Week 1: Data Cleaning and Feature Engineering Report

Intern Name:

Date of Submission: 18 August, 2025

Team Members:

- Hammad Mengrani
- Hemavarshini J P
- Kareem Mamdouh
- Kashaf Laiq
- Laiba Jawaid
- Mohammad Khan
- Priyanka Kute
- Ramu Mallah
- Reeba Joshi
- Rifat Islam
- Sultan Khan
- Sumaya Mateen
- Tyler Ligon

Cleaned Dataset Link:

https://drive.google.com/file/d/1tKl0i_bEejvdLun18mEj5fXd8C3uWoXs/view?usp=sharing

Spreadsheet link

https://docs.google.com/spreadsheets/d/1tJo9miCSjpej84iWHta97iEqT48OaMx_Js2jq3_YI1E/edit?usp=sharing

1. Introduction

- **Purpose:**

The primary objective of Week 1 was to clean and preprocess the raw dataset to ensure its accuracy, consistency, and readiness for further analysis. The purpose of this report is to document the data cleaning and preprocessing steps. The key tasks included handling missing values, removing duplicates, standardizing formats, and applying necessary transformations to improve overall data quality. In addition, new features were created from the cleaned dataset to enable more detailed and insightful analysis in the upcoming weeks.

- **Data Description:**

The dataset contains 8,558 rows and 16 columns. Each row represents a learner's application to an opportunity, while the columns contain learner details, opportunity information, and application status. The detailed description of each column is provided below:

Column Name	Description
Learner SignUp DateTime	The exact timestamp when the learner signed up or created their profile.
Opportunity Id	An identifier assigned to each opportunity.
Opportunity Name	The name of the opportunity.
Opportunity Category	The category of the opportunity (e.g., Course, Internship, Engagement).
Opportunity End Date	The end date and time of the opportunity.
First Name	The learner's first name.
Date of Birth	The learner's date of birth .
Gender	The learner's gender.
Country	The country of the learner.
Institution Name	The name of the institution/school/university of the learner.
Current/Intended Major	The current or intended field of study/major of the learner.

Entry created at	The timestamp when the learner's application entry for the opportunity was made.
Status Description	The status of the learner's opportunity application (e.g., Started, Team Allocated, Rejected).
Status Code	A numeric code corresponding to the status description.
Apply Date	The timestamp when the learner applied for the opportunity.
Opportunity Start Date	The start timestamp of the opportunity.

2. Data Cleaning Process

- Converted columns to their appropriate data types (e.g., Datetime, INT, String).
- Trimmed extra and unnecessary white spaces across all fields.
- Cleaned and standardized key text columns (Name, Major, Institution, Country) by:
    - Correcting misspellings and variations.
    - Removing invalid or unusually short entries.
    - Stripping digits and symbols.
    - Normalizing spaces.
    - Ensuring consistent capitalization.
    - Mapping inconsistent values to standard forms.
    - Translating non-English text into English for consistency.
    - Replacing unusable values with NaN.
- Check the duplicate records in the data.
- Filled missing values in "Opportunity Start Date" with the mode date of each group, after grouping by Opportunity Name.
- Dropped rows that still contained null values after standardization to ensure a clean and consistent dataset.

3. Data Validation

- **Data Type:**

Data Type Before	Data Type After

Learner SignUp DateTime	object
Opportunity Id	object
Opportunity Name	object
Opportunity Category	object
Opportunity End Date	object
First Name	object
Date of Birth	object
Gender	object
Country	object
Institution Name	object
Current/Intended Major	object
Entry created at	datetime64[ns]
Status Description	object
Status Code	int64
Apply Date	object
Opportunity Start Date	object

Learner SignUp DateTime	datetime64[ns]
Opportunity Id	object
Opportunity Name	string[python]
Opportunity Category	string[python]
Opportunity End Date	datetime64[ns]
First Name	string[python]
Date of Birth	object
Gender	string[python]
Country	string[python]
Institution Name	string[python]
Current/Intended Major	string[python]
Entry created at	datetime64[ns]
Status Description	string[python]
Status Code	int64
Apply Date	datetime64[ns]
Opportunity Start Date	datetime64[ns]

- **Null Values:**

Null Values Before Standardization	
Learner SignUp DateTime	0
Opportunity Id	0
Opportunity Name	0
Opportunity Category	0
Opportunity End Date	0
First Name	0
Date of Birth	0
Gender	0
Country	0
Institution Name	5
Current/Intended Major	5
Entry created at	0
Status Description	0
Status Code	0
Apply Date	0
Opportunity Start Date	3794

Null Values After Standardization	
Learner SignUp DateTime	0
Opportunity Id	0
Opportunity Name	0
Opportunity Category	0
Opportunity End Date	0
First Name	83
Date of Birth	0
Gender	0
Country	0
Institution Name	37
Current/Intended Major	151
Entry created at	0
Status Description	0
Status Code	0
Apply Date	0
Opportunity Start Date	59

Null Values After Dropping	
Learner SignUp DateTime	0
Opportunity Id	0
Opportunity Name	0
Opportunity Category	0
Opportunity End Date	0
First Name	0
Date of Birth	0
Gender	0
Country	0
Institution Name	0
Current/Intended Major	0
Entry created at	0
Status Description	0
Status Code	0
Apply Date	0
Opportunity Start Date	0

- **Total records:**

Total Records Before	Total Records After
(8558, 16)	(8227, 16)

4. Feature Engineering

Date/Time Feature Engineering:

```
# 1. DATE/TIME FEATURE ENGINEERING
# ============================================================
print("\n📅 1. DATE/TIME FEATURES:")

# Convert date columns to datetime
date_columns = ['Learner SignUp DateTime', 'Date of Birth', 'Apply Date',
                'Opportunity End Date', 'Opportunity Start Date', 'Entry created at']

for col in date_columns:
    if col in df.columns:
        df[col] = pd.to_datetime(df[col], errors='coerce')


df['Current_Age'] = ((datetime.now() - df['Date of Birth']).dt.days / 365.25).astype(int)
df['Age_Group'] = pd.cut(df['Current_Age'],
                         bins=[0, 20, 25, 30, 35, 100],
                         labels=['Teen', 'Young_Adult', 'Adult', 'Mid_Adult', 'Senior'])


df['Signup_to_Apply_Days'] = (df['Apply Date'] - df['Learner SignUp DateTime']).dt.days
df['Course_Duration_Days'] = (df['Opportunity End Date'] - df['Opportunity Start Date']).dt.days


df['Signup_Month'] = df['Learner SignUp DateTime'].dt.month
df['Signup_Weekday'] = df['Learner SignUp DateTime'].dt.day_name()
df['Signup_Hour'] = df['Learner SignUp DateTime'].dt.hour
df['Is_Weekend_Signup'] = df['Learner SignUp DateTime'].dt.weekday >= 5


df['Quick_Applicant'] = (df['Signup_to_Apply_Days'] <= 1).astype(int)

print("✅ Date features created:")
print("   - Current_Age, Age_Group")
print("   - Signup_to_Apply_Days, Course_Duration_Days")
print("   - Signup_Month, Signup_Weekday, Signup_Hour")
print("   - Is_Weekend_Signup, Quick_Applicant")

df.head()

df['Invalid_Application_Timing'] = df['Signup_to_Apply_Days'] < 0

df.head()
```

Implementation Overview:

- **Date Conversion**: All temporal columns were converted to proper datetime format using pd.to_datetime() with error handling to manage invalid entries.
- **Age Features**: Calculated current age by subtracting birth dates from current timestamp and categorized learners into age groups (Teen, Young Adult, Adult, Mid Adult, Senior) using predefined bins.
- **Behavioral Timing**: Engineered time-based features including days between signup and application (Signup_to_Apply_Days), course duration calculations, and signup timing patterns (month, weekday, hour).
- **Boolean Flags**: Created binary indicators for weekend signups, quick applicants (those applying within 1 day of registration), and invalid application timing flag to identify data quality issues where apply date occurs before signup date.
- **Business Value**: These temporal features capture user behavior patterns, application urgency, seasonal trends, and engagement levels that transform raw timestamp data into actionable insights for predictive modeling.

Categorical Features:

```
print("\n🏷️  2. CATEGORICAL FEATURES:")


df['Gender_Encoded'] = df['Gender'].map({
    'Male': 1, 'Female': 2, 'Other': 3, "Don't want to specify": 4
})
country_mapping = {
    'United States': 'North_America',
    'Canada': 'North_America',
    'India': 'South_Asia',
    'Pakistan': 'South_Asia',
    'Bangladesh': 'South_Asia',
    'Nigeria': 'Africa',
    'Ghana': 'Africa',
    'Egypt': 'Africa',
    'United Kingdom': 'Europe',
    'Germany': 'Europe'
}
df['Region'] = df['Country'].map(country_mapping).fillna('Other')
def categorize_major(major):
    if pd.isna(major):
        return "Other"
    major_lower = major.lower()

    if any(word in major_lower for word in ["computer", "software", "cyber", "data", "information", "ai", "machine learning", "it"]):
        return "Tech"
    elif any(word in major_lower for word in ["business", "finance", "marketing", "management", "commerce", "accounting"]):
        return "Business"
    elif any(word in major_lower for word in ["health", "medicine", "nursing", "pharmacy", "biomedical", "public health", "medical", "clinical"]):
        return "Health"
    else:
        return "Other"

df["Major_Category"] = df["Current/Intended Major"].apply(categorize_major)

# Status success indicator
success_statuses = ['Started', 'Team Allocated', 'Rewards Award']
df['Is_Successful'] = df['Status Description'].isin(success_statuses).astype(int)
print("✅ Categorical features created:")
print("   - Gender_Encoded")
print("   - Region (country grouping)")
print("   - Major_Category")
print("   - Is_Successful")
```

Implementation Overview

- **Gender Encoding**: Converted categorical gender values to numerical format using mapping (Male=1, Female=2, Other=3, "Don't want to specify"=4) to enable machine learning model compatibility.
- **Regional Grouping**: Created regional classifications by mapping countries to broader geographic regions (e.g., USA/Canada to North America, India/Pakistan/Bangladesh to South Asia) with unknown countries categorized as "Other".
- **Major Categorization**: Implemented keyword-based classification function to group academic majors into four main categories (Tech, Business, Health, Other) using domain-specific terms for automated classification.
- **Success Indicator**: Generated binary success flag by identifying successful application statuses ('Started', 'Team Allocated', 'Rewards Award') to create target variable for predictive modeling.
- **Business Value**: These categorical transformations reduce dimensionality, handle data inconsistencies, and create meaningful groupings that improve model interpretability and performance.

Behavioral Features:

```
print("\n🎯 4. BEHAVIORAL FEATURES:")


df['Same_Day_Application'] = (df['Signup_to_Apply_Days'] == 0).astype(int)
df['Late_Applicant'] = (df['Signup_to_Apply_Days'] > 30).astype(int)


opportunity_counts = df['Opportunity Name'].value_counts()
df['Opportunity_Popularity'] = df['Opportunity Name'].map(opportunity_counts)
df['Popular_Opportunity'] = (df['Opportunity_Popularity'] > df['Opportunity_Popularity'].median()).astype(int)

print("✅ Behavioral features created:")
print("   - Same_Day_Application, Late_Applicant")
print("   - Opportunity_Popularity, Popular_Opportunity")
```

Implementation Overview

- **Application Timing Patterns**: Created binary flags for same-day applications (Same_Day_Application) and late applicants who apply more than 30 days after signup (Late_Applicant) to identify user engagement behaviors and procrastination patterns.

- **Opportunity Popularity Metrics**: Calculated popularity scores by counting total applications per opportunity and created binary indicators for opportunities receiving above-median applications (Popular_Opportunity) to understand demand patterns and competition levels.
- **Business Value**: These behavioral features capture user decision-making patterns, opportunity attractiveness, and application urgency that help predict success rates and optimize opportunity placement strategies.

Interaction Features:

```
print("\n🔄 5. INTERACTION FEATURES:")


df['Age_Major_Combo'] = df['Age_Group'].astype(str) + '_' + df['Major_Category']


df['Country_Gender'] = df['Country'] + '_' + df['Gender']


df['Region_Opportunity'] = df['Region'] + '_' + df['Opportunity Category']

print("✅ Interaction features created:")
print("   - Age_Major_Combo")
print("   - Country_Gender")
print("   - Region_Opportunity")
```

Implementation Overview

- **Cross-Categorical Combinations**: Created composite features by combining different categorical variables including age group with major category (Age_Major_Combo), country with gender (Country_Gender), and region with opportunity type (Region_Opportunity).
- **Feature Interaction Capture**: These interaction features capture complex relationships between demographic, geographic, and academic variables that individual features might miss, enabling the model to identify nuanced patterns in user behavior.
- **Business Value**: Interaction features reveal hidden patterns such as regional preferences for specific opportunity types, gender-based country trends, and age-major combinations that

drive application success, providing deeper insights for targeted marketing and opportunity design.

Feature Scaling:

```
from sklearn.preprocessing import StandardScaler

print("\n📊 5. FEATURE SCALING:")


numerical_cols = ['Current_Age', 'Course_Duration_Days',
                  'Opportunity_Popularity', 'Signup_to_Apply_Days']




scaler = StandardScaler()
df[numerical_cols] = scaler.fit_transform(df[numerical_cols])

print("✅ Features scaled using StandardScaler!")
print(f"   - Age mean: {df['Current_Age'].mean():.2f}, std: {df['Current_Age'].std():.2f}")
print(f"   - Duration mean: {df['Course_Duration_Days'].mean():.2f}, std: {df['Course_Duration_Days'].std():.2f}")
print(f"   - Popularity mean: {df['Opportunity_Popularity'].mean():.2f}, std: {df['Opportunity_Popularity'].std():.2f}")
```

Implementation Overview

- **StandardScaler Application**: Applied StandardScaler to numerical features including Current Age, Course Duration Days, Opportunity Popularity, and Signup to Apply Days to normalize all values to have zero mean and unit variance.
- **Scale Normalization**: Scaling ensures that features with different units and ranges (e.g., age in years vs. popularity counts) contribute equally to machine learning algorithms, preventing features with larger scales from dominating the model.
- **Business Value**: Proper feature scaling improves model convergence, reduces training time, and ensures fair feature importance across all numerical variables, leading to more accurate and stable predictive performance.

Engagement Score:

```
print("\n🏆 6. ENGAGEMENT SCORE:")


df['Engagement_Score'] = (
    df['Quick_Applicant'] * 0.25 +
    df['Same_Day_Application'] * 0.15 +
    df['Popular_Opportunity'] * 0.20 +
    df['Is_Successful'] * 0.40
)


df['Engagement_Level'] = pd.cut(df['Engagement_Score'],
                                bins=[0, 0.3, 0.6, 1.0],
                                labels=['Low', 'Medium', 'High'])

print("✅ Engagement Score created:")
print(f"   - Score range: {df['Engagement_Score'].min():.2f} to {df['Engagement_Score'].max():.2f}")
print(f"   - High engagement: {(df['Engagement_Level'] == 'High').sum()} students")
print(f"   - Medium engagement: {(df['Engagement_Level'] == 'Medium').sum()} students")
print(f"   - Low engagement: {(df['Engagement_Level'] == 'Low').sum()} students")

df.head()

df.to_csv('fully_cleaned_data.csv', index=False)

feature_engineering (3).py
Displaying feature_engineering (3).py.
```

Implementation Overview

- **Weighted Composite Score**: Created a comprehensive engagement score by combining multiple behavioral indicators with assigned weights: Is_Successful (40%), Quick_Applicant (25%), Popular_Opportunity (20%), and Same_Day_Application (15%) to reflect relative importance of each engagement factor.
- **Categorical Binning**: Converted continuous engagement scores into interpretable categories (Low: 0-0.3, Medium: 0.3-0.6, High: 0.6-1.0) using pd.cut() to enable easy segmentation and analysis of learner engagement levels.
- **Business Value**: The engagement score provides a single metric to identify highly engaged learners, optimize resource allocation, and develop targeted retention strategies based on comprehensive behavioral patterns rather than individual indicators.

Timing Features:

- Apply_vs_Start_Gap: Days between application and start date
- Signup_vs_Start_Gap: Days between signup and start date
- Signup_DayPart: Time of day (Morning/Afternoon/Evening/Night)

User Behavior Features:

- Applications_per_User: Total applications per user
- Unique_Categories_per_User: Different categories applied to

```python
# ---------------- Convert Dates ----------------
user_data['Apply Date'] = pd.to_datetime(user_data['Apply Date'], errors='coerce')
user_data['Opportunity Start Date'] = pd.to_datetime(user_data['Opportunity Start Date'], errors='coerce')
user_data['Learner SignUp DateTime'] = pd.to_datetime(user_data['Learner SignUp DateTime'], errors='coerce')

# -------------------------
#Timing Features:
# 1. Apply vs Start Gap
user_data["Apply_vs_Start_Gap"] = (
    user_data["Apply Date"] - user_data["Opportunity Start Date"]
).dt.days

# 2. Signup vs Start Gap
user_data["Signup_vs_Start_Gap"] = (
    user_data["Learner SignUp DateTime"] - user_data["Opportunity Start Date"]
).dt.days

#2.User Behavior Features:
# 3. Applications per User
user_data["Applications_per_User"] = user_data.groupby("First Name")["Opportunity Id"].transform("count")

# 4. Unique Categories per User
user_data["Unique_Categories_per_User"] = user_data.groupby("First Name")["Category"].transform("nunique")

# 5. User Choice Ratio (Unique Categories / Applications)
user_data["User_Choice_Ratio"] = (
    user_data["Unique_Categories_per_User"] / (user_data["Applications_per_User"] + 1e-6)
```

Competition & Success Features:

- Applications_for_Opportunity: Applicants per opportunity
- User_Success_Rate: Success percentage per user

Demographic Feature:

- Region_AgeGroup: Combines region and age group

```python
#Demographic Features:
# 6. Region + Age Group Feature
user_data["Region_AgeGroup"] = user_data["Region"].astype(str) + "_" + user_data["Age Group"].astype(str)

# 7. Signup Season
user_data["Signup_Season"] = user_data["Learner SignUp DateTime"].dt.month % 12 // 3 + 1

# 8. Apply Weekday
user_data["Apply_Weekday"] = user_data["Apply Date"].dt.day_name()

# 9. Signup DayPart
def daypart(hour):
    if 5 <= hour < 12:
        return "Morning"
    elif 12 <= hour < 17:
        return "Afternoon"
    elif 17 <= hour < 21:
        return "Evening"
    else:
        return "Night"

user_data["Signup_DayPart"] = user_data["Learner SignUp DateTime"].dt.hour.apply(daypart)

# 10. Region Normalized Applications
user_data["Region_Normalized_Applications"] = (
    user_data.groupby("Region")["Applications_per_User"]
    .transform(lambda x: (x - x.min()) / (x.max() - x.min() + 1e-6))
```



```python

# 11. Category Diversity
user_data["Category_Diversity"] = user_data.groupby("Category")["Opportunity Id"].transform("nunique")

# 12. User Success Rate (Applications vs Wins)
if "Win" in user_data.columns:  # assuming 'Win' column marks successful applications
    user_data["User_Success_Rate"] = (
        user_data.groupby("First Name")["Win"].transform("sum") / (user_data["Applications_per_User"] + 1e-6)
    )
else:
    user_data["User_Success_Rate"] = np.nan  # placeholder if 'Win' column not available

#Competition & Success Features:
# 13. Applications for Opportunity (Competition Feature)
user_data["Applications_for_Opportunity"] = user_data.groupby("Opportunity Id")["First Name"].transform("count")

# -------------------------
# Export CSV with specific feature columns only
feature_columns = [
    'Apply_vs_Start_Gap',
    'Signup_vs_Start_Gap',
    'Applications_per_User',
    'Unique_Categories_per_User',
    'User_Choice_Ratio',
    'Region_AgeGroup',
    'Signup_Season',
    'Apply_Weekday',
    'Signup_DayPart',
```

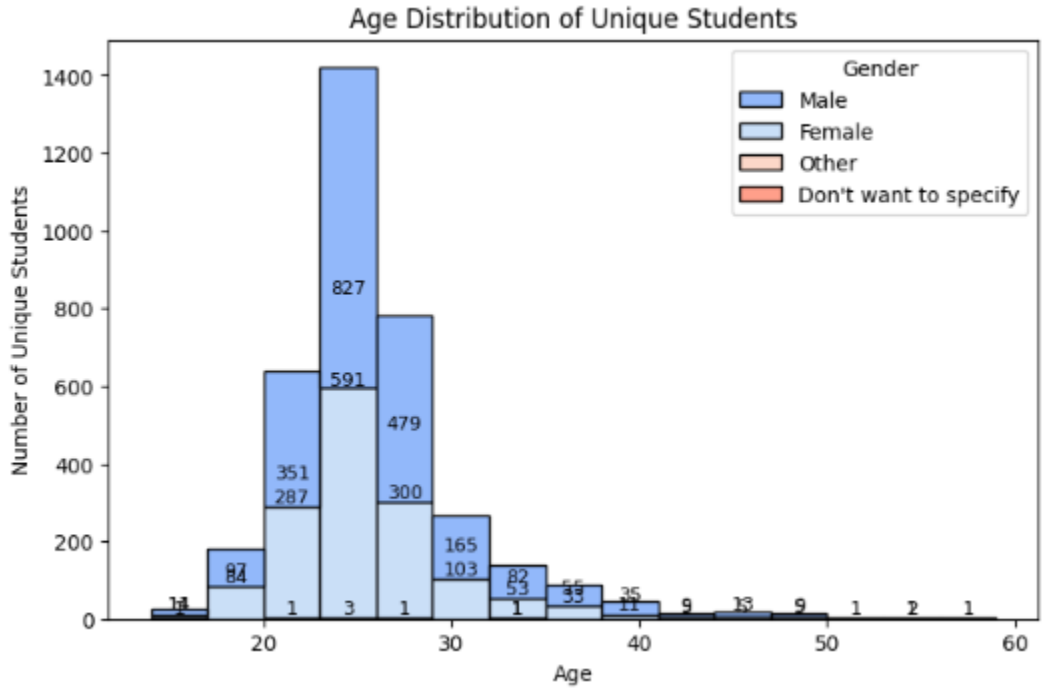

Impact:

- Better predictive modeling and user segmentation
- Improved recommendation systems and A/B testing
- Supports business intelligence decisions

Age Distribution of Unique Students by Gender:

```
# Make a safe copy to avoid SettingWithCopyWarning
unique_students = unique_students.copy()

# Convert Date of Birth and calculate Age
unique_students['Date of Birth'] = pd.to_datetime(unique_students['Date of Birth'], errors='coerce')
unique_students['Age'] = (pd.to_datetime("today") - unique_students['Date of Birth']).dt.days // 365

# Plot histogram with hue (Gender) and stacked bars
plt.figure(figsize=(8,5))
ax = sns.histplot(data=unique_students, x='Age', bins=15, hue='Gender', multiple='stack', palette='coolwarm')

plt.title("Age Distribution of Unique Students")
plt.xlabel("Age")
plt.ylabel("Number of Unique Students")

# Add counts on top of each stacked bar
for p in ax.patches:
    height = p.get_height()
    if height > 0:  # only annotate bars with height > 0
        ax.annotate(f'{int(height)}',
                    (p.get_x() + p.get_width() / 2., height),
                    ha='center', va='bottom', fontsize=9)
```

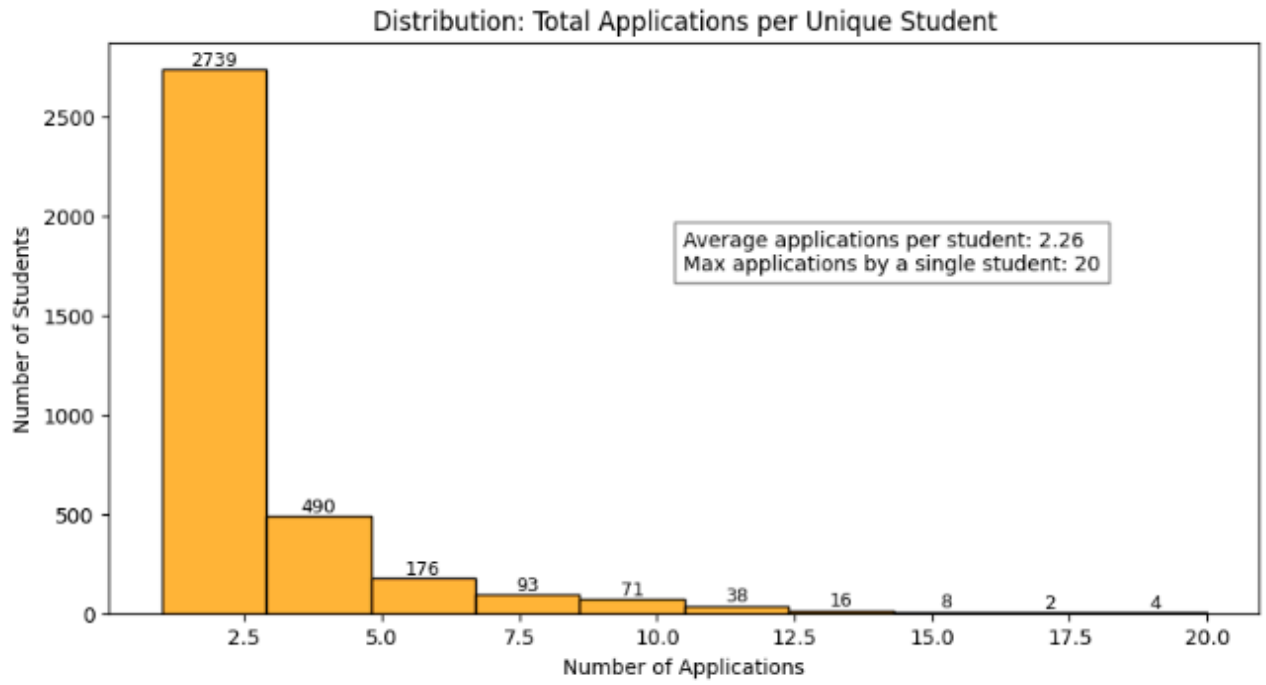
Output:



Distribution Of Total Applications Per Unique Student:

```
# Count total applications per student
applications_per_student = df.groupby('Student_ID')['Opportunity Id'].count().reset_index()
applications_per_student.rename(columns={'Opportunity Id':'Num_Applications'}, inplace=True)

plt.figure(figsize=(10,5))
ax = sns.histplot(applications_per_student['Num_Applications'], bins=10, kde=False, color="orange")
plt.title("Distribution: Total Applications per Unique Student")
plt.xlabel("Number of Applications")
plt.ylabel("Number of Students")

# Add counts on top of each bar
for p in ax.patches:
    height = p.get_height()
    if height > 0:
        ax.annotate(f'{int(height)}',
                    (p.get_x() + p.get_width() / 2., height),
                    ha='center', va='bottom', fontsize=9)

# Add summary statistics on the plot
avg_apps = applications_per_student['Num_Applications'].mean()
max_apps = applications_per_student['Num_Applications'].max()
plt.text(x=0.5*ax.get_xlim()[1], y=0.6*ax.get_ylim()[1],
         s=f"Average applications per student: {avg_apps:.2f}\nMax applications by a single student: {max_apps}",
         bbox=dict(facecolor='white', alpha=0.5), fontsize=10)

plt.show()
```

Output:



5. Conclusion

- **Summary:**

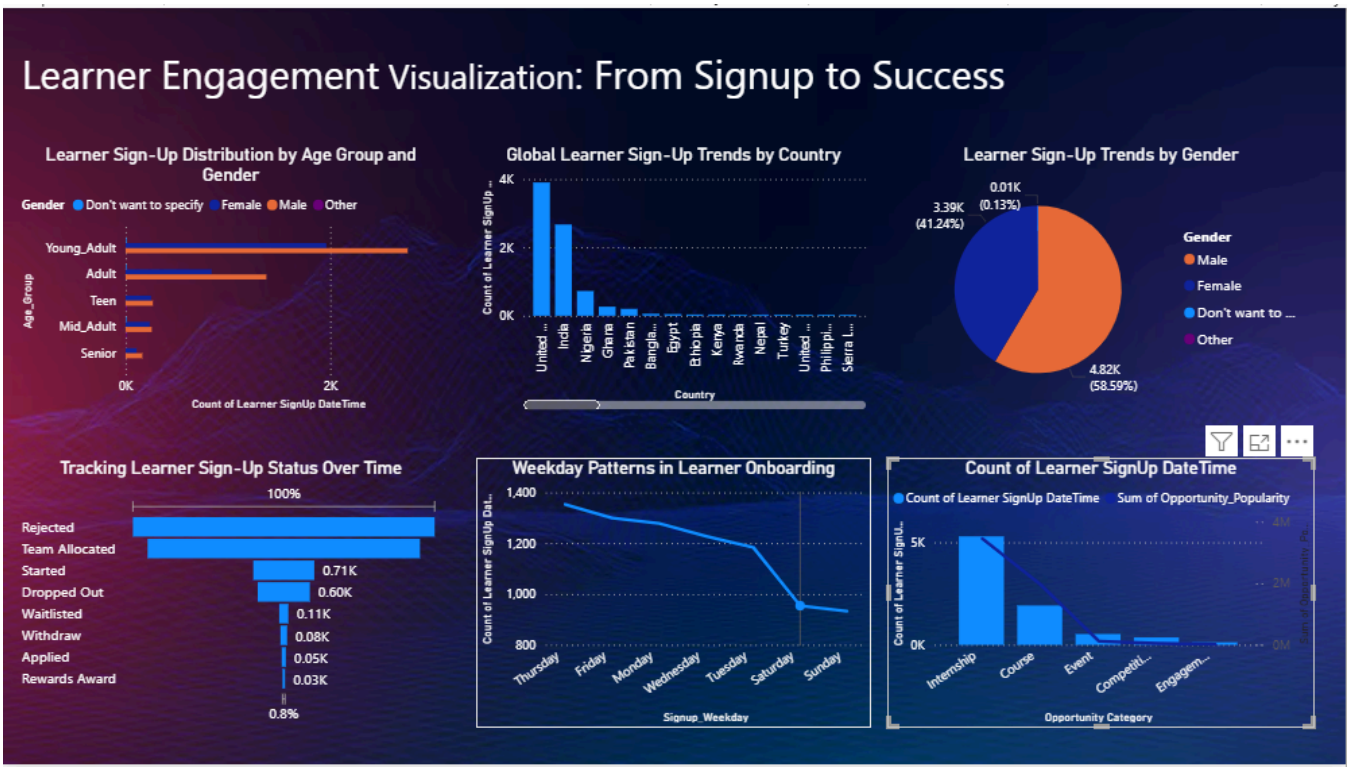
The dataset was thoroughly cleaned and transformed to ensure accuracy, consistency, and reliability. Key tasks included handling missing dates, checking the duplicates, converting columns to appropriate data types, standardizing text fields, and translating non-English values. Feature engineering was also performed that enriched the dataset, making it more suitable for deeper analysis. As a result, the dataset was refined from 8,558 records to 8,227 records, effectively eliminating noise and ensuring that only valid, high-quality data remains for further analysis.

- **Next Steps:**

In Week 2, the focus will shift from data cleaning to exploratory data analysis (EDA). The cleaned dataset will be leveraged to generate descriptive statistics, uncover meaningful patterns, and visualize key trends across learners, opportunities, and application statuses. These insights will lay the foundation for more advanced analysis in the subsequent weeks.

6. Appendix:

Visualization Summary of Learning Engagement Dataset



Data Cleaning Codes:

```
def clean_date_columns(df, date_columns=None):
    if date_columns is None:
        date_columns = [
            'Learner SignUp DateTime',
            'Opportunity End Date',
            'Entry created at',
            'Apply Date',
            'Opportunity Start Date'
        ]
    EXCEL_BASE_DATE = datetime(1899, 12, 30)
    def parse_date_flexible(value):
        if pd.isna(value):
            return pd.NaT
        if isinstance(value, datetime):
            return value
        value = str(value).strip()
        if 'days' in value:
            try:
                days = float(re.search(r'(\d+)', value).group(1))
                return EXCEL_BASE_DATE + timedelta(days=days)
            except:
                return pd.NaT
        value = re.sub(r'24:(\d{2}:\d{2})', r'23:\1', value)
        try:
            dt = parser.parse(value, fuzzy=True, dayfirst=False, yearfirst=False)
            return dt
        except:
            return pd.NaT
    for col in date_columns:
        if col in df.columns:
            df[col] = df[col].apply(parse_date_flexible)
            # Optional: uniform string format
            df[col] = df[col].apply(lambda x: x.strftime('%Y-%m-%d %H:%M:%S') if pd.notna(x) else pd.NaT)
            # Convert back to datetime type
            df[col] = pd.to_datetime(df[col], errors='coerce')
    return df
df = clean_date_columns(df)
df['Date of Birth'] = pd.to_datetime(df['Date of Birth'], errors='coerce')
df['Date of Birth'] = df['Date of Birth'].dt.date
string_columns = ["First Name", "Country", "Institution Name", "Current/Intended Major", "Opportunity Name",
                  "Opportunity Category", "Status Description", "Gender"]
df[string_columns] = df[string_columns].astype("string")
print(df.dtypes)
```

This code cleans and standardizes date and string columns in a dataset. It handles messy date formats, Excel-style serial dates, and invalid times, converting them into a uniform YYYY-MM-DD HH:MM:SS format. The “Date of Birth” column is kept only as a date, while selected text columns are converted to string type for consistency.

```python
import pandas as pd
from scipy.stats import mode

df['Opportunity Start Date'] = pd.to_datetime(df['Opportunity Start Date'], errors='coerce')

def fill_with_mode(group):
    if group['Opportunity Start Date'].notna().any():
        mode_date = group['Opportunity Start Date'].mode()[0]
        group['Opportunity Start Date'] = group['Opportunity Start Date'].fillna(mode_date)
    return group

df = df.groupby('Opportunity Name', group_keys=False).apply(fill_with_mode)

print(df[['Opportunity Name', 'Opportunity Start Date']].sort_values('Opportunity Start Date').to_string(index=False))
```

The code handles missing values in the "Opportunity Start Date" column by filling them with the most frequent date for each "Opportunity Name". First, it converts all entries in the column into proper datetime format, replacing invalid ones with NaT. Then, it groups the data by opportunity name and applies a function that replaces missing dates with the group's mode.

```python
NAME_CORRECTIONS = {
    "zynb n`mn": "Zaynab Numan",
    "grimaa": "Garima"
}
INVALID_NAMES = ["M", "C", "S", "Sk", "Sp", "Md", "Fnu", "Rb", "G.",
                 "E", "V", "K", "I'm", "G", "Nu", "Om", "Mr", "De", "F"]
def clean_first_name(name):
    if pd.isna(name):
        return pd.NA
    # Convert to string, strip spaces
    name = str(name).strip()
    # Transliterate to ASCII
    name_ascii = unidecode(name).lower()
    # Apply corrections first
    if name_ascii in NAME_CORRECTIONS:
        return NAME_CORRECTIONS[name_ascii]
    # Remove names that are only digits, dashes, or spaces
    if re.fullmatch(r'[\d\-\s]+', name_ascii):
        return pd.NA
    # Remove words with digits
    words = [w for w in name_ascii.split() if not any(c.isdigit() for c in w)]
    # Remove invalid words
    words = [w for w in words if w.title() not in INVALID_NAMES]
    if not words:
        return pd.NA
    cleaned_name = ' '.join([w.capitalize() for w in words])
    # Discard initials like "A." or "B"
    if re.fullmatch(r"[A-Za-z]\.?", cleaned_name):
        return pd.NA
    # Minimum length check
    if len(cleaned_name) < 3:
        return pd.NA
    return cleaned_name
# Apply to your DataFrame
df['First Name'] = df['First Name'].apply(clean_first_name)
```

This code is designed to clean and standardize first names in your dataset. It starts by handling missing values and stripping extra spaces. Then it transliterates names into plain English letters (removing accents) and applies manual corrections for known misspellings. After cleaning, it capitalizes the remaining valid words to produce a neat and standardized first name.

```python
country_mapping = {
    'Iran Islamic Republic Of Persian Gulf': 'Iran',
    'Iran, Islamic Republic Of Persian Gulf': 'Iran',
    'Korea, Republic Of South Korea': 'South Korea',
    'Tanzania, United Republic Of Tanzania': 'Tanzania',
    'Libyan Arab Jamahiriya': 'Libya',
    'Virgin Islands, U.S.': 'U.S. Virgin Islands',
    'Falkland Islands (Malvinas)': 'Falkland Islands'
}

df['Country'] = df['Country'].replace(country_mapping)
df['Country'] = df['Country'].str.strip().str.title()

unique_countries = sorted(df['Country'].unique())
print(unique_countries)
```

This code is cleaning and standardizing country names in your dataset. First, it applies a mapping (country_mapping) to replace inconsistent or long variations. Then, it trims any extra spaces and converts country names into title case. Finally, it extracts all unique country names, sorts them alphabetically, and prints the cleaned list.