ML Project - Part 1

Mexican Tourist Profiles

Team Members

Pranav Laddhad - IMT2022074

Rishit Mane - IMT2022564

Himanshu Khatri - IMT2022584

Problem Description

In the competitive world of tourism, understanding traveler spending behavior is essential for enhancing visitor experiences and optimizing resource allocation. This competition challenges participants to predict the spending category—low, medium, or high—of tourists visiting Mexico based on various demographic and trip-related features.

The goal is to predict each visitor's spending category.

"category" : The range of expenditures a tourist spends in Mexico: 0-High; 1-Medium; 2-Low

Dataset

Data Collection

The dataset includes anonymized records of travelers to Mexico with various attributes that describe their demographics, trip plans, and purchasing preferences. Each row represents an individual trip record, and the columns include: **Trip Information**, **Traveler Demographics**, **Package Details**, **Target Variable**.

Data Attributes

Below is a list and description of the attributes in the dataset:

- **trip_ID**: Unique identifier for each tourist.
- **visitor_nation**: Country of origin of the visitor.
- **age_bracket**: Age group of the tourist.
- **travelling_with**: Relationship of the people a tourist is traveling with.
- **female_count**: Total number of females in the group.
- **male_count**: Total number of males in the group.
- **key_activity**: Main activity or experience the visitor intends to engage in while in Mexico.
- **trip_purpose**: Purpose of visiting Mexico.
- **first_time_visitor**: Indicates whether this is the visitor's first trip to Mexico (Yes/No).
- **mainland_nights**: Number of nights spent on the Mexican mainland.
- **island_nights**: Number of nights spent on islands off the Mexican coast.
- **tour_arrangement**: Arrangement type of visiting Mexico (self-arranged, package, etc.).

- **transport_package_international**: Indicates if the tour package includes international transportation service.
- **source_of_info**: Source from which the visitor learned about the trip or destination.
- **package_accommodation**: Indicates if the tour package includes accommodation.
- **food_package**: Indicates if the tour package includes food services.
- **transport_package_mx**: Indicates if the tour package includes transportation service within Mexico.
- **sightseeing_package**: Indicates if the tour package includes sightseeing services.
- **guided_tour_package**: Indicates if the tour package includes a tour guide.
- **insurance_package**: Indicates if the tour package includes insurance services.
- **days_before_booked**: Number of days in advance the trip was booked.
- **weather_at_arrival**: Weather condition at the time of arrival.
- **tour_length**: Length of the tour (number of days).
- **special_requirements**: Indicates if the tourist had any special requirements.
- **category**: Spending range of a tourist in Mexico; categorized as 0 for High, 1 for Medium, and 2 for Low.

Data Processing

Describe the preprocessing steps taken to clean and prepare the dataset for analysis. For example:

Handling Duplicates

First, we checked if there are any duplicate rows. There weren't any as the shape remained same thus there were no duplicate rows.

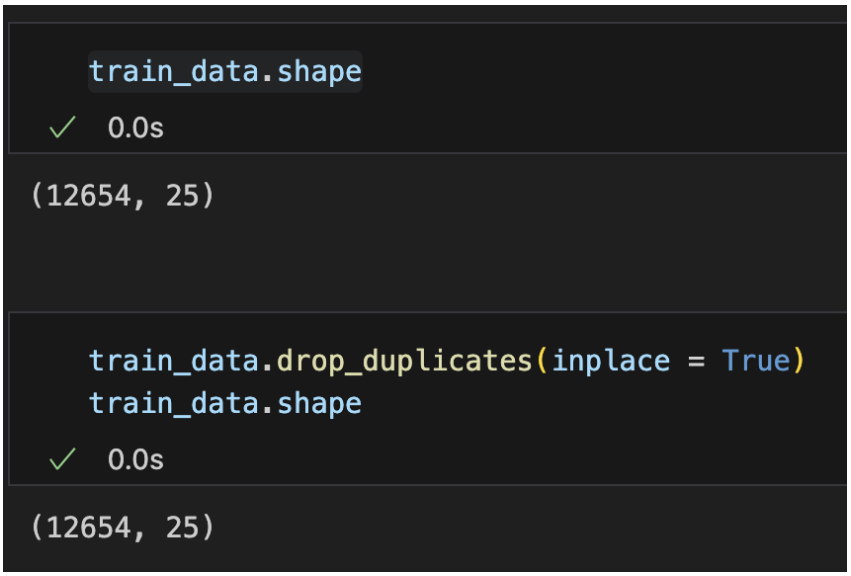

Figure 1: Checking for duplicates in the training data

Dropping columns

We dropped the following columns as they were not relevant to predicting tourist categories:
- *trip_ID*
- *source_of_info*
- *weather_at_arrival*
- *special_requirements*

We thought that the removing of these columns was necessary to avoid additional meaningless features that did not contribute significantly to the model's performance.

Handling Missing Values

We then checked for the null (missing) values in different columns of the training data

```
# Display the number of nulls in each column
null_counts = train_data.isnull().sum()
print(null_counts)
```

✓ 0.0s

```
visitor_nation                    230
age_bracket                         8
travelling_with                   737
female_count                        2
male_count                          4
key_activity                      128
trip_purpose                        0
first_time_visitor                 99
mainland_nights                     0
island_nights                       0
tour_arrangement                    0
transport_package_international   147
package_accomodation              143
food_package                      171
transport_package_mx                0
sightseeing_package                 0
guided_tour_package                 0
insurance_package                 236
days_before_booked               1553
tour_length                       402
category                           34
dtype: int64
```

Figure 2: Number of null values

```
# Display the percentage of nulls in each column
null_value_percentages = (train_data.isna().sum()/train_data.shape[0])*100
null_value_percentages
```

✓ 0.0s

```
visitor_nation                    1.817607
age_bracket                       0.063221
travelling_with                   5.824245
female_count                      0.015805
male_count                        0.031611
key_activity                      1.011538
trip_purpose                      0.000000
first_time_visitor                0.782361
mainland_nights                   0.000000
island_nights                     0.000000
tour_arrangement                  0.000000
transport_package_international   1.161688
package_accomodation              1.130077
food_package                      1.351351
transport_package_mx              0.000000
sightseeing_package               0.000000
guided_tour_package               0.000000
insurance_package                 1.865023
days_before_booked               12.272799
tour_length                       3.176861
category                          0.268690
dtype: float64
```

Figure 3: Percentage of null values

Handling by deletion:

The number and also percentage of missing values column wise can be seen here. Since all the columns had very less value for this % we decided that dropping of any column on this basis is not a good idea.

Handling by imputation : Next, we categorized the columns into two groups: *categorical columns* and *numerical columns*.
- Categorical columns were imputed using their respective mode values.
- Numerical columns were imputed using their respective median values.
This approach ensured a consistent treatment of missing data based on the type of attribute.

Encoding Categorical Variable

We tried to use the following methods for encoding categorical columns:

1. **Ordinal Encoding:**

   - We used the `OrdinalEncoder` for encoding the categorical columns in our dataset.

     For categories that contain new or unseen values (i.e., values that were not present during the model fitting process), we configured the encoder to handle these values by assigning them a special

integer value of `-1`. This ensures that any new or missing values are mapped to a consistent placeholder, rather than causing errors or disruptions in the encoding process. This method helps to handle null or unseen categories in a practical manner while preserving the integrity of the model.

2. **Label Encoding:**

   - Label encoding was used to convert non-ordinal categorical values into numeric values by assigning a unique integer to each category. This method is computationally efficient, but it assumes that the integers have a meaningful relationship, which might not always be true.

3. **One-Hot Encoding:**

   - One-hot encoding was applied to categorical columns to transform each category into a separate binary column (0 or 1). The goal of one-hot encoding is to represent categorical values in a way that captures the uniqueness of each category without implying any ordinal relationship.

Why One-Hot Encoding was Preferred:

- One-hot encoding was preferred for categorical columns because the binary columns it generates are independent of each other. Unlike label encoding, where numerical labels can unintentionally imply an ordinal relationship, one-hot encoding ensures that the machine learning model treats each category distinctly.

- Empirical results showed that one-hot encoding provided better performance, likely because it avoids introducing spurious relationships between categories, especially when the features lack inherent order.

Outlier Removal

- Using the Interquartile Range (IQR) method, the percentage of outliers was calculated. Outliers are identified as data points that fall significantly below the first quartile (Q1) or above the third quartile (Q3), where these quartiles represent the 25th and 75th percentiles of the data, respectively.

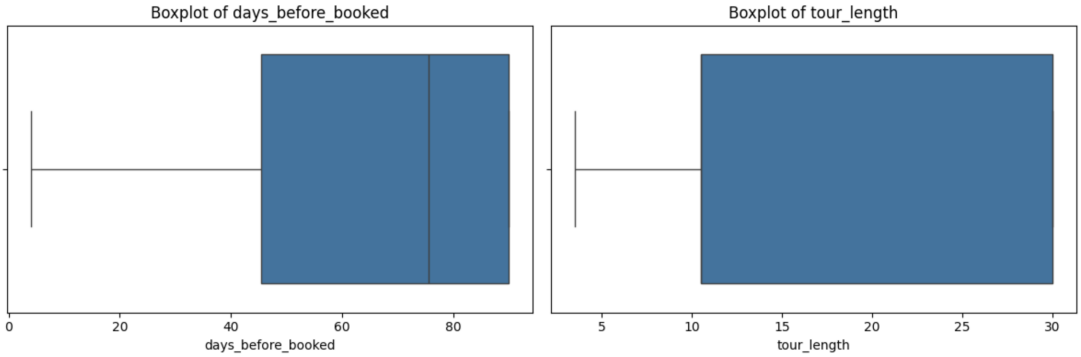

Figure 4: Number of null values

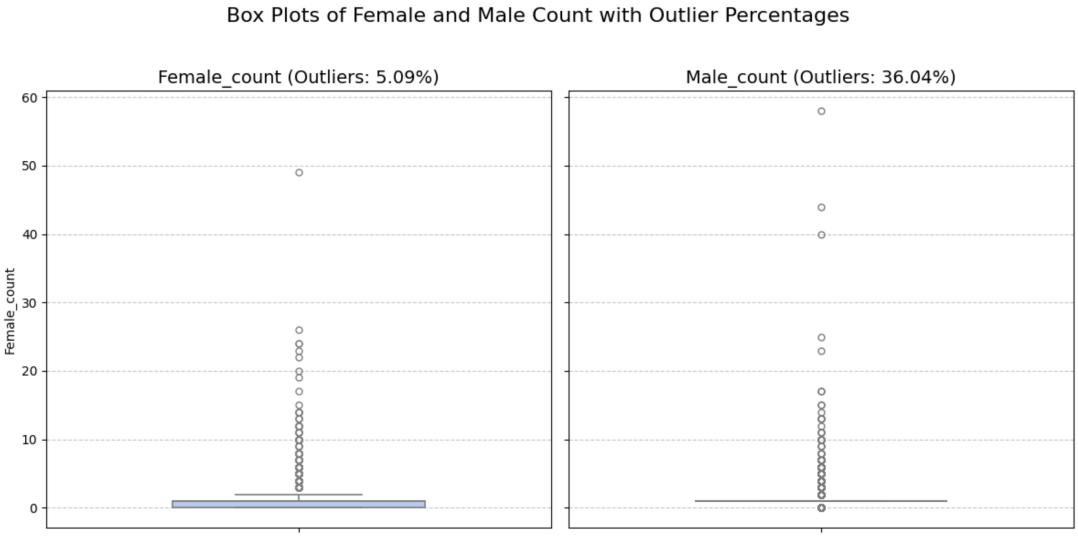

Figure 5: Number of null values

- The function `calculate_outlier_percentage` computes these boundaries for a specified column in a DataFrame, identifies the rows that fall outside these limits, and calculates the percentage of such rows relative to the total number of rows. This percentage is returned as a measure of how prevalent outliers are in that column. To analyze multiple columns, we iterate over a list of column names and apply the function to each column.

- From the above graphs, it is evident that there are no outliers for the features `days_before_booked` and `tour_length`. However, we can observe that there are outliers present for the features `female_count` and `male_count`.

- We did not remove the outliers from the data as removing them decreased our accuracy. The most probable reason could be that the removed outliers might be providing necessary variance in the data,

and their absence could lead to overfitting the model by oversimplifying the dataset.

Scaling and Normalization of Numerical Features

- We applied scaling to the numerical features using StandardScaler to ensure all features contribute equally to the model. StandardScaler standardizes features by removing the mean and scaling to unit variance, making each feature have a mean of 0 and standard deviation of 1.
- Why?

  To improve model training, especially for algorithms sensitive to feature scales (e.g., gradient descent-based models). To enhance distance calculations in distance-based models like KNN. To ensure fair variance contribution for PCA.

Principal Component Analysis

- An analysis of the correlation matrix reveals that the majority of correlations between features are close to zero, suggesting weak linear relationships. This observation implies that most features do not share substantial variance, indicating a lack of linear dependence among them. Principal Component Analysis (PCA) is typically more effective in situations where features exhibit strong correlations, as it can leverage these relationships to reduce redundancy by combining highly correlated features into principal components.
- Given the weak correlations observed here, PCA may not provide significant dimensionality reduction or improve model performance, as it would likely require a large number of components to capture the overall variance. Consequently, applying PCA might not yield substantial benefits for this dataset, especially if interpretability of individual features is desired.

Application of SMOTE for Handling Class Imbalance

- In many datasets, especially in classification tasks, there can be an imbalance between classes, where one class has significantly more samples than another. This imbalance can lead to biased model predic-

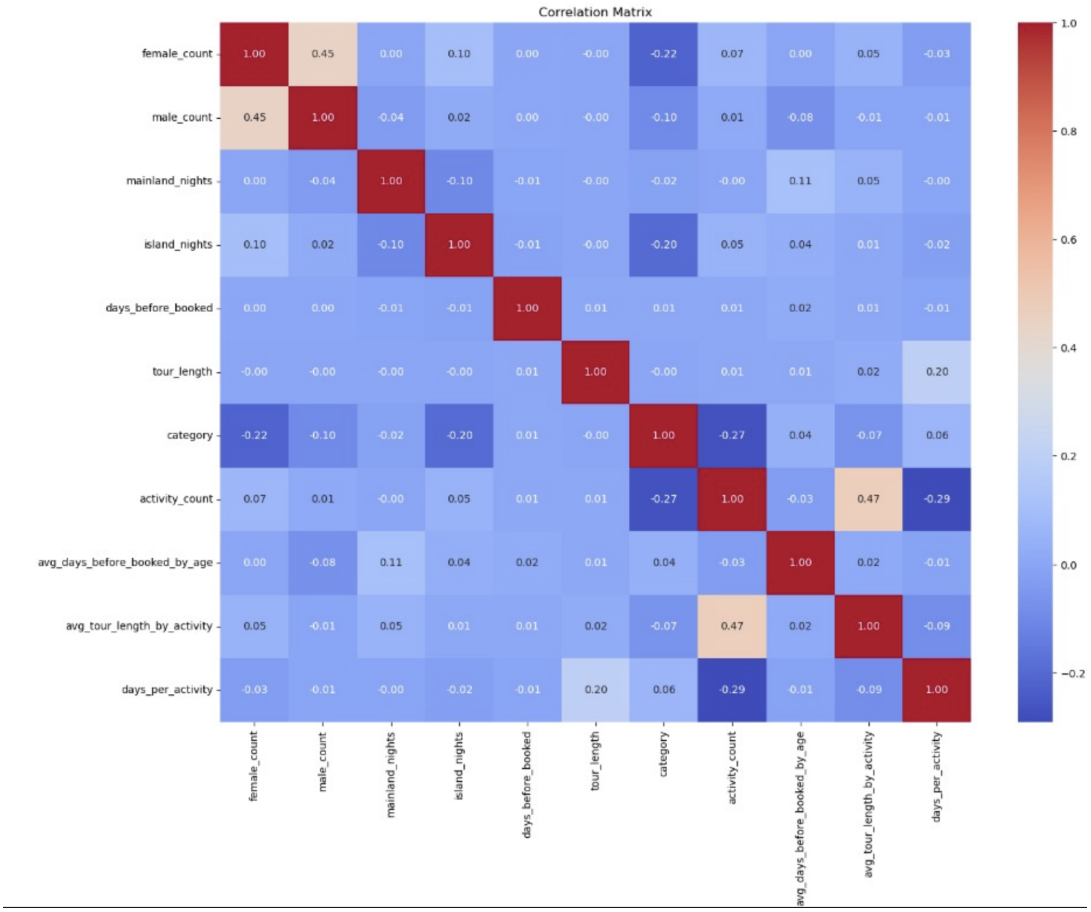

Figure 6: Correlation Matrix

tions, as the model may favor the majority class and underperform on the minority class.

- To address this issue, we apply Synthetic Minority Over-sampling Technique (SMOTE). SMOTE is an oversampling method that generates synthetic samples for the minority class by interpolating between existing minority class samples. By doing so, it balances the dataset, helping the model learn a more robust decision boundary that accurately represents both classes.

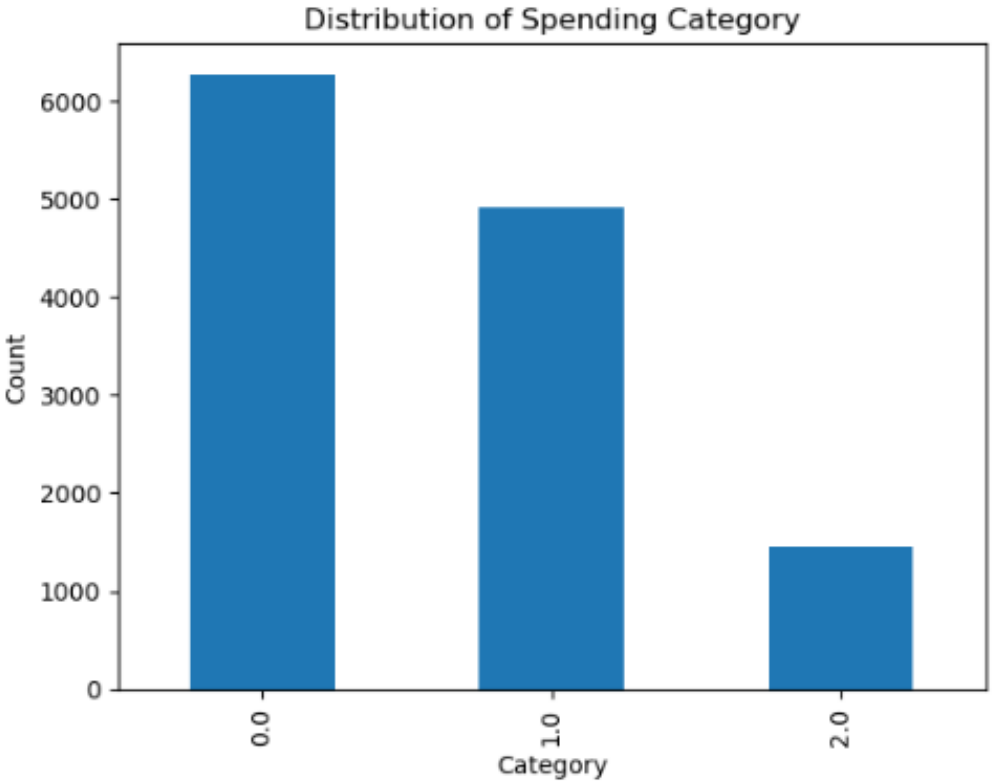

Figure 7: Category Count

Feature Engineering

Feature engineering is the process of creating new features (variables) or modifying existing ones in a dataset to improve the performance of a machine learning model. We have implemented feature engineering using the function `feature_engineering()`.

Using the preprocessing techniques mentioned above, we gained insights into theimportance of features.We then started dropping unnecessary columns.

What We Tried

- First, we combined the `female_count` and `male_count` columns to calculate the total number of companions a tourist is traveling with. This helps us understand group sizes and travel dynamics.
- Next, we calculated the `island_to_mainland_ratio` by dividing `island_nights` by `mainland_nights`, which indicates a tourist's preference for staying on islands compared to the mainland. To avoid division by zero,

```python
def feature_engineering(data):
    # data['total_companions'] = data['female_count'] + data['male_count']
    # data['island_to_mainland_ratio'] = data['island_nights'] / (data['mainland_nights'] + 1)  # Avoid division by zero
    # data['number_of_island_days'] = data['island_nights'] + data['mainland_nights']

    activity_count_map = data['key_activity'].value_counts().to_dict()
    data['activity_count'] = data['key_activity'].map(activity_count_map)

    # age_count_map = data['age_bracket'].value_counts().to_dict()
    # data['age_count'] = data['age_bracket'].map(age_count_map)

    # trip_count_map = data['trip_purpose'].value_counts().to_dict()
    # data['trip_count'] = data['trip_purpose'].map(trip_count_map)

    data['avg_days_before_booked_by_age'] = data.groupby('age_bracket')['days_before_booked'].transform('mean')

    # data['avg_tour_length_by_age'] = data.groupby('age_bracket')['tour_length'].transform('mean')
    data['avg_tour_length_by_activity'] = data.groupby('key_activity')['tour_length'].transform('mean')

    data['days_per_activity'] = data['tour_length'] / (data['activity_count'] + 1)  # Avoid divide by zero

    # data.drop(columns = ["female_count", "male_count"])
    # data.drop(columns = ["island_nights", "mainland_nights"])
    return data
```

Figure 8: Feature Engineering Function

we added a constant of 1 to `mainland_nights` in the denominator.

- We summed `island_nights` and `mainland_nights` to determine the total number of nights a tourist is spending in Mexico, capturing their overall duration of stay across different regions. By doing this, we are extracting meaningful patterns from the data to improve our predictive models.
- We calculated the frequency of each `age_bracket` and mapped it to create a new feature, `age_count`, which represents how common each age group is.
- Similarly, we mapped the frequency of each `trip_purpose` to create `trip_count`, helping us understand the popularity of different travel purposes.
- We removed the columns `female_count` and `male_count` since they are no longer needed after calculating the `total_companions` feature. Similarly, we dropped the `island_nights` and `mainland_nights` columns, as their information is now captured in the new features we created.

But these methods did not help us get the best results.

Combination for Best Results

- We created a mapping of the frequency of each `key_activity` using `value_counts()` and converted it into a dictionary.
- We mapped this frequency to a new feature, `activity_count`, which indicates how often each activity occurs in the dataset.
- We calculated the average number of days before booking a trip for each `age_bracket` and stored it in the feature `avg_days_before_booked_by_age`.
- Similarly, we calculated the average tour length for each `key_activity` and stored it in `avg_tour_length_by_activity`.
- We created a new feature, `days_per_activity`, by dividing `tour_length` by the `activity_count` (with an offset of 1 to avoid division by zero). This new feature represents the number of days spent per activity.
- Finally, we returned the modified dataset after these transformations.

This combination of feature engineering gave us the best outputs.

Model Building

The following models were used in the project:

Model 1: K-Nearest Neighbours (KNN) – A simple, instance-based learning algorithm that classifies new instances based on the majority class of the nearest data points.

Model 2: Decision Tree – A supervised learning algorithm that creates a model based on a series of binary decisions, which can be visualized as a tree structure.

Model 3: Random Forest – An ensemble learning method that builds multiple decision trees and merges them to improve predictive accuracy and control overfitting.

Model 4: Gradient Boost Classifier – A boosting algorithm that sequentially fits models to the residual errors of previous models, leading to strong predictive performance.

Model 5: XGBoost – A highly efficient and scalable implementation of gradient boosting, used for classification and regression tasks.

Model 6: AdaBoost – An ensemble technique that combines the predictions of several base learners to improve the model's performance, focusing on misclassified points in each iteration.

Model Evaluation

Here, we discuss the metrics used for evaluating the model performance, such as accuracy, F1 score, ROC-AUC, etc., but we have shown it for F1 scores and validation accuracy on test-validation data

Model	Validation Accuracy	F1 Score
XGB	0.7528	0.75
Random Forest	0.7512	0.7485
Gradient Boosting	0.7488	0.7458
Decision Tree	0.704	0.7035
Ada boost	0.6977	0.6957
KNN	0.6204	0.6204

Figure 9: Validation Accuracies and F1 Scores for different models

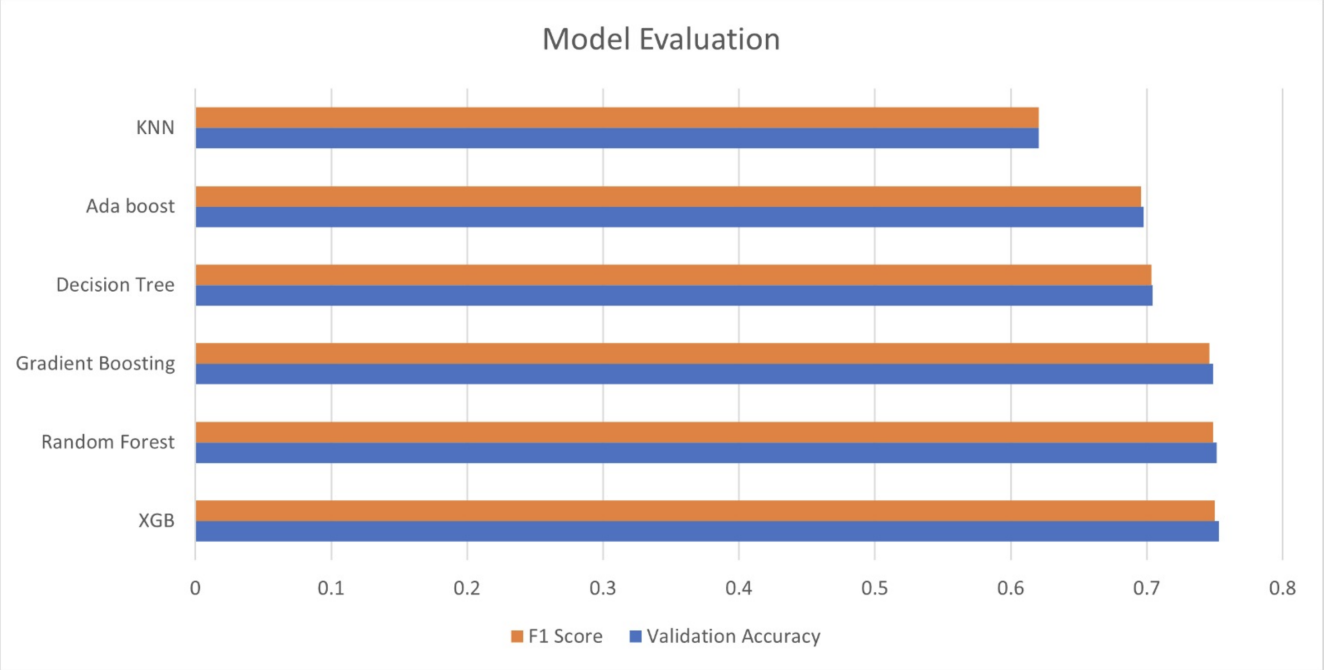

Figure 10: Using GridSearchCV to find best parameters for XGBClassifier

Hyperparameter Tuning

As can be seen, the best F1 validation scores were obtained for Random Forest and XGboost and hence using GridSearch CV we found the best parameters for both of these models :

```python
# Define the parameter grid
param_grid_xgb = {
    'model__n_estimators': [100, 150, 200],       # XGBoost parameters with the model step
    'model__max_depth': [4, 6, 8],
    'model__learning_rate': [0.01, 0.05, 0.1],
    'model__subsample': [0.75, 0.85],
    'model__colsample_bytree': [0.8, 0.9],
}

# Initialize the XGBClassifier
xgb1_model = XGBClassifier(
    random_state=42,
    enable_categorical=True,
    use_label_encoder=False,
    eval_metric='mlogloss'  # Avoid warnings
)

pipeline_xgb1 = ImbPipeline(steps=[
    ('preprocessor', preprocessor),
    ('smote', SMOTE(random_state=42, sampling_strategy='auto')),
    ('model', xgb1_model)
])


# GridSearchCV to find the best parameters
grid_search_xgb = GridSearchCV(
    estimator=pipeline_xgb1,
    param_grid=param_grid_xgb,
    scoring='accuracy',  # Metric for evaluation
    cv=5,                # 5-fold cross-validation
    verbose=2,           # Detailed logs
    n_jobs=-1            # Use all available processors

)

# Fit the GridSearchCV on the training data
grid_search_xgb.fit(X_train, y_train)

# Display the best parameters and best score
print(f"Best Parameters for XGBClassifier: {grid_search_xgb.best_params_}")
print(f"Best Cross-Validation Accuracy: {grid_search_xgb.best_score_:.4f}")

# Use the best model for evaluation
gs_xgb_model = grid_search_xgb.best_estimator_
```

Figure 11: Using GridSearchCV to find best parameters for XGBClassifier

```python
# Define the parameter grid for RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
param_grid = {
    'model__n_estimators': [100, 150],
    'model__max_depth': [4, 6, None],
    'model__min_samples_split': [2, 5],
    'model__min_samples_leaf': [1, 2],
    'model__max_features': ['sqrt', 'log2', None]
}

# Initialize the RandomForestClassifier
rf1_model = RandomForestClassifier(random_state=42)

# Create the pipeline with preprocessing, SMOTE, and the model
pipeline_rf = ImbPipeline(steps=[
    ('preprocessor', preprocessor),  # Replace 'preprocessor' with your actual preprocessing pipeline
    ('smote', SMOTE(random_state=42, sampling_strategy='auto')),
    ('model', rf1_model)
])

# GridSearchCV to find the best parameters
grid_search = GridSearchCV(
    estimator=pipeline_rf,
    param_grid=param_grid,
    scoring='accuracy',  # Change to 'f1_weighted', 'roc_auc', etc. as needed
    cv=5,  # 5-fold cross-validation
    verbose=2,
    n_jobs=-1  # Use all available processors
)

# Fit the GridSearchCV to the training data
grid_search.fit(X_train, y_train)

# Display the best parameters and best score
print()
print(f"Best Parameters for RandomForest classifier: {grid_search.best_params_}")
#print(f"Best Cross-Validation Accuracy: {grid_search.best_score_:.4f}")

# Use the best model for the pipeline
gs_rf_model = grid_search.best_estimator_
```

Figure 12: Using GridSearchCV to find best parameters for RandomForestClassifier

Below are the results:-

```
Best Parameters for XGBClassifier: {'model__colsample_bytree': 0.8, 'model__learning_rate': 0.05, 'model__max_depth': 6, 'model__n_estimators': 200, 'm
odel__subsample': 0.85}
Best Cross-Validation Accuracy: 0.7612
```

Figure 13: Best parameters for XGBClassifier

```
Best Parameters for RandomForest classifier:
{'model__max_depth': None, 'model__max_features': 'sqrt', 'model__min_samples_leaf': 1, 'model__min_samples_split': 5, 'model__n_estimators': 150}
```

Figure 14: Best parameters for RandomForestClassifier

These two figures show the `param_grid` and the part of the code used for hyperparameter tuning.

Test Results

The predictions were saved in the required submission format (e.g., as CSV files containing IDs and predicted labels) and were uploaded to Kaggle for scoring. The table above summarizes the Kaggle scores for each model

Model	Validation Accuracy	F1 Score	Kaggle Score
XGB	0.7528	0.75	0.71137
Random Forest	0.7512	0.7485	0.68869
Gradient Boosting	0.7488	0.7458	0.70238
Decision Tree	0.704	0.7035	0.67911
Ada boost	0.6977	0.6957	0.65697
KNN	0.6204	0.6204	0.58884

Figure 15: Kaggle scores for different models

Kaggle Submissions

Among these, the **XGBoost** model provided the best score for prediction of the target variable "category", the range of expenditures a tourist spends in Mexico: 0-High; 1-Medium; 2-Low for the given dataset.

We had a kaggle score (F1 score) equal to **0.71137** for this.

Project Part 2

1 Logistic Regression (LR)

Overview

Logistic Regression is a linear model commonly used for binary classification tasks. It predicts the probability of an instance belonging to a class using a logistic function. To handle imbalanced datasets and improve model performance, we used an integrated pipeline with oversampling (SMOTE) and hyperparameter tuning.

Implementation

The implementation of Logistic Regression included the following steps:

- **Pipeline Setup:** We constructed an imbalanced learning pipeline using `imblearn.pipeline.Pipeline`. This pipeline included:
  1. **Preprocessor:** Preprocessing of numerical and categorical data using scaling and one-hot encoding.
  2. **SMOTE:** Synthetic Minority Oversampling Technique (SMOTE) to address the class imbalance in the dataset.
  3. **Model:** Logistic Regression as the classification model.
- **Hyperparameter Tuning:** The parameters of the Logistic Regression model were optimized using `GridSearchCV`. The parameter grid included:
  - Regularization Strength (`C`): {0.01, 0.1, 1, 10}
  - Solver (`solver`): `liblinear`
  - Regularization Penalty (`penalty`): `l1`, `l2`
- **Evaluation Metrics:** The model was evaluated using accuracy, F1 score, and ROC-AUC score.

Best Hyperparameters and Performance

After performing 5-fold cross-validation, the best hyperparameters and the corresponding model performance on the validation set were as follows:

- **Best Hyperparameters:**

  - Regularization Strength (`C`): 0.1
  - Regularization Penalty (`penalty`): `l1`
  - Solver (`solver`): `liblinear`

- **Performance Metrics:**
  - Validation Accuracy: 0.7072
  - Validation F1 Score: 0.7082

Code Explanation

The steps of the implementation were executed as follows:

1. Define a parameter grid for Logistic Regression hyperparameters.
2. Initialize a Logistic Regression model with `random_state` set to 42 for reproducibility and `max_iter` set to 1000 to ensure convergence.
3. Create an imbalanced learning pipeline with:
   - A preprocessor for handling numerical and categorical features.
   - SMOTE for oversampling minority class instances.
   - The Logistic Regression model.
4. Use `GridSearchCV` to search for the optimal hyperparameters and train the pipeline.
5. Evaluate the model on the validation set to obtain accuracy and F1 score.

2 Support Vector Machine (SVM) Implementation

To implement a Support Vector Machine (SVM) model for classification, a pipeline was created combining data preprocessing, oversampling using SMOTE, and training the SVM classifier with optimal hyperparameters. The implementation details are as follows:

2.1 Data Preprocessing

The dataset contains both numerical and categorical features. To handle these appropriately, separate preprocessing steps were applied:

- **Numerical Features**: Missing values were imputed using the median value of each column, followed by feature scaling with *StandardScaler*.
- **Categorical Features**: Missing values were imputed with the most frequent value in each column, followed by one-hot encoding to convert categorical variables into numerical format.

These transformations were combined into a *ColumnTransformer*, ensuring that both types of features were processed simultaneously.

2.2 Oversampling with SMOTE

To address class imbalance in the dataset, **Synthetic Minority Oversampling Technique (SMOTE)** was applied. SMOTE generates synthetic samples for minority classes by interpolating between existing minority samples. This oversampling ensures a balanced class distribution, which helps the SVM model learn effectively.

2.3 Support Vector Machine Classifier

An SVM classifier was used as the final model in the pipeline. The hyperparameters were set as follows:

- **Kernel**: Radial Basis Function (RBF) kernel was chosen for its ability to handle non-linear decision boundaries.
- **Regularization Parameter (C)**: Set to 0.16, balancing the trade-off between achieving a low error on training data and maintaining a smooth decision boundary.
- **Gamma**: Set to `scale`, which automatically scales the gamma parameter based on the dataset.
- **Probability Estimation**: Enabled to allow for probabilistic predictions.

2.4 Pipeline Implementation

The pipeline was implemented using the `imblearn.pipeline.Pipeline` module, ensuring seamless integration of preprocessing, oversampling, and model training. The pipeline structure is as follows:

1. Preprocessor: Applies the defined `ColumnTransformer` to preprocess both numerical and categorical features.
2. SMOTE: Balances the dataset by generating synthetic samples for minority classes.
3. SVM Model: Trains the SVM classifier with the preprocessed and balanced dataset.

2.5 Model Evaluation

The trained SVM pipeline was evaluated on the validation set. The performance metrics obtained are:

- **Accuracy**: {`0.6933438985736925`}
- **F1-Score**: {`0.6925371591276853`}

These metrics highlight the effectiveness of the SVM pipeline in handling class imbalance and achieving robust classification performance.

3 Neural Network Implementation

A neural network model was implemented to classify the dataset using preprocessed features and a balanced training set. The following steps were performed:

3.1 Data Preprocessing

To ensure the dataset was ready for neural network training:

- **Numerical Features**: Standard scaling was applied using `StandardScaler` to standardize the numerical data.
- **Categorical Features**: One-hot encoding was applied to convert categorical data into a numerical format. Unknown categories were handled gracefully using `handle_unknown='ignore'`.

A *ColumnTransformer* was used to preprocess the numerical and categorical features simultaneously. The resulting dataset was converted to a dense format for compatibility with the neural network model.

3.2 Class Imbalance Handling

To address class imbalance in the training dataset, **Synthetic Minority Oversampling Technique (SMOTE)** was applied. SMOTE creates synthetic samples for minority classes by interpolating between existing samples, ensuring that the neural network is trained on a balanced dataset.

3.3 Neural Network Architecture

The neural network model consisted of the following layers:

- **Input Layer**: Accepts a vector of features corresponding to the preprocessed dataset (`X_train_smote`).
- **Hidden Layers**:
  - Three fully connected (*dense*) layers with `ReLU` activation functions and layer sizes of 256, 128, and 64 neurons, respectively.
  - *L2 Regularization* was applied to all hidden layers to penalize large weights and reduce overfitting.
  - *Dropout layers* (30% dropout rate) were added after the first two hidden layers to further prevent overfitting by randomly dropping connections during training.
- **Output Layer**: A fully connected layer with a `softmax` activation function, outputting probabilities for each class. The number of output neurons matched the number of unique classes in the target variable.

3.4 Model Optimization

The model was optimized using:

- **Loss Function**: Sparse categorical cross-entropy, suitable for multi-class classification problems.
- **Optimizer**: Adam optimizer with a learning rate of 0.001 for faster convergence.
- **Early Stopping**: To avoid overfitting, training was halted early if the validation loss did not improve for 5 consecutive epochs. The model's best weights were restored at the end of training.

3.5 Training and Validation

The model was trained for a maximum of 50 epochs with a batch size of 64. Validation data (20% of the training set) was used to monitor performance during training.

3.6 Performance Evaluation

The trained model was evaluated on the validation set. Predictions were made by converting softmax probabilities to class labels. The performance metrics obtained are:

- **Accuracy**: {`0.7456418383518225`}
- **F1-Score**: {`0.7417948080789537`}

3.7 Analysis

The neural network showed promising performance with an accuracy of {`Insert Accuracy Value`} and an F1-Score of {`Insert F1-Score Value`}. The use of L2 regularization and dropout effectively minimized overfitting, as evidenced by the consistent performance on the validation set. SMOTE ensured the model learned balanced representations for all classes, leading to improved generalization. The use of early stopping further contributed to efficient training by preventing unnecessary epochs. However, further tuning of the architecture or hyperparameters could potentially yield even better results, especially in scenarios with highly complex or imbalanced datasets.

Kaggle Submissions

Among these, the **Neural Network** model provided the best score for prediction of the target variable "category", the range of expenditures a tourist spends in Mexico: 0-High; 1-Medium; 2-Low for the given dataset.

We had a kaggle score (F1 score) equal to **0.69584** for this.

References

- **Dataset Source:** Kaggle: AIM 511 Course Project - Mexican Tourist Profiles
- **Project Code Repository:** GitHub: ML-Project1

Key Libraries Used

- **Data Manipulation:** `pandas`, `numpy`
- **Data Preprocessing:** `sklearn.preprocessing` (`StandardScaler`, `OneHotEncoder`, `LabelEncoder`, `OrdinalEncoder`), `sklearn.compose` (`ColumnTransformer`)
- **Imbalanced Data Handling:** `imblearn.pipeline` (`Pipeline`), `imblearn.over_sampling` (`SMOTE`)
- **Custom Transformers:** `sklearn.base` (`BaseEstimator`, `TransformerMixin`)
- **Visualization:** `seaborn`, `matplotlib.pyplot`
- **Model Evaluation and Selection:** `sklearn.model_selection` (`train_test_split`, `GridSearchCV`), `sklearn.metrics` (`accuracy_score`, `f1_score`)
- **Machine Learning Models:** `xgboost` (`XGBClassifier`), `sklearn.ensemble` (`GradientBoostingClassifier`, `RandomForestClassifier`, `AdaBoostClassifier`), `sklearn.tree` (`DecisionTreeClassifier`), `sklearn.neighbors` (`KNeighborsClassifier`), `sklearn.svm` (`SVC`)
- **Neural Network Development:** `tensorflow` (`Sequential`, `layers`, `Adam`, `EarlyStopping`), `keras.regularizers` (`l2`)