Project Report

Siddharth Ayathu(IMT2022517)

Sreyas Janamanchi(IMT2022554)

Pradyumna G(IMT2022555)

CODE LINK

Data Processing:

ONE-HOT ENCODING

```
[ ] le = LabelEncoder()

    # Fit and transform the specified columns
    for col in ['Education', 'EmploymentType', 'MaritalStatus', 'LoanPurpose', 'HasMortgage', 'HasDependents', 'HasCoSigner']:
        df[col] = le.fit_transform(df[col])
```

```
[ ] features=df.columns.to_numpy()
    features=features[(features!="LoanID") & (features!="Default")]
    features  # Removes values at indices 1 and 3
```

We apply label encoding to convert string values in the data to integers. Afterward, we remove LoanID and Default from the feature list, as they do not contribute to predicting the target values.

CHECKING FOR NULL VALUES

```
null_value_percentages=(df.isna().sum()/df.shape[0])*100
null_value_percentages
```

	0
LoanID	0.0
Age	0.0
Income	0.0
LoanAmount	0.0
CreditScore	0.0
MonthsEmployed	0.0
NumCreditLines	0.0
InterestRate	0.0
LoanTerm	0.0
DTIRatio	0.0
Education	0.0
EmploymentType	0.0
MaritalStatus	0.0
HasMortgage	0.0
HasDependents	0.0
LoanPurpose	0.0
HasCoSigner	0.0
Default	0.0

dtype: float64

On checking for null values, we find there are no null values present. Therefore, no values are dropped.

IMPUTING OUTLIERS (NONE FOUND)

```
class OutlierRemoval:
    def __init__(self, col):
        q1 = col.quantile(0.25)
        q3 = col.quantile(0.75)
        inter_quartile_range = q3 - q1
        self.upper_whisker = q3 + inter_quartile_range * 1.5
        self.lower_whisker = q1 - inter_quartile_range * 1.5
        self.has_outliers = False  # Track if outliers exist

    def replace_outlier(self, value):
        # Check if the value is an outlier and set `has_outliers` to True if it is
        if value < self.lower_whisker or value > self.upper_whisker:
            self.has_outliers = True
            return min(max(value, self.lower_whisker), self.upper_whisker)
        return value

def replace_outliers(df, columns):
    outlier_columns = []  # To store column names with outliers
    for col in columns:
        outlier_remover = OutlierRemoval(df[col])
        df[col] = df[col].apply(outlier_remover.replace_outlier)
        if outlier_remover.has_outliers:  # If outliers were replaced
            outlier_columns.append(col)
    print("Columns with outliers:", outlier_columns)  # Print the columns with outliers
    return df

# Replace outliers in the DataFrame `df` for selected columns
df = replace_outliers(df, features)
df
```

```
Columns with outliers: []
```

On checking for outliers, we find there are none present.

STANDARDIZATION

```python
scaler=StandardScaler()

normalized_features=scaler.fit_transform(df[features])
normalized_df = pd.DataFrame(normalized_features, columns=features)
normalized_df
```

	Age	Income	LoanAmount	CreditScore	MonthsEmployed	NumCreditLines	InterestRate	LoanTerm	DTIRatio	Education	EmploymentType	MaritalStatus	HasMortgage	HasDependents	LoanPurpose	HasCoSigner
0	-1.699838	1.413785	1.151487	1.711544	-0.967182	-0.449530	-0.454811	1.415720	1.339989	-0.442742	0.448348	1.226590	0.999936	-1.000406	-0.707810	-1.000563
1	0.234120	-0.649831	-1.715866	1.094714	-0.851727	-0.449530	0.939092	-0.000645	0.993538	-0.442742	1.343560	-1.224023	-1.000064	0.999594	-0.000312	-1.000563
2	-1.166333	0.046770	-0.458437	-0.762072	-1.515594	-0.449530	1.621727	-1.417010	-0.219039	0.450557	0.448348	0.001284	-1.000064	-1.000406	1.414685	0.999437
3	0.634249	-0.839783	1.440049	-0.258537	1.370784	0.445809	0.143437	1.415720	-1.431615	-1.336042	0.448348	1.226590	0.999936	-1.000406	-1.415308	0.999437
4	0.367496	0.845753	-1.488613	1.673779	-1.717640	1.341148	1.656386	-1.417010	-1.691453	-1.336042	-0.446863	1.226590	-1.000064	0.999594	-0.000312	0.999437
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
204272	-0.232698	0.875867	0.481625	0.484177	0.562598	-0.449530	1.499667	-1.417010	1.599826	-1.336042	-0.446863	-1.224023	-1.000064	-1.000406	0.707187	0.999437
204273	1.567884	-0.501854	0.874343	-0.718013	0.504870	0.445809	-0.632628	-0.000645	-1.691453	-1.336042	0.448348	1.226590	-1.000064	-1.000406	-0.707810	0.999437
204274	1.234443	-1.235732	-0.958332	-0.315185	0.995555	0.445809	-0.567830	1.415720	-1.128471	1.343857	-1.342074	1.226590	0.999936	-1.000406	-1.415308	-1.000563
204275	0.034055	1.636778	1.000728	-0.535481	-1.515594	1.341148	-1.383075	0.707538	-0.868633	-0.442742	0.448348	0.001284	0.999936	-1.000406	0.707187	-1.000563
204276	-0.499451	0.393761	0.861557	-0.201889	-1.659913	0.445809	1.454459	-0.708827	1.426601	-1.336042	-0.446863	-1.224023	-1.000064	-1.000406	0.707187	-1.000563

204277 rows × 16 columns

StandardScaler is used to normalize the feature values, transforming them to have a mean of 0 and a standard deviation of 1. This makes sure that all features contribute equally in model training.

OVERSAMPLING AND UNDERSAMPLING DATA BECAUSE THE GIVEN DATA IS IMBALANCED (CAN CAUSE MODEL TO BIAS TOWARDS LABEL=0)

```python
ros = RandomOverSampler(random_state=2)
X_oversampled, Y_oversampled = ros.fit_resample(normalized_df, df["Default"])

rus = RandomUnderSampler(random_state=2)
X_undersampled, Y_undersampled = rus.fit_resample(normalized_df, df["Default"])

class_counts = Y_oversampled.value_counts()
print("After oversampling: ",class_counts)

class_counts = Y_undersampled.value_counts()
print("After undersampling: ",class_counts)

ros = RandomOverSampler(random_state=2)
X_oversampled_unnormalized, Y_oversampled_unnormalized = ros.fit_resample(df[features], df["Default"])

rus = RandomUnderSampler(random_state=2)
X_undersampled_unnormalized, Y_undersampled_unnormalized = ros.fit_resample(df[features], df["Default"])
```

```
After oversampling:  Default
0    180524
1    180524
Name: count, dtype: int64
After undersampling:  Default
0    23753
1    23753
Name: count, dtype: int64
```

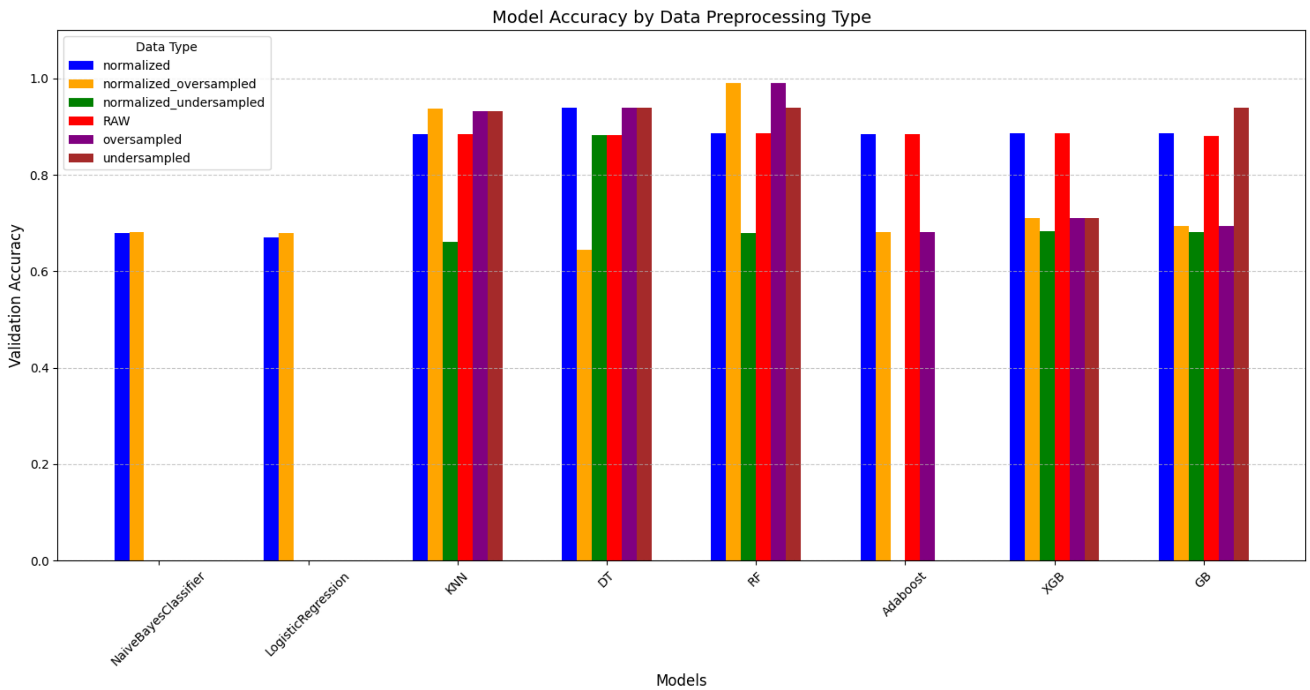
Since the target variable Default has many more 0s than 1s, we use oversampling and undersampling to balance the data. By adding more instances of 1 or reducing instances of 0, we create a more balanced dataset, which helps the model perform better on both classes.

PERFORMING TEST-TRAIN SPLIT

```python
X_train_oversampled, X_test_oversampled, y_train_oversampled, y_test_oversampled = train_test_split(X_oversampled, Y_oversampled, test_size=0.2, random_state=2)
X_train_undersampled, X_test_undersampled, y_train_undersampled, y_test_undersampled = train_test_split(X_undersampled, Y_undersampled, test_size=0.2, random_state=2)
X_train, X_test, y_train, y_test = train_test_split(df[features], df['Default'], test_size=0.2, random_state=2)
```

Performing test-train split. 80 percent of data is used for training the model, the other 20 percent is used for testing.

Models' Discussion:



Naive Bayes Classifier:

```
os.makedirs("NaiveBayesClassifier", exist_ok=True)
train(GaussianNB(),"NaiveBayesClassifier","oversampled",X_train_oversampled,y_train_oversampled,X_test_oversampled,y_test_oversampled,test_normalized_df,test_df["LoanID"])   #TRAINED WITH NORMALIZED AND OVERSAMPLED DATA
train(GaussianNB(),"NaiveBayesClassifier","undersampled",X_train_undersampled,y_train_undersampled,X_test_undersampled,y_test_undersampled,test_normalized_df,test_df["LoanID"])    #TRAINED WITH NORMALIZED AND UNDERSAMPLED DATA

NaiveBayesClassifier with oversampled: 0.6787841019249411
Results saved in NaiveBayesClassifier/NaiveBayesClassifier_oversampled.csv
NaiveBayesClassifier with undersampled: 0.6820669332772048
Results saved in NaiveBayesClassifier/NaiveBayesClassifier_undersampled.csv
0.6820669332772048
```

There's no hyperparameters used for naive bayes classification.

Naive Bayes assumes that all features are independent which may not be true. It also underperforms when there are complex boundaries present. Hence, naive bayes classifier isn't ideal for these kinds of classification problems, as reflected in the low accuracies seen.

We train the model on types of data:

- Oversampled data

- Undersampled data

Logistic Regression Model:

TRAINING LOGISTIC REGRESSION MODELS

```
[16] os.makedirs("LogisticRegression", exist_ok=True)

     train(LogisticRegression(random_state=2, max_iter=500),"LogisticRegression","oversampled",X_train_oversampled,y_train_oversampled,X_test_oversampled,y_test_oversampled,test_normalized_df,test_df["LoanID"
     train(LogisticRegression(random_state=2, max_iter=500),"LogisticRegression","undersampled",X_train_undersampled,y_train_undersampled,X_test_undersampled,y_test_undersampled,test_normalized_df,test_df["Lc

LogisticRegression with oversampled: 0.6793934358122143
Results saved in LogisticRegression/LogisticRegression_oversampled.csv
LogisticRegression with undersampled: 0.6810145232582614
Results saved in LogisticRegression/LogisticRegression_undersampled.csv
0.6810145232582614
```

We set the iterations to 500, this makes sure the model has enough iterations to converge and enough iterations to finish training. We ensure reproducibility by setting random_state= 2.

We train the model on 2 types of data:

- Oversampled data
- Undersampled data

Both the datasets give almost the same accuracy of around 68 percent. The reason logistic regression gives such low accuracy is due to:

- Limited flexibility: Logistic regression assumes a linear relationship between the features and the target variable, which may not capture complex relationships in the data.
- High dimensionality issues: Logistic regression can struggle with large numbers of features. Other complex models which can handle high dimensionality in the data are preferred over logistic regression as logistic regression underperforms on high dimensional data.
- Trade-off: The inability of the model to capture complex relations can be overcome by implementing polynomial scaling but this increases the dimensionality which makes it harder for logistic regression to work with. Therefore, this approach wasn't considered.

KNN:

```python
os.makedirs("KNN", exist_ok=True)

param_grid = {
    'n_neighbors': list(range(1, 5)) + list(range(28, 32))
}

# Initialize the KNeighborsClassifier model
knn = KNeighborsClassifier()

# Initialize GridSearchCV with the model and parameter grid
grid_search = GridSearchCV(
    estimator=knn,
    param_grid=param_grid,
    cv=5,                   # 5-fold cross-validation
    scoring='accuracy',     # Optimizing for accuracy
    n_jobs=-1               # Use all available cores
)

train_grid(grid_search,"KNN","normalized",normalized_df,df['Default'],test_normalized_df)   #TRAINED WITH NORMALIZED DATA
train_grid(grid_search,"KNN","normalized_oversampled",X_oversampled,Y_oversampled,test_normalized_df)   #TRAINED WITH NORMALIZED AND OVERSAMPLED DATA
train_grid(grid_search,"KNN","normalized_undersampled",X_train_undersampled,y_train_undersampled,test_normalized_df)    #TRAINED WITH NORMALIZED AND UNDERSAMPLED DATA
train_grid(grid_search,"KNN","",df[features],df['Default'],test_df[features])   #TRAINED WITHOUT PREPROCESSING DATA
train_grid(grid_search,"KNN","oversampled",X_oversampled_unnormalized,Y_oversampled_unnormalized,test_df[features])   #TRAINED WITH OVERSAMPLED DATA
train_grid(grid_search,"KNN","undersampled",X_undersampled_unnormalized,Y_undersampled_unnormalized,test_normalized_df)   #TRAINED WITH UNDERSAMPLED DATA
```

```
Started training
Best parameters: {'n_neighbors': 29}
Best cross-validation accuracy: 0.8839125296071966
Started training
Best parameters: {'n_neighbors': 1}
Best cross-validation accuracy: 0.9380137819063583
Started training
Best parameters: {'n_neighbors': 31}
Best cross-validation accuracy: 0.6607198568055449
Started training
Best parameters: {'n_neighbors': 30}
Best cross-validation accuracy: 0.8838146217752001
Started training
Best parameters: {'n_neighbors': 1}
Best cross-validation accuracy: 0.9321475197664052
Started training
Best parameters: {'n_neighbors': 1}
Best cross-validation accuracy: 0.9321475197664052
0.9321475197664052
```

KNN does not rely much on the features of the data. KNN can adapt to complex relationships because it relies purely on the proximity of similar cases, rather than fitting a predetermined function.

The only hyperparameter(important) used here is n_neighbours. This value is chosen between the values of 1 to 5 and 28 to 32. 1 to 5 explores smaller neighbourhood sizes, whereas 28 to 32 explores larger neighbourhood size.

We train the model on 6 types of data:

- Normalized Data
- Normalized and Oversampled Data
- Normalized and Undersampled Data
- Raw Data
- Oversampled Data
- Undersampled Data

Comparing the different models:-

- Oversampled and normalized data gave the best performance because they addressed class imbalance while retaining all data points.

- Undersampled data performed worse due to loss of information.
- Oversampling had a significant impact on performance.

Decision Tree:

```
os.makedirs("DecisionTree", exist_ok=True)
dt_model = DecisionTreeClassifier(random_state=2)
param_grid_dt = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 5],
    'max_features': [None, 'sqrt']
}

grid_search_dt = GridSearchCV(estimator=dt_model, param_grid=param_grid_dt,
                              cv=5, scoring='accuracy', n_jobs=-1)

train_grid(grid_search_dt,"DecisionTree","normalized_oversampled",X_oversampled,Y_oversampled,test_normalized_df)   #TRAINED WITH NORMALIZED AND OVERSAMPLED DATA
train_grid(grid_search_dt,"DecisionTree","normalized_undersampled",X_train_undersampled,y_train_undersampled,test_normalized_df)    #TRAINED WITH NORMALIZED AND UNDERSAMPLED DA
train_grid(grid_search_dt,"DecisionTree","",df[features],df['Default'],test_df[features])   #TRAINED WITHOUT PREPROCESSING DATA
train_grid(grid_search_dt,"DecisionTree","normalized",normalized_df,df['Default'],test_normalized_df)   #TRAINED WITH NORMALIZED DATA
train_grid(grid_search_dt,"DecisionTree","oversampled",X_oversampled_unnormalized,Y_oversampled_unnormalized,test_df[features])   #TRAINED WITH OVERSAMPLED DATA
train_grid(grid_search_dt,"DecisionTree","undersampled",X_undersampled_unnormalized,Y_undersampled_unnormalized,test_normalized_df)   #TRAINED WITH UNDERSAMPLED DATA
```

```
Started training
Best parameters: {'criterion': 'gini', 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Best cross-validation accuracy: 0.9390108795904725
Started training
Best parameters: {'criterion': 'entropy', 'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Best cross-validation accuracy: 0.6440901197210893
Started training
Best parameters: {'criterion': 'entropy', 'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Best cross-validation accuracy: 0.8822236496100089
Started training
Best parameters: {'criterion': 'entropy', 'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Best cross-validation accuracy: 0.8822187541285442
Started training
Best parameters: {'criterion': 'gini', 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Best cross-validation accuracy: 0.938958255031711
Started training
Best parameters: {'criterion': 'gini', 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Best cross-validation accuracy: 0.938958255031711
0.938958255031711
```

GRIDSEARCH ON RANDOMFOREST

Decision trees are capable of capturing non-linear patterns in the data. It can be beneficial in capturing the complex relationships in the data. They can also handle multi-dimensional data.

We train the model on 6 types of data:

- Normalized and Oversampled Data
- Normalized and Undersampled Data
- Raw Data
- Normalized Data Only
- Oversampled Data Only
- Undersampled Data Only

If the tree grows too deep, decision trees usually tend to overfit the data. Also, small changes in data can lead to very different tree structures which make them unstable

.

Hyperparameter tuning:

- We use Gridsearch to find the best hyperparameters.
- Max_depth: We take max_depth as 10,20, none. When none is taken, we let the tree completely grow. We chose 10, 20 as tree size to prevent overfitting.
- Min_values_split: Min_values_split is taken as 2,5 by increasing this value, we can make the tree less prone to overfitting.
- Min_samples: Min_samples is set as 1,5. Higher the value, less chance of noisy data in the branches.
- Max_features: Having sqrt as a feature prevents the model from relying too heavily on certain features, reducing overfitting.

Comparing the different models:-

- Oversampled and normalized data gave the best performance because they addressed class imbalance while retaining all data points.
- Undersampled data performed worse due to loss of information.
- Oversampling had a significant impact on the performance of decision tree.

Random Forest:

GRIDSEARCH ON RANDOMFOREST

```python
os.makedirs("RandomForest", exist_ok=True)

rf_model = RandomForestClassifier(random_state=2)

param_grid = {
    'n_estimators': [100],
    'max_depth': [10, None],
    'min_samples_split': [2],
    'min_samples_leaf': [1, 4],
    'max_features': ['sqrt']
}

# Initialize GridSearchCV with the model, parameter grid, and evaluation method
grid_search = GridSearchCV(
    estimator=rf_model,
    param_grid=param_grid,
    cv=5,                # 5-fold cross-validation
    scoring='accuracy', # Evaluation metric
    n_jobs=-1            # Use all available cores
)

train_grid(grid_search,"RandomForest","normalized",normalized_df,df['Default'],test_normalized_df)    #TRAINED WITH NORMALIZED DATA
train_grid(grid_search,"RandomForest","normalized_oversampled",X_oversampled,Y_oversampled,test_normalized_df)    #TRAINED WITH NORMALIZED AND OVERSAMPLED DATA
train_grid(grid_search,"RandomForest","normalized_undersampled",X_train_undersampled,y_train_undersampled,test_normalized_df)   #TRAINED WITH NORMALIZED AND UNDERSAMPLED DATA
train_grid(grid_search,"RandomForest","",df[features],df['Default'],test_df[features])    #TRAINED WITHOUT PREPROCESSING DATA
train_grid(grid_search,"RandomForest","oversampled",X_oversampled_unnormalized,Y_oversampled_unnormalized,test_df[features])    #TRAINED WITH OVERSAMPLED DATA
train_grid(grid_search_dt,"RandomForest","undersampled",X_undersampled_unnormalized,Y_undersampled_unnormalized,test_normalized_df)   #TRAINED WITH UNDERSAMPLED DATA
```

[15] ✓ 18m 27.0s

```
Started training
Best parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 100}
Best cross-validation accuracy: 0.8858168039590986
Started training
Best parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Best cross-validation accuracy: 0.990056725178462
Started training
Best parameters: {'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 100}
Best cross-validation accuracy: 0.6802967961279333
Started training
Best parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 100}
Best cross-validation accuracy: 0.8857874317892296
Started training
Best parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Best cross-validation accuracy: 0.9900788826209362
Started training
Best parameters: {'criterion': 'gini', 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Best cross-validation accuracy: 0.938958255031711
```

```
0.938958255031711
```

We conducted hyperparameter tuning for the RandomForestClassifier using GridSearchCV with the following parameters: n_estimators set to 100, max_depth options set to 10 and None, min_samples_split set to 2, min_samples_leaf values set to 1 and 4, and max_features set to "sqrt". A 5-fold cross-validation approach was used.

We train models on different datasets:

- Normalized Data
- Normalized and Oversampled Data
- Normalized and Undersampled Data
- Raw Data
- Oversampled Data
- Undersampled Data

Comparing the different models:-

- Oversampling is useful for addressing class imbalance.
- Both normalized and raw oversampled data resulted in high accuracy because the model had access to balanced decision boundaries.
- Undersampling is harmful, but preprocessing matters. When the dataset is undersampled and normalized, it performed poorly due to limited data and potential artifacts introduced during preprocessing.
- With raw undersampled data, the model performed better because the raw feature distribution might have provided more meaningful splits for the trees.
- Random Forests are robust to scaling.
- Normalization did not significantly impact best performance, demonstrating that Random Forests works with raw and scaled data effectively.
- Class imbalance is the primary factor affecting accuracy.

Though random forest has the highest validation accuracy it doesn't have the highest test accuracy. Random Forest uses randomness in selecting features and data samples for building trees. While this helps in ensemble diversity, it might also lead to inconsistent performance across different datasets, especially smaller test sets.

Gradient Boosting:

GRIDSEARCH ON GRADIENT BOOSTING

```python
os.makedirs("GradientBoosting", exist_ok=True)

gb_model = GradientBoostingClassifier()

param_grid = {
    'n_estimators': [ 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3],
    'min_samples_split': [2, 5],
}

grid_search = GridSearchCV(
    estimator=gb_model,
    param_grid=param_grid,
    cv=5,  # 5-fold cross-validation
    scoring='accuracy',
    n_jobs=-1  # Use all available cores
)

train_grid(grid_search,"GradientBoosting","normalized",normalized_df,df['Default'],test_normalized_df)    #TRAINED WITH NORMALIZED DATA
train_grid(grid_search,"GradientBoosting","normalized_oversampled",X_oversampled,Y_oversampled,test_normalized_df)    #TRAINED WITH NORMALIZED AND OVERSAMPLED DATA
train_grid(grid_search,"GradientBoosting","normalized_undersampled",X_train_undersampled,y_train_undersampled,test_normalized_df)   #TRAINED WITH NORMALIZED AND UNDERSAMPLED DATA
train_grid(grid_search,"GradientBoosting","",df[features],df['Default'],test_df[features])    #TRAINED WITHOUT PREPROCESSING DATA
train_grid(grid_search,"GradientBoosting","oversampled",X_oversampled_unnormalized,Y_oversampled_unnormalized,test_df[features])    #TRAINED WITH OVERSAMPLED DATA
train_grid(grid_search_dt,"GradientBoosting","undersampled",X_undersampled_unnormalized,Y_undersampled_unnormalized,test_normalized_df)   #TRAINED WITH UNDERSAMPLED DATA
```

✓ 49m 13.7s

```
Started training
Best parameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 2, 'n_estimators': 200}
Best cross-validation accuracy: 0.8861447894748862
Started training
Best parameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 2, 'n_estimators': 200}
Best cross-validation accuracy: 0.6940766855180487
Started training
Best parameters: {'learning_rate': 0.05, 'max_depth': 3, 'min_samples_split': 2, 'n_estimators': 200}
Best cross-validation accuracy: 0.681638783678048
Started training
Best parameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 2, 'n_estimators': 200}
Best cross-validation accuracy: 0.8861447894748862
Started training
Best parameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 2, 'n_estimators': 200}
Best cross-validation accuracy: 0.6940766855180487
Started training
Best parameters: {'criterion': 'gini', 'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2}
Best cross-validation accuracy: 0.938958255031711

0.938958255031711
```

Gradient boosting is robust to class imbalance. Gradient boosting uses decision trees as weak learners which can capture complex patterns effectively. Also, gradient boosting improves the model by focusing on misclassified instances. This is helpful at places where it is harder to predict if the person will default from their loan payment.

Hyperparameter tuning:

- N_estimators (number of weak learners): We set this value to 200 to give a balance between the training time and giving sufficient complexity to the model.
- Learning_rate: We set the value to 0.05, 0.1. These are the commonly used values for learning rate in gradient boosting.
- Max_depth: This value is set to 3. A max_depth of 3 restricts the complexity of each tree, which will reduce overfitting while allowing enough depth to capture significant patterns.

- Min_samples: This value is set to 2, 5. We take the value as 2 to make sure we get complex enough data and not simpler ones. We take 5 as min_samples to make sure we don’t overfit the data.

We train the models on different type of datasets:

- Normalized Data
- Normalized and Oversampled Data
- Normalized and Undersampled Data
- Raw Data
- Oversampled Data
- Undersampled Data

Comparing the different models:-

- Normalization helps gradient boosting handle scaled features better but doesn't guarantee improved accuracy if preprocessing distorts the data distribution.
- Oversampling amplifies noise and redundancy, causing a significant performance drop.

Adaboost

GRIDSEARCH ON ADABOOST

```python
os.makedirs("Adaboost", exist_ok=True)
adaboost_model = AdaBoostClassifier(
    estimator=DecisionTreeClassifier(random_state=2),
    random_state=2,
    algorithm='SAMME'
)

param_grid = {
    'estimator__max_depth': [3],
    'n_estimators': [200],
    'learning_rate': [0.01, 0.1, 1]
}

grid_search = GridSearchCV(
    estimator=adaboost_model,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

train_grid(grid_search,"Adaboost","normalized",normalized_df,df['Default'],test_normalized_df)    #TRAINED WITH NORMALIZED DATA
train_grid(grid_search, "Adaboost", "normalized_oversampled", X_oversampled, Y_oversampled, test_normalized_df)    #TRAINED WITH NORMALIZED AND OVERSAMPLED DATA
train_grid(grid_search,"Adaboost","",df[features],df['Default'],test_df[features])    #TRAINED WITHOUT PREPROCESSING DATA
train_grid(grid_search, "Adaboost", "oversampled", X_oversampled_unnormalized, Y_oversampled_unnormalized, test_df[features])   #TRAINED WITH OVERSAMPLED DATA
```

```
Started training
Best parameters: {'estimator__max_depth': 3, 'learning_rate': 0.1, 'n_estimators': 200}
Best cross-validation accuracy: 0.885459447112181
Started training
Best parameters: {'estimator__max_depth': 3, 'learning_rate': 0.1, 'n_estimators': 200}
Best cross-validation accuracy: 0.6880747121306572
Started training
Best parameters: {'estimator__max_depth': 3, 'learning_rate': 0.1, 'n_estimators': 200}
Best cross-validation accuracy: 0.885459447112181
Started training
Best parameters: {'estimator__max_depth': 3, 'learning_rate': 0.1, 'n_estimators': 200}
Best cross-validation accuracy: 0.6880747121306572
0.6880747121306572
```

We use Adaboost as it is well suited for binary classification tasks and performs well on imbalanced datasets.

We select the best hyperparameters using GridSearch.

We set estimator_max_depth as 3, to make sure we don't overfit nor underfit the data. We test values of 0.01, 0.1, 1 for learning rate values (commonly used values). We use 200 estimator value (number of weak learners), if we increase this value it takes more time to compute the prediction.

We train the model on 4 types of data:

- Normalized data
- Normalized and oversampled data
- Raw data
- Oversampled data

Comparing the different models:-

- AdaBoost models trained on normalized and raw data performed well, indicating AdaBoost's ability to handle imbalanced datasets effectively.
- Oversampling duplicates minority-class samples, amplifying noise and leading to:

- AdaBoost assigns higher weights to noisy misclassified samples, degrading performance.
- Increased risk of overfitting to the minority class.

XGBoost

```python
os.makedirs("XGboost", exist_ok=True)
xgb_model = XGBClassifier(random_state=2)

param_grid = {
    'n_estimators': [ 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3, 5],
    'subsample': [0.8],
    'colsample_bytree': [0.8]
}

# Initialize GridSearchCV with 5-fold cross-validation
grid_search = GridSearchCV(
    estimator=xgb_model,
    param_grid=param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

train_grid(grid_search,"XGboost","normalized",normalized_df,df['Default'],test_normalized_df)   #TRAINED WITH NORMALIZED DATA
train_grid(grid_search,"XGboost","normalized_oversampled",X_oversampled,Y_oversampled,test_normalized_df)   #TRAINED WITH NORMALIZED AND OVERSAMPLED DATA
train_grid(grid_search,"XGboost","normalized_undersampled",X_train_undersampled,y_train_undersampled,test_normalized_df)    #TRAINED WITH NORMALIZED AND UNDERSAMPLED DATA
train_grid(grid_search,"XGboost","",df[features],df['Default'],test_df[features])   #TRAINED WITHOUT PREPROCESSING DATA
train_grid(grid_search,"XGboost","oversampled",X_oversampled_unnormalized,Y_oversampled_unnormalized,test_df[features])   #TRAINED WITH OVERSAMPLED DATA
train_grid(grid_search,"XGboost","undersampled",X_undersampled_unnormalized,Y_undersampled_unnormalized,test_df[features])   #TRAINED WITH UNDERSAMPLED DATA
```

```
Started training
Best parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 200, 'subsample': 0.8}
Best cross-validation accuracy: 0.8862378011064983
Started training
Best parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200, 'subsample': 0.8}
Best cross-validation accuracy: 0.7113707851638533
Started training
Best parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 200, 'subsample': 0.8}
Best cross-validation accuracy: 0.6839017234574398
Started training
Best parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 200, 'subsample': 0.8}
Best cross-validation accuracy: 0.8862378011064983
Started training
Best parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200, 'subsample': 0.8}
Best cross-validation accuracy: 0.7113707851638533
Started training
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/process_executor.py:752: UserWarning: A worker stopped while some jobs were given to the executor. This can be caused by a too short worker t
  warnings.warn(
Best parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200, 'subsample': 0.8}
Best cross-validation accuracy: 0.7113707851638533
0.7113707851638533
```

XGBoost is suitable for datasets with a mix of numerical and categorical data, as it can handle these types effectively without requiring much data preprocessing. Additionally, XGBoost excels in addressing imbalanced datasets due to its ability to weigh classes. These attributes make XGBoost a strong candidate for the given dataset.

Explanation of Hyperparameter Tuning:-

- To optimize the XGBoost model's performance, we tuned the following key hyperparameters:
- **n_estimators**:
  - Specifies the number of trees in the model.
  - We fixed this at 200 to balance model performance with computational efficiency, avoiding additional values to reduce runtime.
- **learning_rate**:

- Controls the step size during gradient updates to minimize loss.
    - Smaller values typically enhance accuracy but require more trees. We tested these values (0.05,0.1) to find the optimal value.
- **max_depth**:
    - Defines the maximum depth of a tree, which determines the complexity of the model.
    - Larger values increase the risk of overfitting but capture finer details in the data. We evaluated depths of 3 and 5 to identify the most effective balance.
- **subsample**:
    - Denotes the fraction of training samples used for each tree.
    - A value of 0.8 was chosen to introduce randomness and reduce overfitting and controlling training time.
- **colsample_bytree**:
    - Specifies the fraction of features sampled for each tree.
    - To speed up training and mitigate overfitting, we used 0.8 as the fixed value.

We train multiple models of XGBoost with different sets of data:

- Normalized data
- Normalized and oversampled data
- Normalized and undersampled data
- Raw data
- Oversampled data
- Undersampled data

Comparing the different models:-

- **XGBoost with normalization (no oversampling or undersampling)** achieves the best performance among all models discussed.
- XGBoost effectively handles imbalanced data, making oversampling or undersampling unnecessary.
- XGBoost also performs well without any data preprocessing, reinforcing its capability to manage imbalanced datasets independently.