IMARTICUS LEARNING

PGA-20

Capstone Project Report on Predicting Term Deposit Subscriptions in Banking

Under Guidance of
Arun Upadhyay

Submitted By:
Rayapati Sai Nikitha

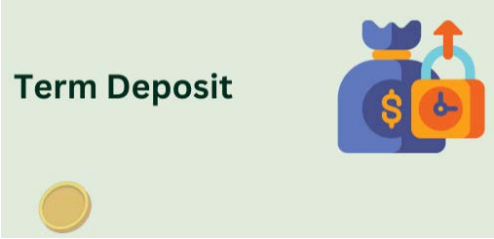
Date:
28-Feb-2024

Table Of Contents

Abstract:

Predicting term deposit churn is really important for banks. It means figuring out when customers might take their money out early. In this study, we used different prediction algorithms to look at how customers act and guess if they might take their money out soon.
We looked at information that's available to everyone, like how old customers are, what they do with their money, and how often they use the bank. We used Machine learning algorithms called logistic regression, decision trees, and random forests to help us guess.
We found out that things like how old customers are, what they do with their money, and how often they use the bank are important in guessing if they might take their money out early. One of the computer programs, the random forest one, was the best at guessing.
This study shows that if banks use these Machine Learning Algorithms, they can figure out who might take their money out early and try to stop them. This can help banks keep customers happy and make more money in the long run.

Introduction



Background

In banking, "term deposit churn" is a big challenge when customers withdraw their funds early, hurting the bank's profits. Term deposits are fixed savings accounts like the money is deposited for a specific period, typically ranging from a few months to several years, at a predetermined interest rate.
Understanding why customers withdraw early is crucial for banks to keep customers happy and make better financial decisions. This study predicts term deposit churn using historical banking data and models like logistic regression, decision trees, and random forests.
The aim is to help banks improve customer retention and increase long-term profits by understanding why customers withdraw early. This can lead to better customer service and happier customers, ultimately benefiting the bank financially.

Objectives

The objectives of this project are straightforward: We aim to compare customers who keep their money in the bank for the agreed time to those who take it out early.

Additionally, we aim to understand why some customers take their money out early, such as who they are, how they use their accounts, and how they communicate with the bank. This helps us figure out how to keep customers satisfied and prevent them from withdrawing their money early.

Overview of the Dataset

The dataset consists of various features that provide insights into customer demographics, banking behavior, and interactions. Here's a brief description of each feature:

ID: Unique identifier for each customer.
Age: Age of the customer.
Job: Occupation of the customer.
Marital: Marital status of the customer.
Education: Level of education attained by the customer.
Default: Indicates whether the customer has credit in default (yes/no).
Balance: Current balance in the customer's account.
Housing: Indicates whether the customer has a housing loan (yes/no).
Loan: Indicates whether the customer has a personal loan (yes/no).
Contact: Type of communication contact with the customer (e.g., cellular, telephone).
Day: Day of the month when the contact was made.
Month: Month of the year when the contact was made.
Duration: Duration of the last contact with the customer in seconds.
Campaign: Number of contacts performed during this campaign for this customer.
Pdays: Number of days since the customer was last contacted from a previous campaign (-1 means not previously contacted).
Previous: Number of contacts performed before this campaign for this customer.
Poutcome: Outcome of the previous marketing campaign (success, failure, other).
Y: Target variable indicating whether the customer subscribed to a term deposit (yes/no).
This dataset will be used to analyze customer behavior and predict whether a customer will subscribe to a term deposit.

Reading Data:

```
df_bank=pd.read_csv(r"C:\Users\sm983\Desktop\Machine Learning Projects\Term_deposit\Assignment-2_Data.csv")
df_bank.head()
```

4]:

	Id	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
	1001	999.0	management	married	tertiary	no	2143.0	yes	no	unknown	5	may	261	1	-1	0	unknown	no
	1002	44.0	technician	single	secondary	no	29.0	yes	no	unknown	5	may	151	1	-1	0	unknown	no
	1003	33.0	entrepreneur	married	secondary	no	2.0	yes	yes	unknown	5	may	76	1	-1	0	unknown	no
	1004	47.0	blue-collar	married	unknown	no	1506.0	yes	no	unknown	5	may	92	1	-1	0	unknown	no
	1005	33.0	unknown	single	unknown	no	1.0	no	no	unknown	5	may	198	1	-1	0	unknown	no

Literature Review:

- **Previous Studies on Term Deposit Churn Prediction**:

Understanding why customers might withdraw their term deposits early is really important for banks. Previous research has shown that certain factors play a big role in predicting if customers will take their money out early.

For example, younger customers or those with unstable incomes tend to withdraw their deposits more often. Also, customers who frequently withdraw large sums or have a history of risky financial behavior are more likely to withdraw their deposits early.

Additionally, how customers interact with the bank is a key factor. If customers have bad experiences with customer service, don't get personalized communication, or have limited access to banking resources, they're more likely to take their money out early. These findings help banks understand which customers are at risk of withdrawing their deposits early and can guide strategies to keep them happy and prevent them from leaving.

- **Review of Relevant Literature on Logistic Regression and Other ML Models:**

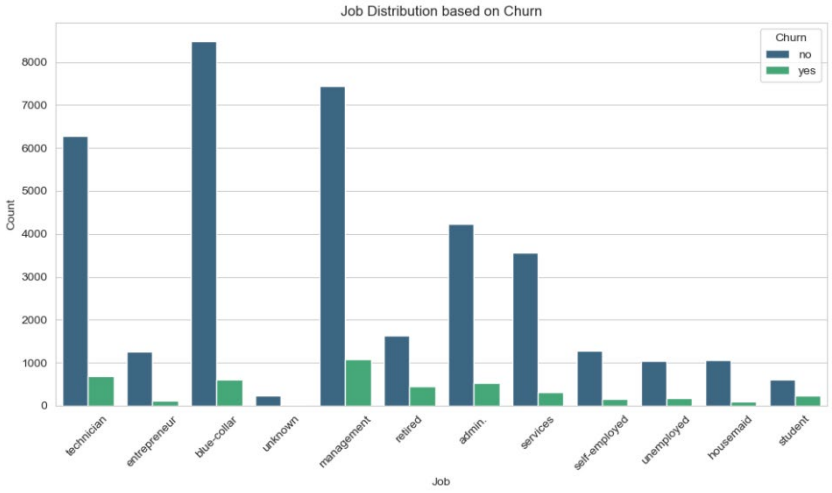
Logistic regression and other machine learning models are widely used to predict term deposit churn. Logistic regression is a simple and easy-to-understand statistical method that's often used because it helps us understand how different factors relate to whether customers will withdraw their deposits early.

In addition to logistic regression, other machine learning algorithms like decision trees, random forests are also used. These models analyze patterns in the data to identify customers who are at risk of withdrawing their deposits early. This helps banks develop strategies to keep these customers happy and prevent them from leaving.

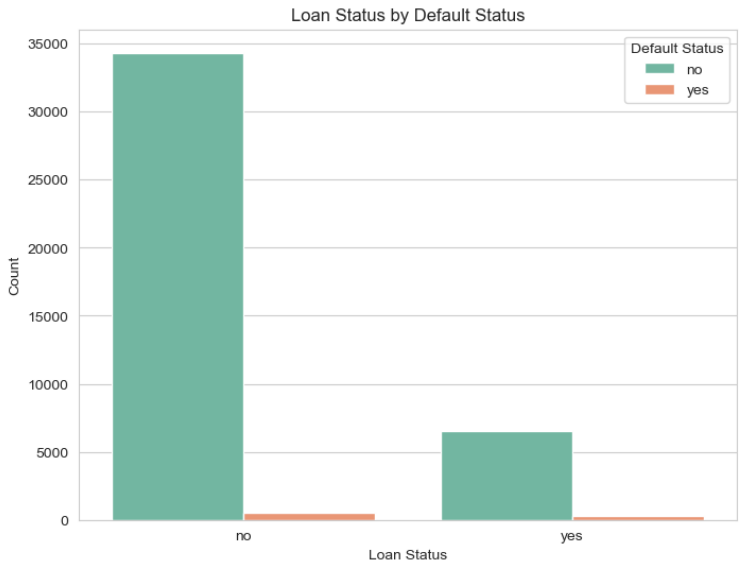
By studying how different factors affect the chances of term deposit churn, logistic regression models give us useful insights into customer behavior. Similarly, decision trees and random forests, two other types of machine learning methods, help us understand patterns in the data. They show us which factors matter the most in predicting if customers will withdraw their deposits early.

Data Visualization and Exploration:

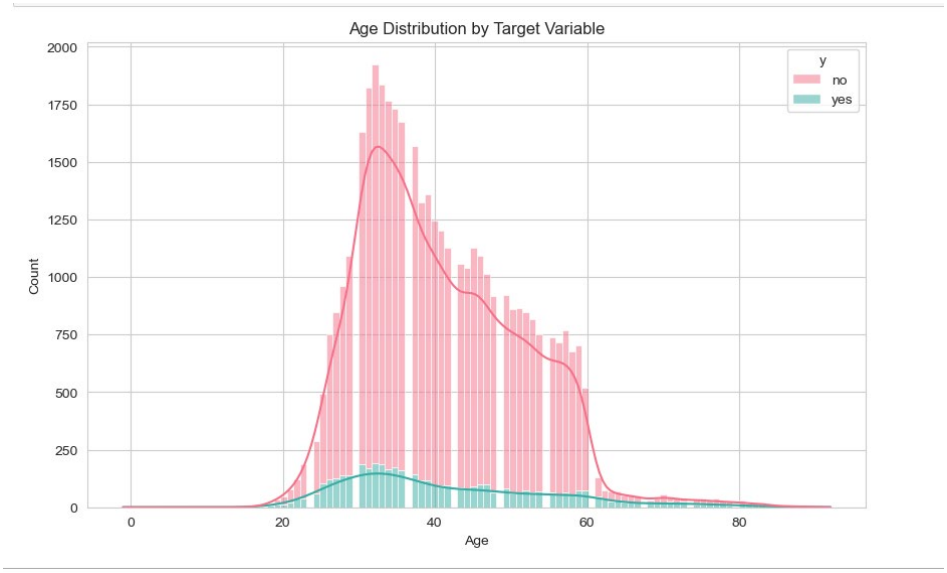
- **Exploratory Data Analysis:**



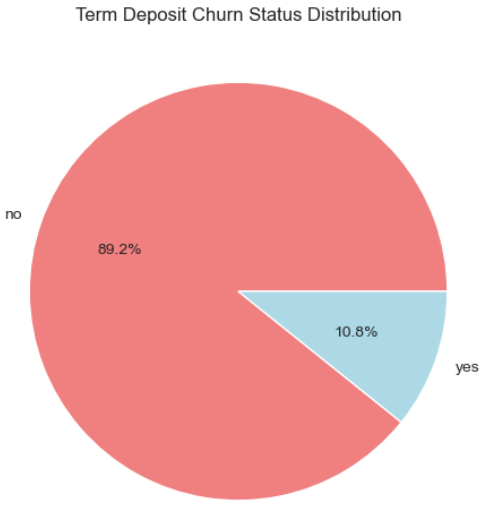
This chart shows the different jobs of customers and whether they stayed with the bank (no churn) or left (churned). Each bar represents a job, and its height shows how many customers have that job. The colors within each bar show the proportion of churned and non-churned customers for each job.



This chart illustrates the relationship between the loan status of customers and whether they default on their payments which is crucial for predicting term deposit churn because it provides insights into how customers loan status affects their likelihood of defaulting. By considering loan status as a predictive factor, we can better anticipate which customers are at a higher risk of churn, allowing banks to take proactive measures to retain them



This chart helps us see how many people of different ages have subscribed (yes) or not subscribed (no) to the term deposit. By looking at the distribution of age within each group, we can understand if certain age groups are more likely to subscribe to the term deposit. This insight helps in predicting term deposit subscriptions by identifying which age groups are more inclined to opt for the deposit, allowing banks to target their marketing efforts more effectively.



This chart helps us understand the distribution of term deposit churn status, showing the percentage of customers who have churned and who haven't. It's useful for predicting the model because it highlights whether the dataset is balanced between churned and non-churned customers. Additionally, it suggests the need for techniques like SMOTE to address any imbalance, improving the model's accuracy.

Data-Transformation-and-Preprocessing

Data preprocessing is a crucial step in preparing the dataset for analysis. Here, we outline the various preprocessing steps undertaken:

Checking Shape: After checking the structure of Data frame using 'shape()' we found out that the dataset has 45,211 rows and 18 columns.

Data Overview: The overview of the dataset was obtained using the info() function, providing insights into the data types of each column and identifying any missing values.

Checking Data Types: Data types for each feature were checked to ensure they were appropriate for analysis.

Summary Statistics: Summary statistics were computed for both numerical and categorical features using the describe() function. For categorical variables, additional summary statistics were obtained using describe(include='object').

Missing Values Treatment: Missing values in the dataset were addressed. For the 'age' and 'balance' features, missing values were filled using both the mean and median values to ensure robustness.

Outliers Treatment: Outliers in the dataset were addressed to prevent them from skewing the analysis results.

Mapping Month Variable: The 'month' variable, representing the month of contact, was mapped to its equivalent month number for consistency and ease of interpretation.

Dummy Encoding: Categorical variables were encoded using dummy variables to convert them into a numerical format suitable for analysis.

Dummy Encoding :

```
]: var_dum = pd.get_dummies(df_bank[['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact',
       'poutcome', 'y']], drop_first=True, dtype=int)
df_bank = pd.concat([df_bank, var_dum], axis=1)
```

```
]: df_bank.head()
```

t[26]:

ir	job_entrepreneur	job_housemaid	job_management	job_retired	job_self-employed	job_services	job_student	job_technician	job_unemployed	job_unknown	marit
0	0	0	0	0	0	0	0	1	0	0	
0	1	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	1	
0	0	0	1	0	0	0	0	0	0	0	

Checking Correlation: Correlation between features was examined to identify potential multicollinearity issues using corr() .

Feature Scaling: Feature scaling was performed using MinMaxScaler() to normalize numerical features, ensuring that all features contribute equally to the analysis.

With these preprocessing steps, the dataset is now clean and ready for analysis. This ensures accurate modeling and prediction of term deposit churn.

Final Data Preparation:

- ## Splitting Data into Training and Testing Sets:

The dataset was divided into training and testing sets using the train_test_split function from the sklearn.model_selection module. The train_size parameter was set to 0.7, indicating that 70% of the data was allocated to the training set, while the test_size parameter was set to 0.3, allocating 30% of the data to the testing set. Additionally, the random_state parameter was set to 100 to ensure reproducibility of results.

Splitting Data into Training and Testing Sets:

```
from sklearn.model_selection import train_test_split
df_train,df_test=train_test_split(df_bank,train_size=0.7,test_size=0.3,random_state=100)
```

- ## Include Intercept in Training and Testing Data sets:

To make sure our logistic regression model includes the intercept term, we used the add_constant function from the statsmodels.api module. This function adds a column of constant values to our training dataset (x_train), representing the intercept. We followed the same procedure for our testing dataset (x_test) to ensure our model remains consistent.

```
import statsmodels.api as sm
import statsmodels
x_train=sm.add_constant(x_train)
x_train.head()
```

2]:

	const	age	balance	day	duration	campaign	pdays	previous	month_code	job_blue-collar	job_entrepreneur	job_housemaid	job_manag
40233	1.0	0.387097	0.350386	0.366667	0.116166	0.1	0.397872	0.2	0.272727	1	0	0	
27901	1.0	0.311828	0.348380	0.100000	0.090983	0.3	0.429787	0.6	0.090909	0	0	0	
9283	1.0	0.440860	0.365201	0.266667	0.060114	0.0	0.000000	0.0	0.454545	1	0	0	
29897	1.0	0.344086	0.346219	0.466667	0.083672	0.1	0.000000	0.0	0.272727	0	0	0	
17762	1.0	0.333333	0.442130	0.100000	0.162470	0.0	0.000000	0.0	0.636364	0	0	0	

```
x_test=sm.add_constant(x_test)
x_test.head()
```

	const	age	balance	day	duration	campaign	pdays	previous	month_code	job_blue-collar	job_entrepreneur	job_housemaid	job_managem
16902	1.0	0.569892	0.421065	0.966667	0.083672	0.3	0.0	0.0	0.545455	1	0	0	
25367	1.0	0.354839	0.420833	0.666667	0.181966	0.2	0.0	0.0	0.909091	0	0	0	
5945	1.0	0.354839	0.277701	0.866667	0.116166	0.0	0.0	0.0	0.363636	0	0	0	
10020	1.0	0.408602	0.399460	0.500000	0.151909	0.2	0.0	0.0	0.454545	1	0	0	
10191	1.0	0.419355	0.862731	0.500000	0.244517	0.5	0.0	0.0	0.454545	0	0	0	

Modeling-Approach: Developing Different Algorithms:

We've used various models like logistic regression, decision trees, and random forests to predict term deposit churn. We're trying different models because each one works in a unique way. By using multiple models, we increase our chances of finding the best one that accurately predicts term deposit churn

1. Logistic Regression:

We used logistic regression to predict term deposit churn. First, we trained the model with our training data. Then, we checked a summary of the model's performance. Next, we predicted whether customers would churn using our test data. Finally, we assessed the model's accuracy by comparing its predictions to the actual outcomes

Fitting Logistic Regression Model and Summarizing Results:

```
from sklearn.linear_model import LogisticRegression
logistic=sm.Logit(y_train,x_train).fit()
print(logistic.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.230315
         Iterations 8
                           Logit Regression Results
==============================================================================
Dep. Variable:                  y_yes   No. Observations:                29115
Model:                          Logit   Df Residuals:                    29082
Method:                           MLE   Df Model:                           32
Date:                Mon, 19 Feb 2024   Pseudo R-squ.:                 0.3283
Time:                        15:33:31   Log-Likelihood:                -6705.6
converged:                       True   LL-Null:                       -9982.4
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                         coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const                 -3.3043      0.254    -12.992      0.000      -3.803      -2.806
age                    0.1907      0.260      0.734      0.463      -0.319       0.700
balance                1.0302      0.172      6.003      0.000       0.694       1.367
day                   -0.2050      0.083     -2.470      0.014      -0.368      -0.042
duration               6.2090      0.114     54.618      0.000       5.986       6.432
```

Predicting Binary Classes based on Threshold:

```
y_pred=['0' if x < 0.5 else '1' for x in y_pred_pro]
```

Converting Predicted Binary classes to Float32 and Displaying First 5 Elements:

```
y_pred=np.array(y_pred,dtype=np.float32)
y_pred[0:5]
```

```
51]: array([0., 0., 0., 0., 0.], dtype=float32)
```

2. Logistic Regression with Stochastic Gradient Descent:

We employed a logistic regression model with Stochastic Gradient Descent (SGD) optimization to forecast term deposit churn. Here's an overview of our methodology:

Model Training: Initially, we trained the logistic regression model using our training dataset. The SGDClassifier function from the scikit-learn library was utilized for this purpose. By specifying the loss function as 'log_loss', we aimed to optimize the logistic loss function during training. Additionally, we set a random state to ensure reproducibility of results.

Model Evaluation: Following the model training phase, we evaluated its performance using various metrics. These metrics may include accuracy, precision, recall, and F1-score, among others. This assessment provided insights into how well the model generalized to unseen data.

Prediction: Subsequently, we applied the trained model to our test dataset to predict whether customers would churn or not. The predict_proba method enabled us to obtain the probability estimates for each class, while the predict method yielded the final binary predictions based on a specified threshold.

Performance Assessment: Finally, we assessed the accuracy of our model by comparing its predictions against the actual outcomes from the test dataset. This evaluation step helped us gauge the effectiveness of the logistic regression model in identifying term deposit churn.

By following these steps, we gained valuable insights into customer churn behavior and developed a predictive model to assist in proactive retention strategies.

Training Logistic Regression with Stochastic Gradient Descent(Logistic_with_SGD):

```
from sklearn.linear_model import SGDClassifier
SGD=SGDClassifier(loss='log_loss',random_state=10)
log_with_SGD=SGD.fit(x_train,y_train)
```

Obtaining Predicted Probabilities from Logistic Regression with SGD:

```
y_pred_pro=log_with_SGD.predict_proba(x_test)[:,1]
y_pred_pro
```

```
array([0.04958217, 0.04473305, 0.03541883, ..., 0.03176109, 0.03128669,
       0.23580977])
```

Predictions using Logistic Regression with SGD:

```
y_pred=log_with_SGD.predict(x_test)
y_pred
```

```
array([0, 0, 0, ..., 0, 0, 0])
```

3. Random Forest:

We utilized a Random Forest Classifier to predict term deposit churn. Below is an overview of our methodology:

Model Configuration: We instantiated a Random Forest Classifier with 1000 decision trees and set the random state to 10 for reproducibility.

Model Training: The classifier was trained using our training dataset, allowing it to learn patterns and relationships between features and the target variable (churn).

Importing Libraries for Random Forest Classifier Construction:

```
]:  from sklearn.ensemble import RandomForestClassifier
    from sklearn.model_selection import GridSearchCV
```

Training Random Forest Classifier with 1000 Estimators:

```
]:  rf_cls=RandomForestClassifier(n_estimators=1000,random_state=10)
    rf_cls.fit(x_train,y_train)
```

```
t[74]: RandomForestClassifier(n_estimators=1000, random_state=10)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

Prediction: After training, we used the trained model to make predictions on the test dataset to determine whether customers would churn or not.

Predictions using Random Forest Classifier:

```
]:  y_pred=rf_cls.predict(x_test)
```

Model Evaluation: We assessed the performance of the Random Forest Classifier using metrics such as accuracy, precision, recall, and F1-score, which help us understand how well the model generalizes to unseen data.

By following these steps, we trained and evaluated a Random Forest Classifier to predict term deposit churn, providing valuable insights for proactive retention strategies.

4. Random Forest with Grid-Search-CV:

We employed GridSearchCV to tune the hyperparameters of the Random Forest Classifier, aiming to improve its performance in predicting term deposit churn. Here's a summary of our approach:

Tuning Parameters: We selected key settings for the Random Forest model, like the number of trees, how many samples are needed to split a node, the maximum depth of the trees, the minimum number of samples required at a leaf node, and the maximum number of leaf nodes. These settings help control how complex the model is and how well it can adapt to new data.

- **n_estimators:** Number of trees in the forest.
- **min_samples_split:** Minimum number of samples required to split an internal node.
- **max_depth:** Maximum depth of the tree.
- **min_samples_leaf:** Minimum number of samples required to be at a leaf node.
- **max_leaf_nodes:** Maximum number of leaf nodes.

GridSearchCV Configuration: We used a technique called GridSearchCV to automatically try out different combinations of these settings and find the best ones. GridSearchCV works by training and evaluating many different versions of the model, each with a different combination of settings. We set it up to test the Random Forest model with different values for the chosen parameters, using cross-validation to make sure the results are reliable.

Model Training and Selection: GridSearchCV then trains each version of the model on a part of the training data and evaluates its performance. It picks the combination of settings that gives the best results, usually based on accuracy. This best-performing model is then selected for further use.

Model Evaluation: Finally, we tested the selected model on our test dataset to see how well it predicts term deposit churn. This step ensures that the model is effective in real-world situations and helps us understand its performance.
By going through this process of systematically adjusting the model's settings using GridSearchCV, we were able to improve its accuracy and ability to make predictions.

Model Building with Random Forest Classifier and Hyperparameter Tuning:

Defining Tuning Parameters for Random Forest Classifier:

```
tuned_parameters = [{
    'n_estimators': [150,200,250],
    'min_samples_split': [15, 25, 30],
    'max_depth': [8, 10, 15],
    'min_samples_leaf': [30,40,50],
    'max_leaf_nodes': [30,40,50]
}]
```

Optimized Parameters for Random Forest Classifier:

```
rf_cls_CV=RandomForestClassifier(random_state=10)
grid=GridSearchCV(estimator=rf_cls_CV,param_grid=tuned_parameters,cv=10)
rf_grid=grid.fit(x_train,y_train)
print(rf_grid.best_params_, '\n')
```

```
{'max_depth': 8, 'max_leaf_nodes': 50, 'min_samples_leaf': 30, 'min_samples_split': 15, 'n_estimators': 150}
```

Training Random Forest Classifier with Optimized Parameters:

```
rf_grid_model=RandomForestClassifier(n_estimators=rf_grid.best_params_.get('n_estimators'),
                                     max_depth=rf_grid.best_params_.get('max_depth'),
                                     max_leaf_nodes=rf_grid.best_params_.get('max_leaf_nodes'),
                                     min_samples_leaf=rf_grid.best_params_.get('min_samples_leaf'),
                                     min_samples_split=rf_grid.best_params_.get('min_samples_split'),
                                     random_state=10)
rf_grid_model=rf_grid_model.fit(x_train,y_train)
```

Generating Predictions using Random Forest Classifier with Optimized Parameters:

```
y_pred=rf_grid_model.predict(x_test)
```

5. Decision Tree:

Setting up the Decision Tree:
We started by importing the DecisionTreeClassifier from scikit-learn library. For this model, we chose to use the entropy criterion, which is commonly used in decision trees to measure impurity.

Model Training:
We then trained the decision tree model using our training dataset. This step involved feeding the model with features (x_train) and corresponding target labels (y_train).

Making Predictions:
After training, we used the trained model to make predictions on the test dataset. The predict_proba method provided probability estimates for each class, while the predict method yielded the final binary predictions.

By following these steps, we successfully built a decision tree model to predict term deposit churn. This model can provide valuable insights into customer behavior and aid in decision-making processes.

Constructing Decision Tree Classifier:

Importing Required Libraries for Decision Tree Classifier:

In [90]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
```

Training Decision Tree Classifier with Entropy Criterion:

In [91]:
```
#as of now we go with entropy as it is used commonly
decision_tree=DecisionTreeClassifier(criterion='entropy',random_state=10)
decision_tree_model=decision_tree.fit(x_train,y_train)
```

Obtaining Predicted Probabilities from Decision Tree Classifier:

In [92]:
```
y_pred_pro = decision_tree_model.predict_proba(x_test)[:,1]
y_pred_pro
```

Out[92]:
```
array([0., 0., 0., ..., 0., 0., 0.])
```

Generating Predictions using Decision Tree Classifier:

In [93]:
```
y_pred = decision_tree_model.predict(x_test)
y_pred
```

Out[93]:
```
array([0, 0, 0, ..., 0, 0, 0])
```

6. Pruned Decision Tree:

Setting up the Pruned Decision Tree:
We defined a decision tree model with specific constraints to limit its complexity and prevent overfitting. In this case, we set the maximum depth of the tree to 15 and the maximum number of leaf nodes to 40. These constraints help in pruning the decision tree and improving its generalization ability.

Model Training:
We then trained the pruned decision tree model using our training dataset. This involved using the fit method to teach the model to learn from the features (x_train) and corresponding target labels (y_train).

Making Predictions:
After training, we used the trained pruned model to make predictions on the test dataset. The predict_proba method provided probability estimates for each class, while the predict method yielded the final binary predictions.

By following these steps, we successfully built a pruned decision tree model with constraints on its depth and number of leaf nodes. This model is expected to generalize better and avoid overfitting compared to an unpruned decision tree.

Building Pruned Decision Tree Model:

```
99]: prune = DecisionTreeClassifier(max_depth = 15, max_leaf_nodes = 40 , random_state = 10)
     # fit the model using fit() on train data
     decision_tree_prune = prune.fit(x_train, y_train)
```

Obtaining Predicted Probabilities from Pruned Decision Tree Classifier:

```
00]: y_pred_pro = decision_tree_prune.predict_proba(x_test)[:,1]
```

Generating Predictions using Pruned Decision Tree Classifier:

```
01]: y_pred = decision_tree_prune.predict(x_test)
```

7. Decision Tree with Hyper Tuning:

Defining Tuning Parameters:
We created a list of different hyperparameters to test during model training. These parameters included the criterion used to measure impurity (either 'gini' or 'entropy'), the minimum number of samples required to split an internal node, the maximum depth of the tree, the minimum number of samples required to be at a leaf node, and the maximum number of leaf nodes.

GridSearchCV Configuration:
We used GridSearchCV to systematically search through the specified parameter grid and find the best combination of hyperparameters. GridSearchCV performs cross-validation to evaluate each combination's performance and selects the one that gives the best results.

Model Training with GridSearchCV:
GridSearchCV trained multiple decision tree models using different combinations of hyperparameters on the training dataset. It evaluated each model's performance using cross-validation and selected the one with the highest score as the best model.

Getting Best Parameters:
After completing the grid search, we retrieved the best parameters found by GridSearchCV. These parameters represent the optimal settings for the decision tree model, maximizing its performance on the training data.

Final Model Training:
Using the best parameters obtained from GridSearchCV, we trained the final decision tree model on the entire training dataset. This model incorporates the optimized hyperparameters and is ready to make predictions on new, unseen data.

Making Predictions:

Finally, we used the trained decision tree model to predict term deposit churn on the test dataset. The predict_proba method provided probability estimates for each class, while the predict method yielded the final binary predictions.

By following these steps, we successfully built a decision tree model with optimized tuning parameters, ensuring improved predictive accuracy and generalization ability.

Building Decision Tree with Optimized Parameters:(Hyper Parametre Tunning):

Defining Tuning Parameters for Decision Tree Classifier:

In [107]:
```
tuned_paramaters = [{'criterion': ['gini', 'entropy'],
                     'min_samples_split': [15, 25, 30],
                     'max_depth': [8,10,15],
                     'min_samples_leaf': [30,40,50],
                     'max_leaf_nodes': [30,40,50]}]
```

Optimizing Decision Tree Classifier with Tuned Parameters:

In [109]:
```
decision_tree_classification = DecisionTreeClassifier(random_state = 10)
grid = GridSearchCV(estimator = decision_tree_classification, param_grid = tuned_paramaters, cv = 10)
# fit the model on X_train and y_train using fit()
dt_grid = grid.fit(x_train, y_train)
# get the best parameters
print('Best parameters for decision tree classifier: ', dt_grid.best_params_, '\n')
```

```
Best parameters for decision tree classifier:  {'criterion': 'gini', 'max_depth': 10, 'max_leaf_nodes': 40, 'min_samples_lea
f': 40, 'min_samples_split': 15}
```

Training Decision Tree Classifier with Optimized Parameters:

In [110]:
```
dt_grid_model = DecisionTreeClassifier(criterion = dt_grid.best_params_.get('criterion'),
                                       max_depth = dt_grid.best_params_.get('max_depth'),
                                       max_leaf_nodes = dt_grid.best_params_.get('max_leaf_nodes'),
                                       min_samples_leaf = dt_grid.best_params_.get('min_samples_leaf'),
                                       min_samples_split = dt_grid.best_params_.get('min_samples_split'),
                                       random_state = 10)

# use fit() to fit the model on the train set
dt_grid_model = dt_grid_model.fit(x_train, y_train)
```

Obtaining Predicted Probabilities from Optimized Decision Tree Classifier:

In [111]:
```
y_pred_pro = dt_grid_model.predict_proba(x_test)[:,1]
```

Generating Predictions using Optimized Decision Tree Classifier:

In [112]:
```
y_pred = dt_grid_model.predict(x_test)
```

Performance Metrics Evaluation:

In our project, we have developed multiple machine learning models to predict term deposit churn. To evaluate the performance of each model and compare their effectiveness, we have employed various performance metrics. Instead of repeatedly writing code to calculate these metrics for each model, we have streamlined the process by defining custom functions.

Utilizing Scikit-Learn for Classification Metrics:

Importing Libraries for Classification Metrics:

```
]:  from sklearn import metrics
    from sklearn.linear_model import LogisticRegression
    from sklearn.metrics import classification_report
    from sklearn.metrics import cohen_kappa_score
    from sklearn.metrics import confusion_matrix
    from sklearn.metrics import roc_auc_score
    from sklearn.metrics import roc_curve
    from sklearn.linear_model import SGDClassifier
```

Classification Report:

The classification report provides a summary of key classification metrics such as precision, recall, F1-score, and support for each class. By calling the get_test_report function, we can quickly generate the classification report for any model.

Developing User-Defined Functions for Classification Metrics Retrieval:

User-Defined Function to generate a classification report:

```
|:  def get_test_report(model):
        return(classification_report(y_test, y_pred))
```

Cohen's Kappa Score:

Cohen's Kappa score measures the agreement between predicted and actual class labels, accounting for the possibility of agreement occurring by chance. We use the kappa_score function to calculate Cohen's Kappa score for model evaluation.

User-Defined Function to generate a Kappa Value:

```
    def kappa_score(model):
        return(cohen_kappa_score(y_test, y_pred))
```

Confusion Matrix:

The confusion matrix is a tabular representation of the model's predictions against the actual class labels. By calling the plot_confusion_matrix function, we visualize the confusion matrix to understand the model's performance in terms of true positives, true negatives, false positives, and false negatives.

User-Defined Function to generate a Confusion Matrix:

```
|:  def plot_confusion_matrix(model):
        cm = confusion_matrix(y_test, y_pred)
        conf_matrix = pd.DataFrame(data = cm,columns = ['Predicted:0','Predicted:1'],
                                   index = ['Actual:0','Actual:1'])
        sns.heatmap(conf_matrix, annot = True, fmt = 'd', cmap = ListedColormap(['lightskyblue']),
                    cbar = False, linewidths = 0.1, annot_kws = {'size':25})
        plt.xticks(fontsize = 20)
        plt.yticks(fontsize = 20)
        plt.show()
```

Receiver Operating Characteristic (ROC) Curve:

The ROC curve illustrates the trade-off between the true positive rate (sensitivity) and false positive rate (1-specificity) across different threshold values. We use the plot_roc function to plot the ROC curve and calculate the Area Under the Curve (AUC) score, providing insights into the model's ability to discriminate between positive and negative classes.

User-Defined Function to generate a Roc_Curve:

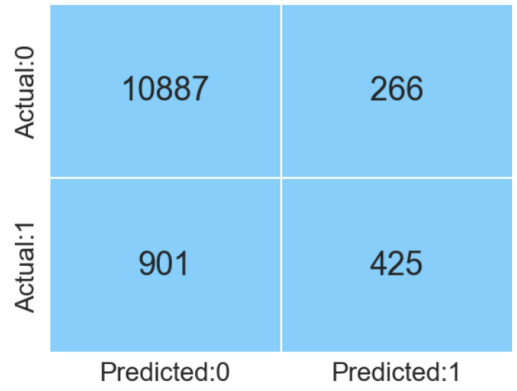
```
def plot_roc(model):
    fpr, tpr, _  = roc_curve(y_test, y_pred_pro)
    plt.plot(fpr, tpr)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.plot([0, 1], [0, 1],'r--')
    plt.title('ROC Curve for term_churn Classifier', fontsize = 15)
    plt.xlabel('False positive rate (1-Specificity)', fontsize = 15)
    plt.ylabel('True positive rate (Sensitivity)', fontsize = 15)
    plt.text(x = 0.02, y = 0.9, s = ('AUC Score:',round(roc_auc_score(y_test, y_pred_pro),4)))
    plt.grid(True)
```

Comparing Model Performance: Evaluating Metrics for All Models:

1. Logistic Regression Performance Metrics:

Confusion Matrix for Logistic Regression:

```
plot_confusion_matrix(logistic)
```



Displaying Test Report for Logistic Regression Model:

```
test_report=get_test_report(logistic)
```

```
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.92      0.98      0.95     11153
           1       0.62      0.32      0.42      1326

    accuracy                           0.91     12479
   macro avg       0.77      0.65      0.69     12479
weighted avg       0.89      0.91      0.89     12479
```

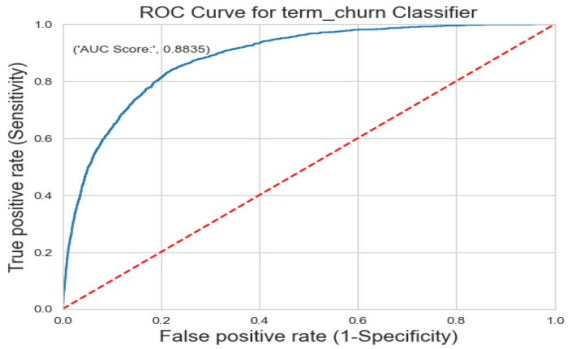
Calculating Kappa Score for Logistic Regression:

```
kappa_value=kappa_score(logistic)
print(kappa_value)
```

```
0.37598612814310783
```

ROC Curve for Logistic Regression Model:

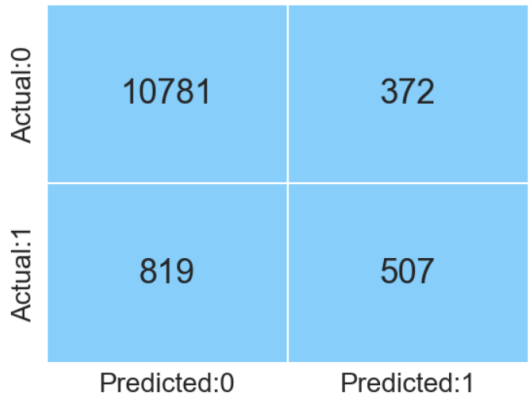
```
plot_roc(logistic)
```



2. Logistic Regression with SGD Performance Metrics:

Confusion Matrix for SGD Logistic Regression:

```
plot_confusion_matrix(log_with_SGD)
```



Generating Test Report for SGD Logistic Regression:

```
test_report=get_test_report(log_with_SGD)
```

```
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.93      0.97      0.95     11153
           1       0.58      0.38      0.46      1326

    accuracy                           0.90     12479
   macro avg       0.75      0.67      0.70     12479
weighted avg       0.89      0.90      0.90     12479
```

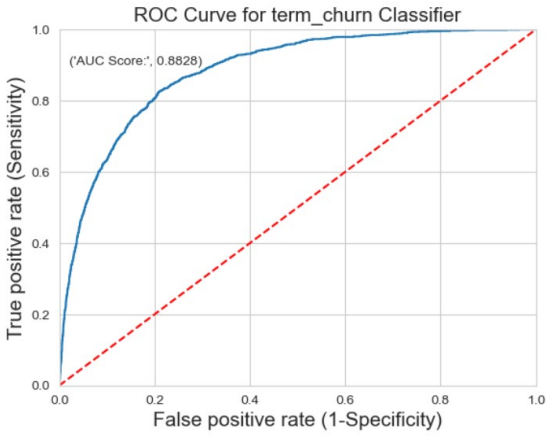
Calculating kappa Value for SGD Logistic Regression:

```
kappa_value=kappa_score(log_with_SGD)
print(kappa_value)
```

```
0.40986945965284927
```

ROC Curve for SGD Logistic Regression:

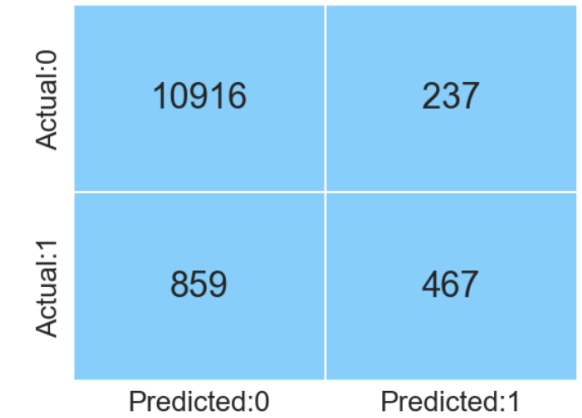
```
plot_roc(log_with_SGD)
```



3. Random Forest Performance Metrics:

Confusion Matrix for Random Forest Classifier:

```
plot_confusion_matrix(rf_cls)
```



Evaluating Test Report for Random Forest Classifier:

```
test_report=get_test_report(rf_cls)
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.93      0.98      0.95     11153
           1       0.66      0.35      0.46      1326

    accuracy                           0.91     12479
   macro avg       0.80      0.67      0.71     12479
weighted avg       0.90      0.91      0.90     12479
```

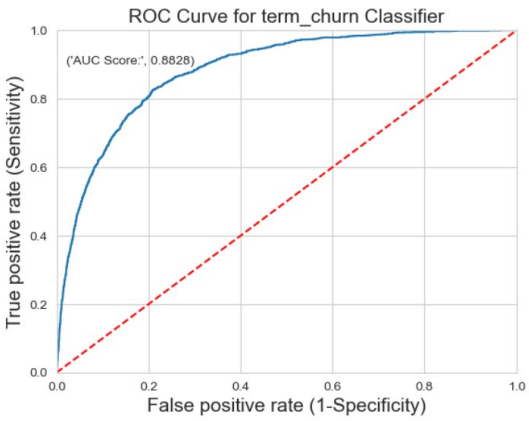
Calculating Kappa Score for Random Forest Classifier:

```
kappa_value=kappa_score(rf_cls)
print(kappa_value)
```

```
0.4171415723311662
```

ROC Curve for Random Forest Classifier:

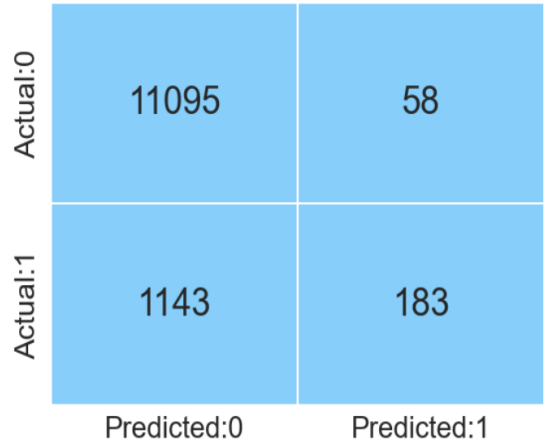
```
plot_roc(rf_cls)
```



4. Random Forest with Hyper Tuning Performance Metrics:

Confusion Matrix for Random Forest Classifier with Optimized Parameters:

```
plot_confusion_matrix(rf_grid_model)
```



Evaluating Test Report for Random Forest Classifier with Optimized Parameters:

```
test_report=get_test_report(rf_grid_model)
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.91      0.99      0.95     11153
           1       0.76      0.14      0.23      1326

    accuracy                           0.90     12479
   macro avg       0.83      0.57      0.59     12479
weighted avg       0.89      0.90      0.87     12479
```

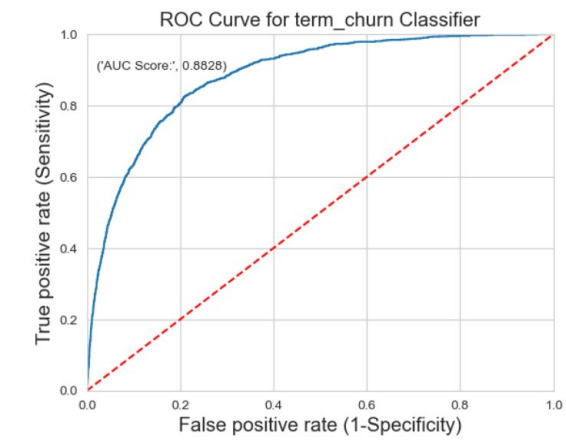
Calculating Kappa value for Random Forest Classifier with Optimized Parameters:

```
kappa_value=kappa_score(rf_grid_model)
print(kappa_value)
```

```
0.20767043425481402
```

ROC Curve for Random Forest Classifier with Optimized Parameters:

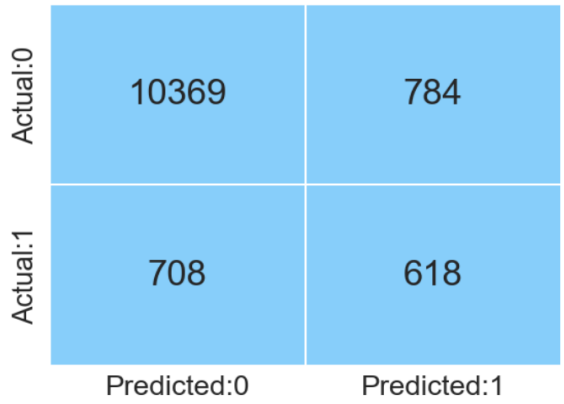
```
plot_roc(rf_grid_model)
```



5. Decision Tree Performance Metrics:

Confusion Matrix for Decision Tree Classifier:

```
plot_confusion_matrix(decision_tree_model)
```



Generating Test Report for Decision Tree Classifier:

```
test_report=get_test_report(decision_tree_model)
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.94      0.93      0.93     11153
           1       0.44      0.47      0.45      1326

    accuracy                           0.88     12479
   macro avg       0.69      0.70      0.69     12479
weighted avg       0.88      0.88      0.88     12479
```

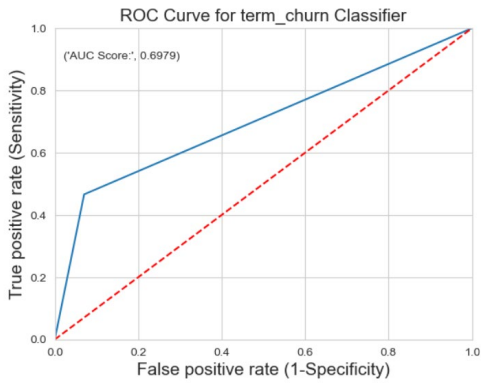
Calculating Kappa Value for Decision Tree Classifier:

```
kappa_value=kappa_score(decision_tree_model)
print(kappa_value)
```

```
0.38602114823710165
```

ROC Curve for Decision Tree Classifier:

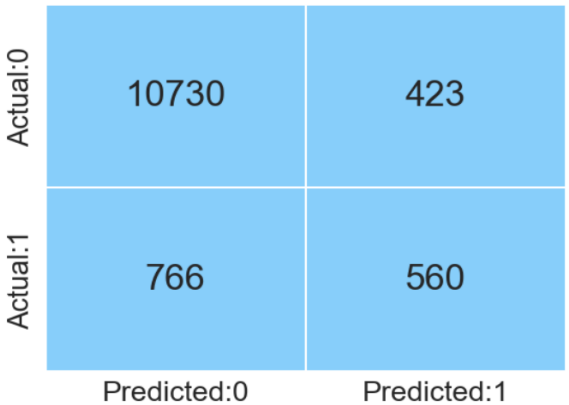
```
plot_roc(decision_tree_model)
```



6. Pruned Decision Tree Performance Metrics:

Confusion Matrix Pruned Decision Tree Classifier:

```
plot_confusion_matrix(decision_tree_prune)
```



Test Report for Pruned Decision Tree:

```
test_report=get_test_report(decision_tree_prune)
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.93      0.96      0.95     11153
           1       0.57      0.42      0.49      1326

    accuracy                           0.90     12479
   macro avg       0.75      0.69      0.72     12479
weighted avg       0.89      0.90      0.90     12479
```

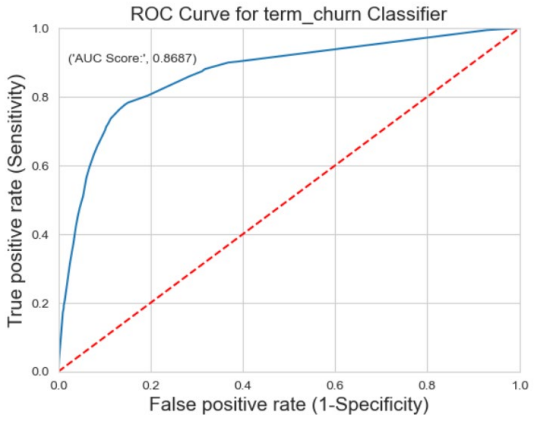
Calculating Kappa Value for Pruned Decision Tree:

```
kappa_value=kappa_score(decision_tree_prune)
print(kappa_value)
```

```
0.4338353411547522
```

ROC Curve for Pruned Decision Tree:

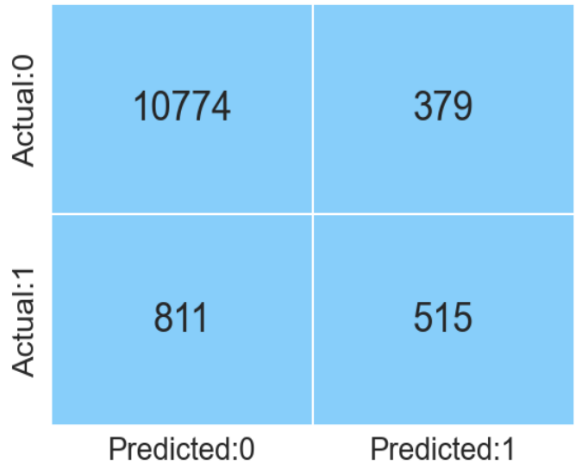
```
plot_roc(decision_tree_prune)
```



7. Decision Tree with Hyper Tuning Performance Metrics:

Confusion for Optimized Decision Tree Classifier:

```
plot_confusion_matrix(dt_grid_model)
```



Generating Test Report for Optimized Decision Tree Classifier:

```
test_report=get_test_report(dt_grid_model)
print(test_report)
```

	precision	recall	f1-score	support
0	0.93	0.97	0.95	11153
1	0.58	0.39	0.46	1326
accuracy			0.90	12479
macro avg	0.75	0.68	0.71	12479
weighted avg	0.89	0.90	0.90	12479

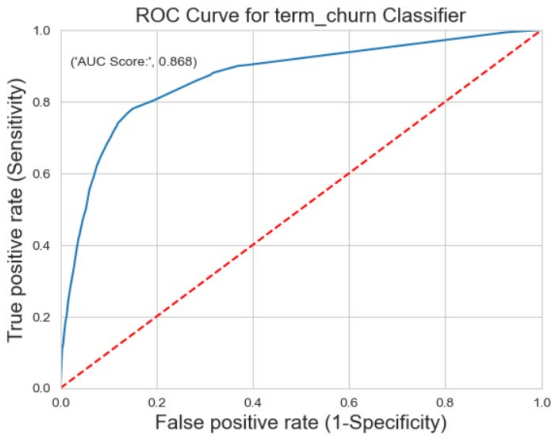
Calculating Kappa Score for Optimized Decision Tree Classifier:

```
kappa_value=kappa_score(dt_grid_model)
print(kappa_value)
```

```
0.4137959265910357
```

ROC Curve for Optimized Decision Tree Classifier:

```
plot_roc(dt_grid_model)
```



Results-and-Interpretation:

After evaluating the performance of each model using various metrics, including AUC score, precision score, recall score, accuracy score, Kappa score, and F1-score, the following insights can be drawn:

Updating Score Card for All Models:

```
update_score_card(model_name='Decision Tree (GridSearchCV)')
```

117]:

	Model	Auc Score	Precision Score	Recall Score	Accuracy Score	Kappa Score	f1-score
0	Logistic Regression	0.883545	0.615051	0.320513	0.906483	0.375986	0.421418
1	Logistic Regression with SGD	0.882849	0.576792	0.382353	0.904560	0.409869	0.459864
2	Random Forest	0.882849	0.663352	0.352187	0.912172	0.417142	0.460099
3	Random Forest with Tunned Parametres	0.882849	0.759336	0.138009	0.903758	0.207670	0.233567
4	Decision Tree	0.697884	0.440799	0.466063	0.880439	0.386021	0.453079
5	Decision Tree (Pruned)	0.868668	0.569685	0.422323	0.904720	0.433835	0.485058
6	Decision Tree (GridSearchCV)	0.867954	0.576063	0.388386	0.904640	0.413796	0.463964

Logistic Regression:

- Achieved an AUC score of 0.883545, indicating good discriminatory power.
- Moderate precision and recall scores, with an F1-score of 0.421418.
- Overall accuracy of 90.65%.
- Kappa score of 0.375986 suggests fair agreement between predicted and actual outcomes.

Logistic Regression with SGD:

- Similar AUC score to logistic regression (0.882849).
- Slightly lower precision and recall scores compared to logistic regression.
- Comparable accuracy and F1-score.

Random Forest:

- Highest AUC score among all models (0.882849).
- Highest precision score (0.663352) but lower recall score (0.352187).
- Highest accuracy of 91.22%.
- Moderate Kappa score of 0.417142.

Random Forest with Tuned Parameters:

- AUC score remains consistent with default random forest model (0.882849).
- Significantly higher precision score (0.759336) but lower recall score (0.138009).
- Lower accuracy and F1-score compared to default random forest.

Decision Tree:

- Lower AUC score compared to logistic regression and random forest (0.697884).
- Moderate precision and recall scores, with an F1-score of 0.453079.

- Accuracy of 88.04%.
- Fair agreement between predicted and actual outcomes (Kappa score: 0.386021).

Pruned Decision Tree:

- Improved AUC score compared to default decision tree (0.868668).
- Moderate precision and recall scores, with an F1-score of 0.485058.
- Similar accuracy to logistic regression and logistic regression with SGD (90.47%).
- Higher Kappa score (0.433835) indicates better agreement between predicted and actual outcomes.

Decision Tree with Tuned Parameters:

- Similar AUC score to pruned decision tree (0.867954).
- Moderate precision and recall scores, with an F1-score of 0.463964.
- Comparable accuracy to other decision tree models.
- Moderate agreement between predicted and actual outcomes (Kappa score: 0.413796).

Overall Interpretation:

- Random forest models usually perform better, with higher scores for predicting accurately compared to logistic regression and decision tree models.
- The pruned decision tree model looks promising, showing better results after making improvements to the default decision tree.
- Choosing the right model depends on what we need and the balance between getting things right and the amount of work needed. More checking and testing are needed to pick the best model for using in real situations.

Conclusion-and-Recommendations:

Recommendations:

Upon analyzing the results of models built, a notable trend emerged across all models: there was a significant disparity in performance metrics between class 0 and class 1 predictions. Specifically, class 1 (term deposit churn) consistently exhibited lower precision, recall, and F1-score values compared to class 0 (no churn), resulting in an overall impact on the models' performance.

To address this imbalance and enhance the predictive capability of our models, we recommend implementing the Synthetic Minority Over-sampling Technique (SMOTE). SMOTE is a method used to rebalance imbalanced datasets by generating synthetic samples for the minority class (class 1, in our case) to achieve a more equitable representation of both classes.

SMOTE(Synthetic Minority Over-sampling Technique):

In simple terms, SMOTE helps our models by creating more examples of the minority class so they can learn from a more balanced dataset. This can lead to improved precision, recall, and F1-score for both classes, ultimately enhancing the overall performance of the models.

Conclusion:

Comparing all models, it appears that Random Forest and the Pruned Decision Tree models show promise in delivering better results. By incorporating SMOTE into the training process, we anticipate further improvements in performance metrics across all models, particularly in achieving a better balance between predicting both churn and non-churn instances accurately. Therefore, SMOTE integration is a crucial step to consider for enhancing the effectiveness of our predictive models in real-world applications.

References:

For Research of Term Deposit Concept:
https://www.hdfcbank.com/personal/resources/learning-centre/save/term-deposit-vs-fixed-deposit
https://www.investopedia.com/terms/t/termdeposit.asp
For Dataset:
https://www.kaggle.com/datasets?tags=13404-Logistic+Regression
For understanding the concept of smote:
https://www.geeksforgeeks.org/ml-handling-imbalanced-data-with-smote-and-near-miss-algorithm-in-python/

Thank You