less relevant for our case.

In [21]:
```
X_train = X_train.drop('age', axis=1)
X_test = X_test.drop('age', axis=1)
```

Implementing Logistic Regression: We observe that the ROC score is a lot lower than other model

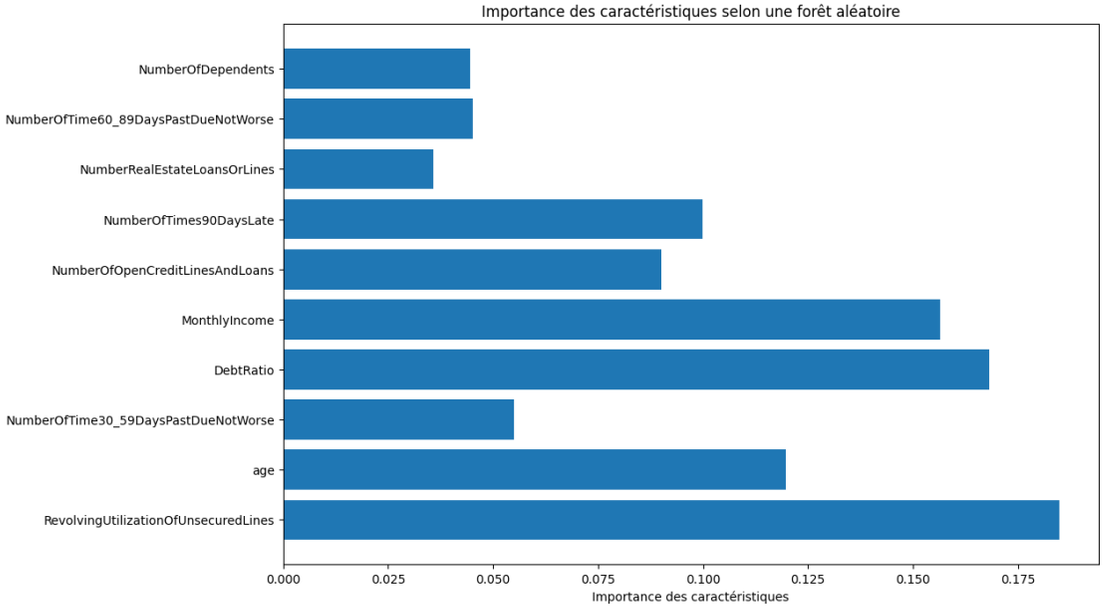
In [22]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score

# Initialize the model
logreg = LogisticRegression(max_iter=10000, random_state=42)

# Fit the model
logreg.fit(X_train, y_train)

# Predict on the test set
y_pred = logreg.predict(X_test)

# Evaluate the model
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("ROC-AUC Score:", roc_auc_score(y_test, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

         0.0       0.75      0.79      0.77     16325
         1.0       0.78      0.73      0.75     16221

    accuracy                           0.76     32546
   macro avg       0.76      0.76      0.76     32546
weighted avg       0.76      0.76      0.76     32546

Confusion Matrix:
 [[12970  3355]
 [ 4381 11840]]
ROC-AUC Score: 0.7622024953378927
```

Autre approche essayée

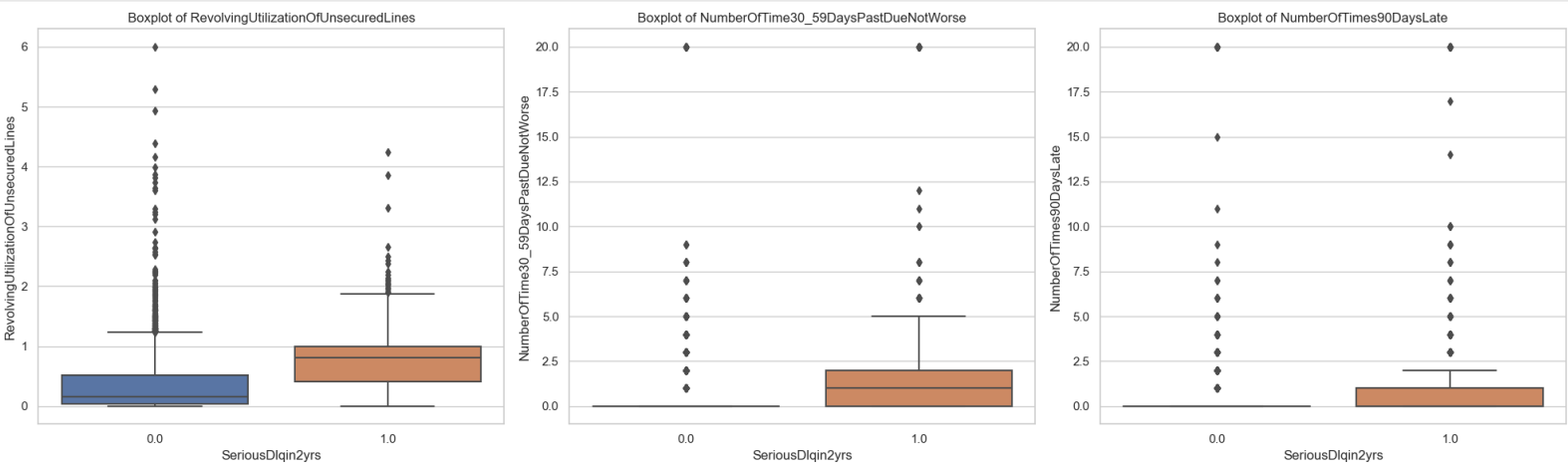
`Nous avons également essayer de retirer des colones inutile en utilisant une random forest pour connaitre les plus intéressantes mais sans résultat concluant`

In [23]:
```
X = df.drop('SeriousDlqin2yrs', axis=1)
y = df['SeriousDlqin2yrs']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the Random Forest Classifier
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the classifier
rf_clf.fit(X_train, y_train)

# Get feature importances
feature_importances = rf_clf.feature_importances_

plt.figure(figsize=(12, 8))
plt.barh(range(len(feature_importances)), feature_importances, align='center')
plt.yticks(np.arange(len(feature_importances)), X.columns)
plt.xlabel('Importance des caractéristiques')
plt.title('Importance des caractéristiques selon une forêt aléatoire')
plt.show()
```



`Nous avons également essayé de retirer les données abérantes mes sans grand succes`

In [24]:
```
import matplotlib.pyplot as plt
import seaborn as sns


sns.set(style="whitegrid")
most_correlated_vars = ['RevolvingUtilizationOfUnsecuredLines', 'NumberOfTime30_59DaysPastDueNotWorse', 'NumberOfTimes90DaysLate']
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(20, 6))
for i, var in enumerate(most_correlated_vars):
    sns.boxplot(x='SeriousDlqin2yrs', y=var, data=df, ax=axes[i])
    axes[i].set_title(f'Boxplot of {var}')

plt.tight_layout()
plt.show()
```



Enregistrement des résultats

In [25]:
```
test_df = pd.read_csv('test.csv')
clf = DecisionTreeClassifier(random_state=0, ccp_alpha= 6.67331305e-06)
clf.fit(X_train, y_train)
test_predictions = clf.predict(test_df)
to_be_submitted = pd.DataFrame({'id': test_df.index+1, 'SeriousDlqin2yrs': test_predictions})
to_be_submitted['SeriousDlqin2yrs'] = to_be_submitted['SeriousDlqin2yrs'].astype(int)
to_be_submitted.to_csv("to_be_submitted.csv", index=False)
to_be_submitted
```

Out[25]:

	id	SeriousDlqin2yrs
0	1	0
1	2	0
2	3	0
3	4	0
4	5	1
...	...	...
1451	1452	0
1452	1453	0
1453	1454	0
1454	1455	0
1455	1456	1

1456 rows × 2 columns