1. Overview

Based on the descriptive and exploratory analysis done in notebook 00_data_understanding, this Python Script will work on 2 models: logistic and decission tree classifier, we will chose the best model based on the one that has better evaluation metrics. We will then improve the chosen model with tuned hyperparameters.

2. Data Understanding

2.1 Data Description

This notebook will use the dataset: df_data_processed excel sheet created in the previous notebook: 01_data_preprocessing

2.2 Import Necessary Libraries

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from sklearn.exceptions import ConvergenceWarning


from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, roc_auc_score

import pickle
import warnings
warnings.filterwarnings('ignore', category=ConvergenceWarning)
```

3. Code

3.1 Import the database

In [2]:
```
df = pd.read_excel('df_data_processed.xlsx')
df.head()
```

Out[2]:

	amount_tsh	gps_height	population	basin	region	extraction_type_class	payment_type
0	-0.084999	2.053863	-0.041306	-0.540016	-0.633090	-0.521411	-0.897587
1	-0.100621	-0.965049	-0.379739	-0.540016	0.555492	-0.463637	0.771866
2	-0.100621	-0.965049	-0.379739	1.471270	0.131062	2.617222	0.771866
3	-0.100621	-0.965049	-0.379739	-1.053126	0.131062	-0.521411	-1.330306
4	-0.006889	0.511216	-0.125914	0.697368	0.135714	2.617222	-0.641415

5 rows × 36 columns

In [3]:
```
df.shape
```

Out[3]: (59400, 36)

3.2 Import the database

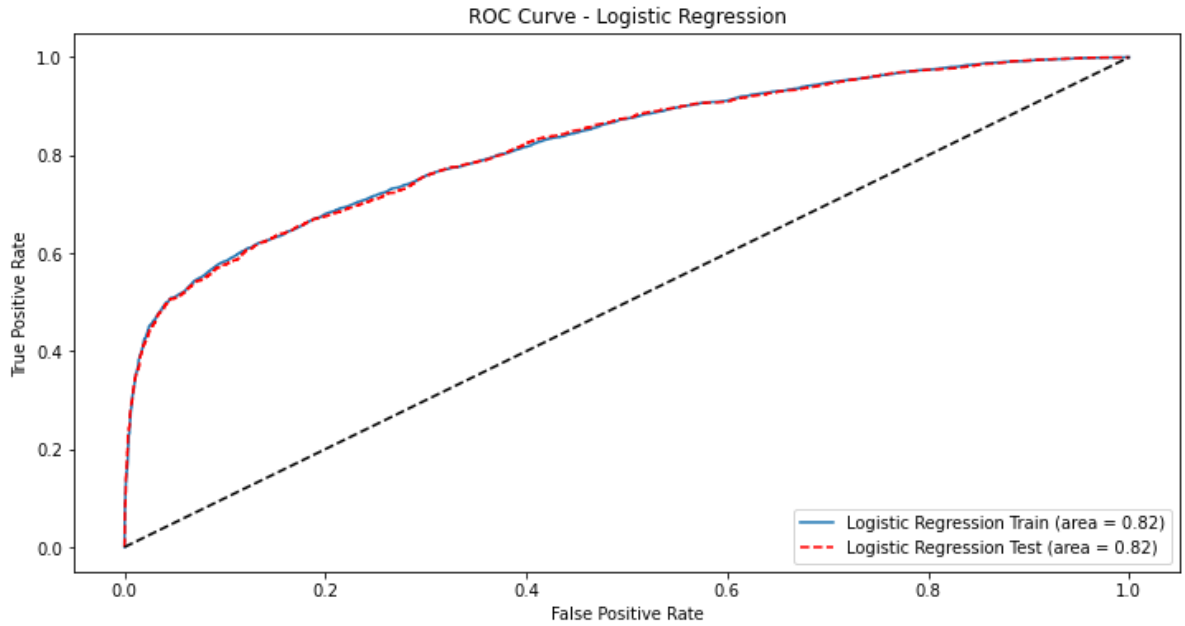
In [4]:
```
df_train = df[df['is_test']==0]
df_test = df[df['is_test']==1]
```

In [5]:
```
y_train = df_train['status_group']
X_train = df_train.drop(['status_group','is_test'], axis=1)

y_test = df_test['status_group']
X_test = df_test.drop(['status_group','is_test'], axis=1)
```

3.3 Baseline model creations

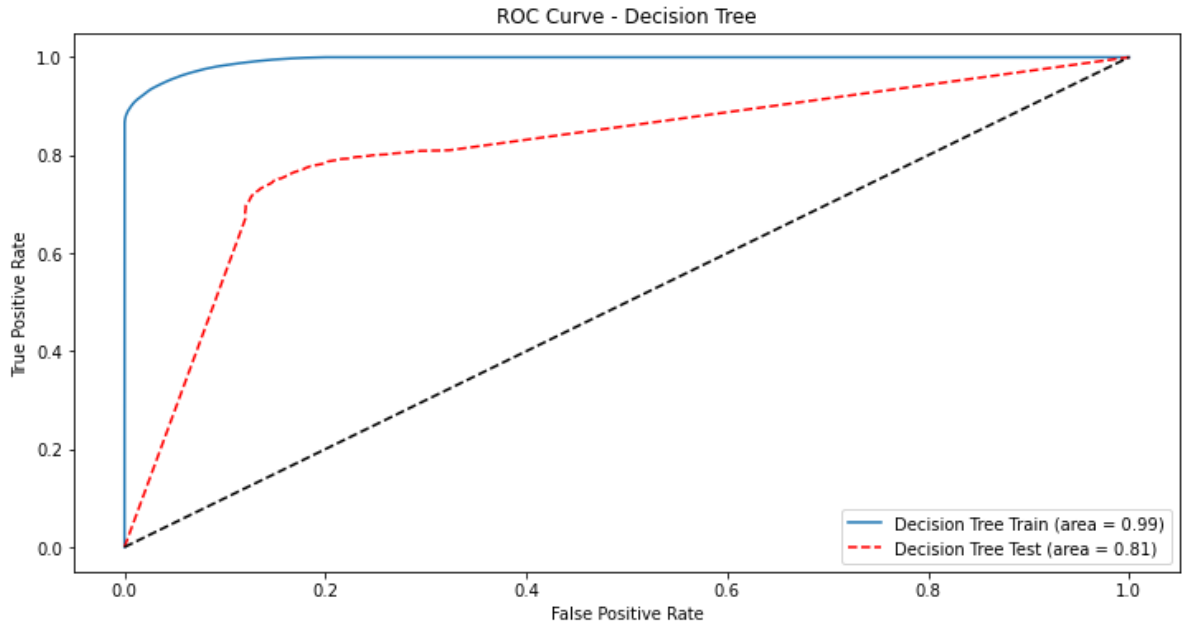
3.3.1 Logistic regression

In [6]:
```
# Initialize the Logistic Regression model
log_reg = LogisticRegression()

# Fit the model to the training data
log_reg.fit(X_train, y_train)

# Predict probabilities on the training and test set
y_pred_prob_log_reg_train = log_reg.predict_proba(X_train)[:, 1]  # Training p
y_pred_prob_log_reg_test = log_reg.predict_proba(X_test)[:, 1]  # Test probabi

# Compute ROC curve and AUC for training data
fpr_log_reg_train, tpr_log_reg_train, _ = roc_curve(y_train, y_pred_prob_log_r
auc_log_reg_train = auc(fpr_log_reg_train, tpr_log_reg_train)

# Compute ROC curve and AUC for test data
fpr_log_reg_test, tpr_log_reg_test, _ = roc_curve(y_test, y_pred_prob_log_reg_
auc_log_reg_test = auc(fpr_log_reg_test, tpr_log_reg_test)

# Plotting ROC Curves
plt.figure(figsize=(12, 6))
plt.plot(fpr_log_reg_train, tpr_log_reg_train, label='Logistic Regression Trai
plt.plot(fpr_log_reg_test, tpr_log_reg_test, color='red', linestyle='--', labe
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Logistic Regression')
plt.legend(loc="lower right")
plt.show()
```



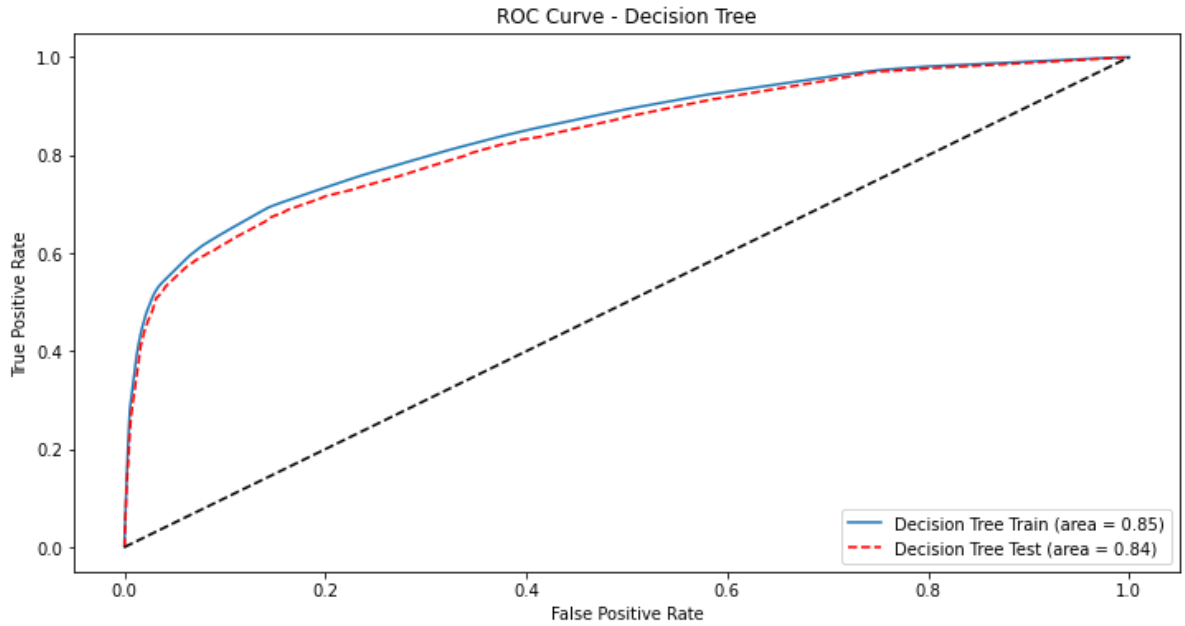
3.3.2 Decision Tree

In [7]:
```
# Initialize the Decision Tree model
decision_tree = DecisionTreeClassifier()

# Fit the model to the training data
decision_tree.fit(X_train, y_train)

# Predict probabilities on the training and test set
y_pred_prob_tree_train = decision_tree.predict_proba(X_train)[:, 1]  # Trainin
y_pred_prob_tree_test = decision_tree.predict_proba(X_test)[:, 1]  # Test prob

# Compute ROC curve and AUC for training data
fpr_tree_train, tpr_tree_train, _ = roc_curve(y_train, y_pred_prob_tree_train)
auc_tree_train = auc(fpr_tree_train, tpr_tree_train)

# Compute ROC curve and AUC for test data
fpr_tree_test, tpr_tree_test, _ = roc_curve(y_test, y_pred_prob_tree_test)
auc_tree_test = auc(fpr_tree_test, tpr_tree_test)

# Plotting ROC Curves
plt.figure(figsize=(12, 6))
plt.plot(fpr_tree_train, tpr_tree_train, label='Decision Tree Train (area = {:
plt.plot(fpr_tree_test, tpr_tree_test, color='red', linestyle='--', label='Dec
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Decision Tree')
plt.legend(loc="lower right")
plt.show()
```



checking max_depth to mitigate overfitting

In [8]:
```
# Initialize the Decision Tree model
decision_tree = DecisionTreeClassifier(max_depth=7)

# Fit the model to the training data
decision_tree.fit(X_train, y_train)

# Predict probabilities on the training and test set
y_pred_prob_tree_train = decision_tree.predict_proba(X_train)[:, 1]  # Trainin
y_pred_prob_tree_test = decision_tree.predict_proba(X_test)[:, 1]  # Test prob

# Compute ROC curve and AUC for training data
fpr_tree_train, tpr_tree_train, _ = roc_curve(y_train, y_pred_prob_tree_train)
auc_tree_train = auc(fpr_tree_train, tpr_tree_train)

# Compute ROC curve and AUC for test data
fpr_tree_test, tpr_tree_test, _ = roc_curve(y_test, y_pred_prob_tree_test)
auc_tree_test = auc(fpr_tree_test, tpr_tree_test)

# Plotting ROC Curves
plt.figure(figsize=(12, 6))
plt.plot(fpr_tree_train, tpr_tree_train, label='Decision Tree Train (area = {:
plt.plot(fpr_tree_test, tpr_tree_test, color='red', linestyle='--', label='Dec
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Decision Tree')
plt.legend(loc="lower right")
plt.show()
```



3.4 Hyper tuning with Decision Tree Classifier

3.4.1 Decision Tree Classifier

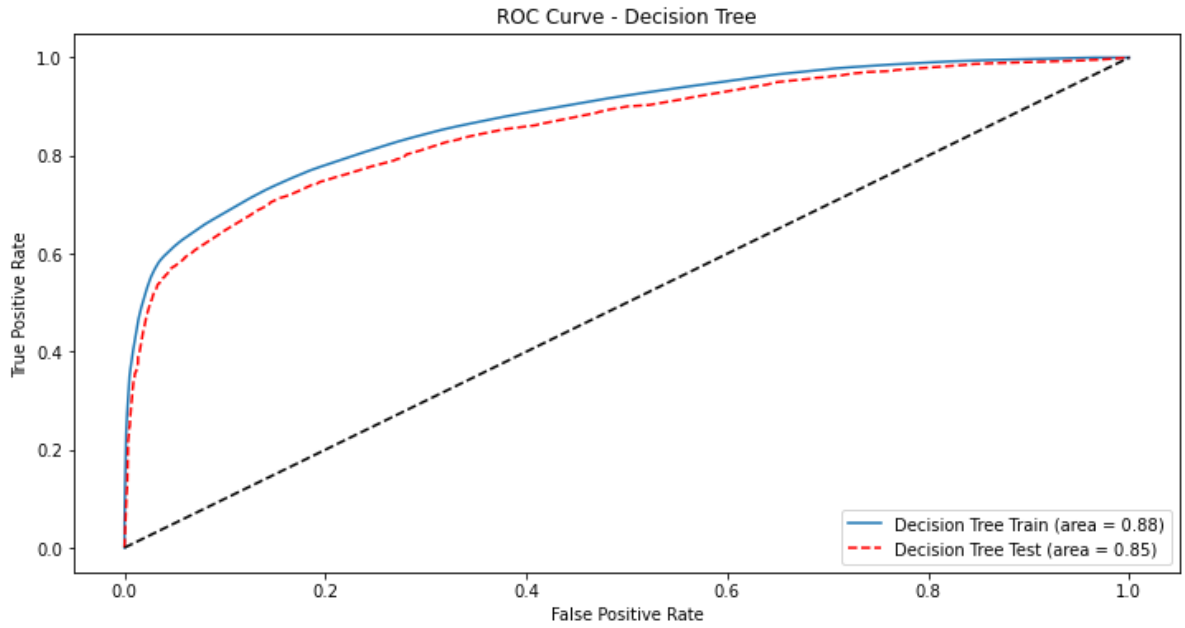
We are going to do hyper parameter tuning with Decision Tree classifier and the Logistic regression and we will keep the model that gives the best results

In [9]:
```
# Initialize the Decision Tree model
decision_tree = DecisionTreeClassifier()

# Define the parameter grid to search
param_grid = {
    'max_depth': range(5, 10),  # Explore depths from 1 to 20
    'min_samples_split': range(5, 15, 2),  # Minimum number of samples require
    'min_samples_leaf': range(5, 10),  # Minimum number of samples required to
    'max_features': ['auto', 'log2', None]  # Number of features to consider w
}

# Define the scoring function using AUC
scorer = make_scorer(roc_auc_score, needs_proba=True)

# Setup the grid search with cross-validation
grid_search = GridSearchCV(estimator=decision_tree, param_grid=param_grid, sco

# Fit grid search on the training data
grid_search.fit(X_train, y_train)

# Find the best model
best_tree = grid_search.best_estimator_

# Save the best_tree
pickle.dump(best_tree, open(f"model_objects/best_tree.pickle", 'wb'))

# Output the best parameter combination and the corresponding score
print("Best parameters found:", grid_search.best_params_)
print("Best AUC achieved:", grid_search.best_score_)

# Optional: Evaluate the best model on the test set
y_pred_proba_best_tree = best_tree.predict_proba(X_test)[:, 1]
test_auc = roc_auc_score(y_test, y_pred_proba_best_tree)
print("Test AUC of best model:", test_auc)
```

```
Best parameters found: {'max_depth': 9, 'max_features': None, 'min_samples_le
af': 8, 'min_samples_split': 5}
Best AUC achieved: 0.8537600383302213
Test AUC of best model: 0.8546157245191096
```

Let's do the curve ROC and see the values AUC with the values for this Decision TreeClassifier

In [10]:
```
# Predict probabilities on the training and test set
y_pred_prob_tree_train = best_tree.predict_proba(X_train)[:, 1]  # Training pr
y_pred_prob_tree_test = best_tree.predict_proba(X_test)[:, 1]  # Test probabil

# Compute ROC curve and AUC for training data
fpr_tree_train, tpr_tree_train, _ = roc_curve(y_train, y_pred_prob_tree_train)
auc_tree_train = auc(fpr_tree_train, tpr_tree_train)

# Compute ROC curve and AUC for test data
fpr_tree_test, tpr_tree_test, _ = roc_curve(y_test, y_pred_prob_tree_test)
auc_tree_test = auc(fpr_tree_test, tpr_tree_test)

# Plotting ROC Curves
plt.figure(figsize=(12, 6))
plt.plot(fpr_tree_train, tpr_tree_train, label='Decision Tree Train (area = {:
plt.plot(fpr_tree_test, tpr_tree_test, color='red', linestyle='--', label='Dec
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Decision Tree')
plt.legend(loc="lower right")
plt.show()
```



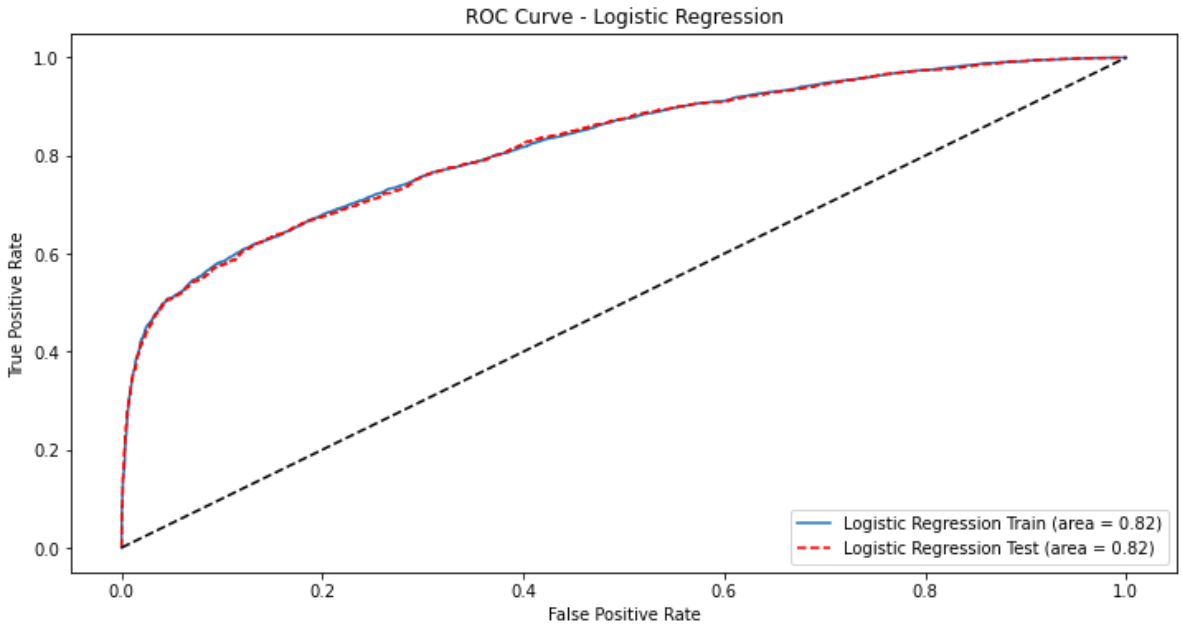
3.4.2 logistic regression

In [11]:
```
# Initialize the Logistic Regression model
logistic_regression = LogisticRegression()

# Define the parameter grid to search
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Inverse of regularization strength
    'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],  # Algorithm
    'max_iter': [100, 200, 300],  # Maximum number of iterations taken for the
}

# Define the scoring function using AUC
scorer = make_scorer(roc_auc_score, needs_proba=True)

# Setup the grid search with cross-validation
grid_search = GridSearchCV(estimator=logistic_regression, param_grid=param_gri

# Fit grid search on the training data
grid_search.fit(X_train, y_train)

# Find the best model
best_log_reg = grid_search.best_estimator_

# Output the best parameter combination and the corresponding score
print("Best parameters found:", grid_search.best_params_)
print("Best AUC achieved:", grid_search.best_score_)

# Optional: Evaluate the best model on the test set
y_pred_proba_best_log_reg = best_log_reg.predict_proba(X_test)[:, 1]
test_auc = roc_auc_score(y_test, y_pred_proba_best_log_reg)
print("Test AUC of best model:", test_auc)
```

```
Best parameters found: {'C': 100, 'max_iter': 100, 'solver': 'newton-cg'}
Best AUC achieved: 0.8226230640046135
Test AUC of best model: 0.8225125325569935
```

Let's do the curve ROC and see the values AUC with the values for this Logistic Regressor

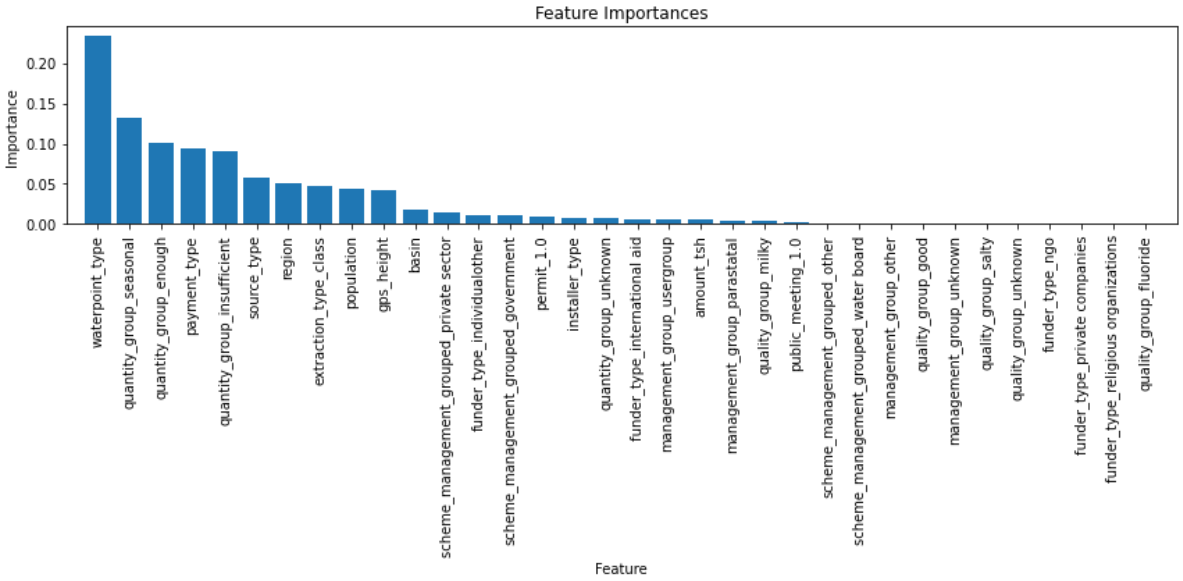
In [12]:
```
# Predict probabilities on the training and test set using the Logistic Regres
y_pred_prob_log_reg_train = best_log_reg.predict_proba(X_train)[:, 1]  # Train
y_pred_prob_log_reg_test = best_log_reg.predict_proba(X_test)[:, 1]  # Test pr

# Compute ROC curve and AUC for training data
fpr_log_reg_train, tpr_log_reg_train, _ = roc_curve(y_train, y_pred_prob_log_r
auc_log_reg_train = auc(fpr_log_reg_train, tpr_log_reg_train)

# Compute ROC curve and AUC for test data
fpr_log_reg_test, tpr_log_reg_test, _ = roc_curve(y_test, y_pred_prob_log_reg_
auc_log_reg_test = auc(fpr_log_reg_test, tpr_log_reg_test)

# Plotting ROC Curves
plt.figure(figsize=(12, 6))
plt.plot(fpr_log_reg_train, tpr_log_reg_train, label='Logistic Regression Trai
plt.plot(fpr_log_reg_test, tpr_log_reg_test, color='red', linestyle='--', labe
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Logistic Regression')
plt.legend(loc="lower right")
plt.show()
```



4. Feature importance

We are now going to execute a feature importance code to be able to see the level of importance of all variables when doing the predictions

In [13]:
```
# Obtain the most important features affecting the status of a pump
importances = best_tree.feature_importances_

# Obtener los nombres de las características
feature_names = X_train.columns

# Create a bar graph for the importance of the characteristics
indexes = np.argsort(importances)[::-1]  # Order importances in descending ord

plt.figure(figsize=(12, 6))
plt.title("Feature Importances")
plt.bar(range(X_train.shape[1]), importances[indexes], align="center")
plt.xticks(range(X_train.shape[1]), feature_names[indexes], rotation=90)
plt.xlim([-1, X_train.shape[1]])
plt.xlabel("Feature")
plt.ylabel("Importance")
plt.tight_layout()
plt.show()
```



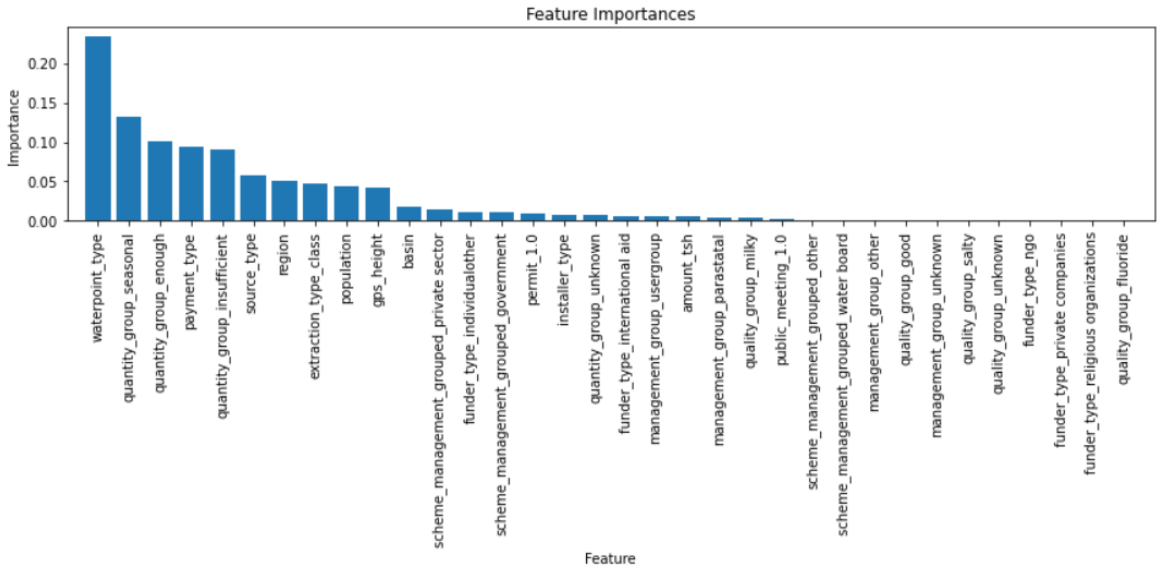
5. Conclusion

The conclusion of this report is that the best AUC is obtained with a Decision Tree Classifier. As is observable, the AUC is of 0.85 for the test. In the case of the Logistic Regression model, an AUC of 0.82 is obtained for the test.

The variables that are most important and that permit us to best descriminate are:

1. waterpoint_type
2. quantity_group_seasonal
3. quantity_group_enough
4. payment_type
5. quantity_group_insufficient

We are interested in these 5 variables because they are the ones that have the most influence when determining if a pump is functional or non-functional.



Considering that we used a one-hot encoder and that the categories for each variable were treated as independent variables, the three variables that contribute the most to the model are:

1. waterpoint_type
2. quantity_group
3. payment_type

Here we will show the contingency tables for each variable divided into functional, functional with repairs, and non functional pumps:

Table for payment_type:

payment_type status_group	annually	monthly	never pay	on failure	other	per bucket	unknown
functional	8.49	16.99	35.27	7.53	1.89	18.88	10.94
functional needs repair	5.72	21.47	44.17	6.42	2.73	9.47	10.01
non functional	2.87	8.29	52.85	5.29	1.42	10.89	18.39

Table for quantity_group:

quantity_group status_group	dry	enough	insufficient	seasonal	unknown
functional	0.49	67.11	24.54	7.21	0.66
functional needs repair	0.86	55.59	33.59	9.64	0.32
non functional	26.52	40.04	25.25	5.74	2.46

Table for waterpoint_type:

waterpoint_type	cattle trough	communal standpipe	communal standpipe multiple	dam	hand pump	improved spring	other
status_group							
functional	0.26	54.95	6.93	0.02	33.49	1.75	2.60
functional needs repair	0.05	52.35	15.01	0.00	23.84	1.97	6.79
non functional	0.13	37.40	14.11	0.00	24.77	0.60	22.99

6. Recommendations

1. Considering that most of the functional pumps have monthly payment plans or a per bucket, the Tanzanian government can consider modifying the existing payment plans of those pumps where the payments are different from those payment types, so that the chance of the pump being functional can be increased.

2. Considering that almost none of the functional pumps are dry, it is possible to verify which pumps are dry as a proxy variable to know if they are functional or not and thus focus efforts on repairing them.

3. Considering that non-functional pumps have in most cases a waterpoint_type different from cattle trough, communal standpipe, communal standpipe multiple, dam, hand pump and improved spring, it is possible to verify which pumps do not have these waterpoint_types as a proxy variable to know if they are functional or not and thus focus efforts on repairing them.