Task Description

In Task-1, you worked on the Student Performance Dataset and learned how to clean and analyze data. Now, you will take one of the most famous datasets in Data Science – the Titanic Dataset. This dataset contains passenger details (age, class, gender, etc.) and whether they survived. Your task is to analyze survival patterns and visualize them. What You'll Build (Requirements)

1. Load Titanic dataset from Kaggle.
2. Clean data (handle missing values like Age).
3. Answer questions such as: o Who survived more: males or females? o Did passenger class affect survival chances? o What was the survival rate by age group?
4. Visualize using Seaborn / Matplotlib: o Bar chart of survival by gender. o Bar chart of survival by class. o Histogram of passenger ages. Why This Task Is Important o Teaches you how to handle real-world messy data. o Introduces data visualization, a key skill for Data Scientists. o Helps you practice asking and answering business questions with data.

1. Introduction

The Titanic dataset is one of the most widely used beginner datasets in Data Science. It contains information about passengers on the Titanic — including their age, gender, class, and whether they survived the tragic disaster.

The goal of this assignment is to analyze the dataset, engineer meaningful features, handle missing values, build predictive models, and understand survival patterns.

Installing and Importing Libraries

➤ pandas → used for data loading, cleaning, and manipulation

➤ numpy → used for numerical operations and array handling

➤ matplotlib / seaborn → used for data visualization and plotting graphs

➤ missingno → used to visualize missing values in the dataset

➤ statsmodels → used to create mosaic plots (equivalent to R's mosaicplot)

➤ scikit-learn (sklearn) → used for machine learning tasks such as Random Forest models and data preprocessing

In [ ]:
```
#  Install packages used
!pip install missingno statsmodels --quiet

# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno
from statsmodels.graphics.mosaicplot import mosaic

from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

sns.set_style('whitegrid')
%matplotlib inline
```

Loading Data —

we upload train.csv and test.csv from your local system using Colab's upload tool.

Then:

Load both datasets using pandas.read_csv()

Combine them into a single DataFrame (full)

In [ ]:
```
#  Upload files and load
from google.colab import files
print("Please upload train.csv and test.csv (use the upload dialog).")
uploaded = files.upload()

train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')

# Combine for feature engineering
full = pd.concat([train, test], sort=False).reset_index(drop=True)
print("Shapes -> train:", train.shape, "test:", test.shape, "full:", full.shap

# Quick peek
display(full.head())
display(full.info())
```

```
Please upload train.csv and test.csv (use the upload dialog).
```

Browse... Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving test.csv to test (1).csv
Saving train.csv to train (2).csv
Shapes -> train: (891, 12) test: (418, 11) full: (1309, 12)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket
0	1	0.0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171
1	2	1.0	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599
2	3	1.0	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282
3	4	1.0	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803
4	5	0.0	3	Allen, Mr. William Henry	male	35.0	0	0	373450

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  1309 non-null   int64  
 1   Survived     891 non-null    float64
 2   Pclass       1309 non-null   int64  
 3   Name         1309 non-null   object 
 4   Sex          1309 non-null   object 
 5   Age          1046 non-null   float64
 6   SibSp        1309 non-null   int64  
 7   Parch        1309 non-null   int64  
 8   Ticket       1309 non-null   object 
 9   Fare         1308 non-null   float64
 10  Cabin        295 non-null    object 
 11  Embarked     1307 non-null   object 
dtypes: float64(3), int64(4), object(5)
memory usage: 122.8+ KB
None
```

Extracting Titles and Surnames

➤ Titles such as Mr, Mrs, Miss, Master, Dr, Rev, etc. are strong predictors of survival because they reflect: ➤ Social status ➤ Gender ➤ Age group ➤ Family role

➤ Passenger Titles were extracted from the Name column using string operations.

➤ Rare titles were grouped into a single “Rare Title” category to reduce noise and improve model stability.

➤ Surnames were extracted to help identify family relationships among passengers.

➤ These engineered features improved the model’s ability to capture social and family-based survival patterns.

In [ ]:
```
#  Titles and Surnames
full['Title'] = full['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)

# normalize rare titles
full['Title'] = full['Title'].replace(['Mlle','Ms'],'Miss')
full['Title'] = full['Title'].replace('Mme','Mrs')
rare_titles = ['Dr','Rev','Col','Major','Capt','Sir','Lady','Countess','Jonkhe
full['Title'] = full['Title'].replace(rare_titles, 'Rare Title')

full['Surname'] = full['Name'].str.split(',', expand=True)[0]

print("Title counts:")
print(full['Title'].value_counts())
print("Unique surnames:", full['Surname'].nunique())
```

```
Title counts:
Title
Mr            757
Miss          264
Mrs           198
Master         61
Rare Title     28
Don             1
Name: count, dtype: int64
Unique surnames: 875
```

```
<>:2: SyntaxWarning: invalid escape sequence '\.'
<>:2: SyntaxWarning: invalid escape sequence '\.'
/tmp/ipython-input-236139407.py:2: SyntaxWarning: invalid escape sequence '\.'
  full['Title'] = full['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)
```

Family Size & Family ID

➤ Family-related features were created because passengers traveling alone or in very large families showed different survival patterns.

➤ Fsize represents the total family size and is calculated as: ➤ SibSp + Parch + 1 (including the passenger)

➤ Family ID was created by combining the passenger's Surname with their Family Size.

➤ A bar plot and mosaic plot were used to visualize the relationship between family size and survival.

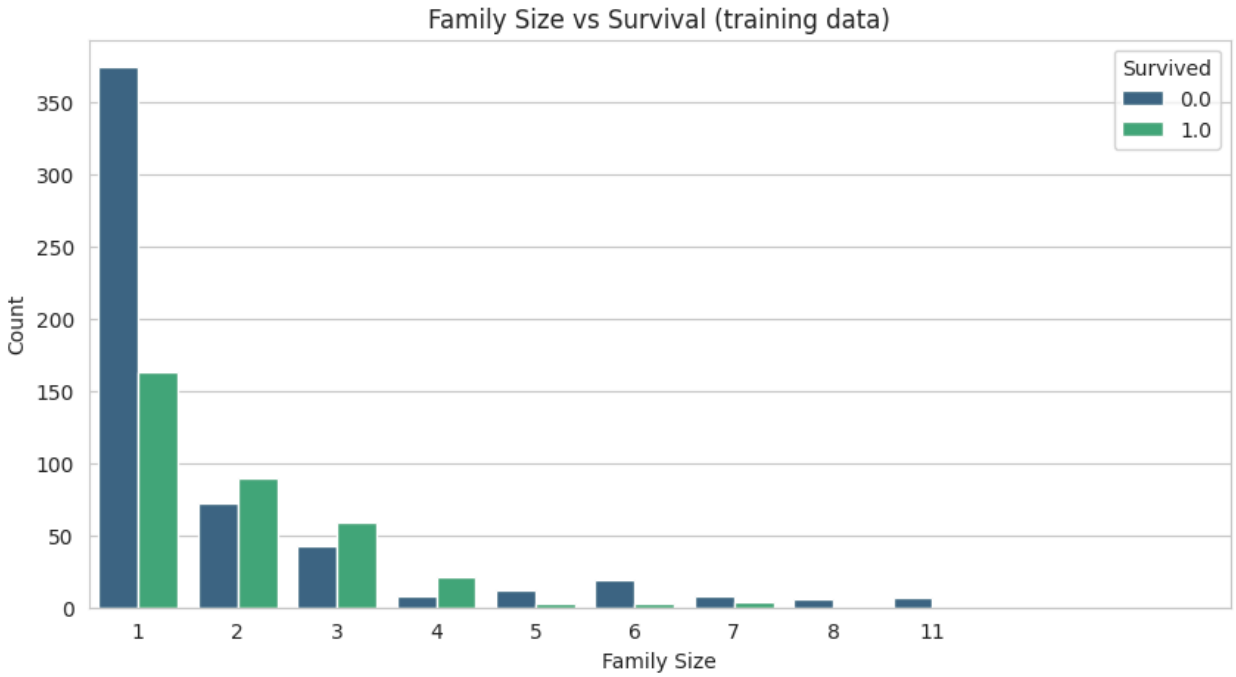
➤ Family size was then grouped into three meaningful categories: ➤ Singleton → family size = 1 ➤ Small → family size between 2 and 4 ➤ Large → family size of 5 or more

➤ This grouping reduced sparsity and improved the model's ability to learn survival patterns.

In [ ]:
```
#  Family size and Family ID
full['Fsize'] = full['SibSp'] + full['Parch'] + 1
full['Family'] = full['Surname'] + "_" + full['Fsize'].astype(str)

display(full[['Surname','SibSp','Parch','Fsize','Family']].head())

# Barplot similar to ggplot2 geom_bar(stat='count', position='dodge')
plt.figure(figsize=(10,5))
sns.countplot(data=full.iloc[:891], x='Fsize', hue='Survived', palette='viridi
plt.title("Family Size vs Survival (training data)")
plt.xlabel("Family Size")
plt.ylabel("Count")
plt.xticks(range(0, full['Fsize'].max()+1))
plt.legend(title='Survived')
plt.show()

# Discretize family size into singleton / small / large (same thresholds as R)
def fsize_cat(x):
    if x == 1:
        return 'singleton'
    elif 1 < x < 5:
        return 'small'
    else:
        return 'large'

full['FsizeD'] = full['Fsize'].apply(fsize_cat)
full['FsizeD'] = full['FsizeD'].astype('category')
```

```
print("FsizeD value counts (training data):")
print(full.iloc[:891]['FsizeD'].value_counts())
```

	Surname	SibSp	Parch	Fsize	Family
0	Braund	1	0	2	Braund_2
1	Cumings	1	0	2	Cumings_2
2	Heikkinen	0	0	1	Heikkinen_1
3	Futrelle	1	0	2	Futrelle_2
4	Allen	0	0	1	Allen_1




```
FsizeD value counts (training data):
FsizeD
singleton    537
small        292
large         62
Name: count, dtype: int64
```

Mosaic Plot for Family Size vs Survival

➤ A mosaic plot was used to visualize how survival is distributed across different family size groups.

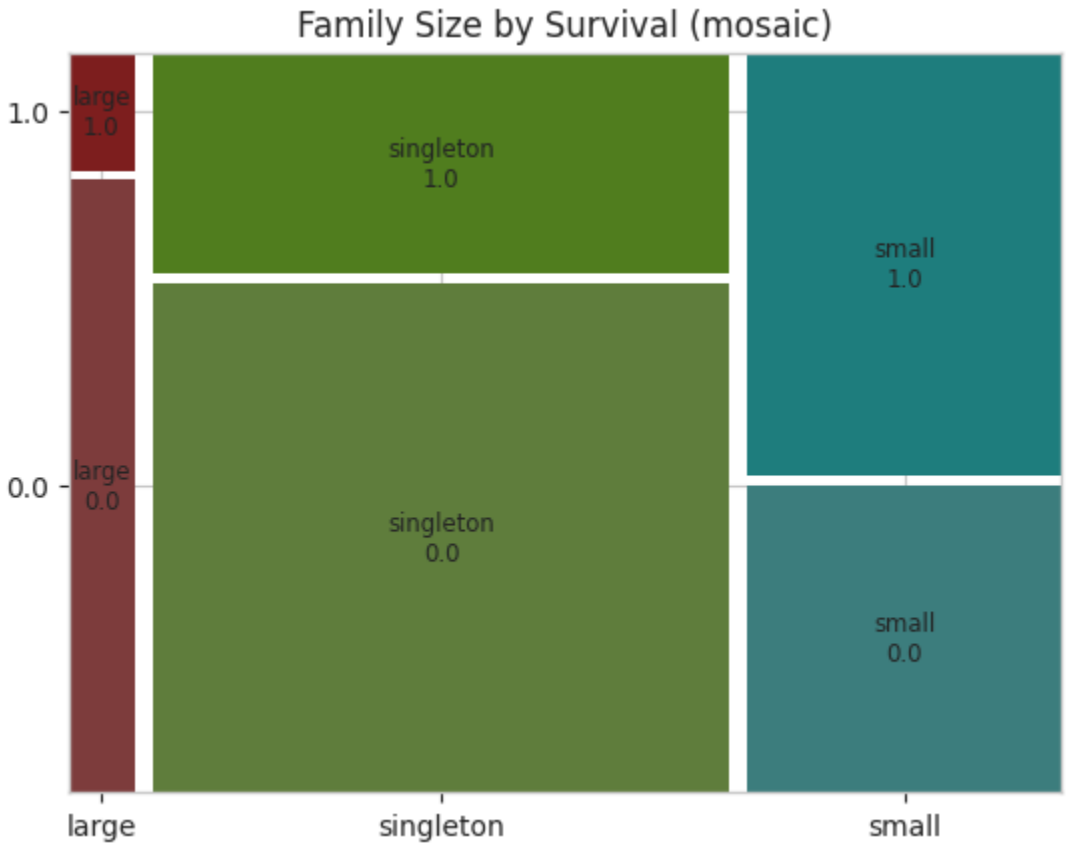
➤ Each block in the plot represents the proportion of passengers who survived or did not survive within: ➤ Singleton families ➤ Small families ➤ Large families

➤ This visualization clearly highlights survival patterns: ➤ Single travelers had lower survival chances ➤ Small families showed better survival rates ➤ Large families had low survival probability

➤ The mosaic plot complements bar charts by emphasizing proportions rather than absolute counts.

In [ ]:
```
#  Mosaic plot for Family Size by Survival (training rows only)
ct = pd.crosstab(full.iloc[:891]['FsizeD'], full.iloc[:891]['Survived'])
ct_dict = {}
# mosaic requires mapping keys like ('singleton', 0): count
for fsize in ct.index:
    for surv in ct.columns:
        ct_dict[(fsize, str(surv))] = ct.loc[fsize, surv]

plt.figure(figsize=(8,6))
mosaic(ct_dict, gap=0.02, title='Family Size by Survival (mosaic)')
plt.show()
```

```
<Figure size 800x600 with 0 Axes>
```



Embarkation vs Fare Boxplot

➤ Two passengers (PassengerId 62 and 830) were missing the Embarked value.

➤ To infer their correct embarkation port, we compared: ➤ Fare paid by the passenger ➤ Passenger class (Pclass) ➤ Typical fare distributions for each port (C, Q, S)

➤ These two passengers were temporarily removed from the dataset to avoid distortion.

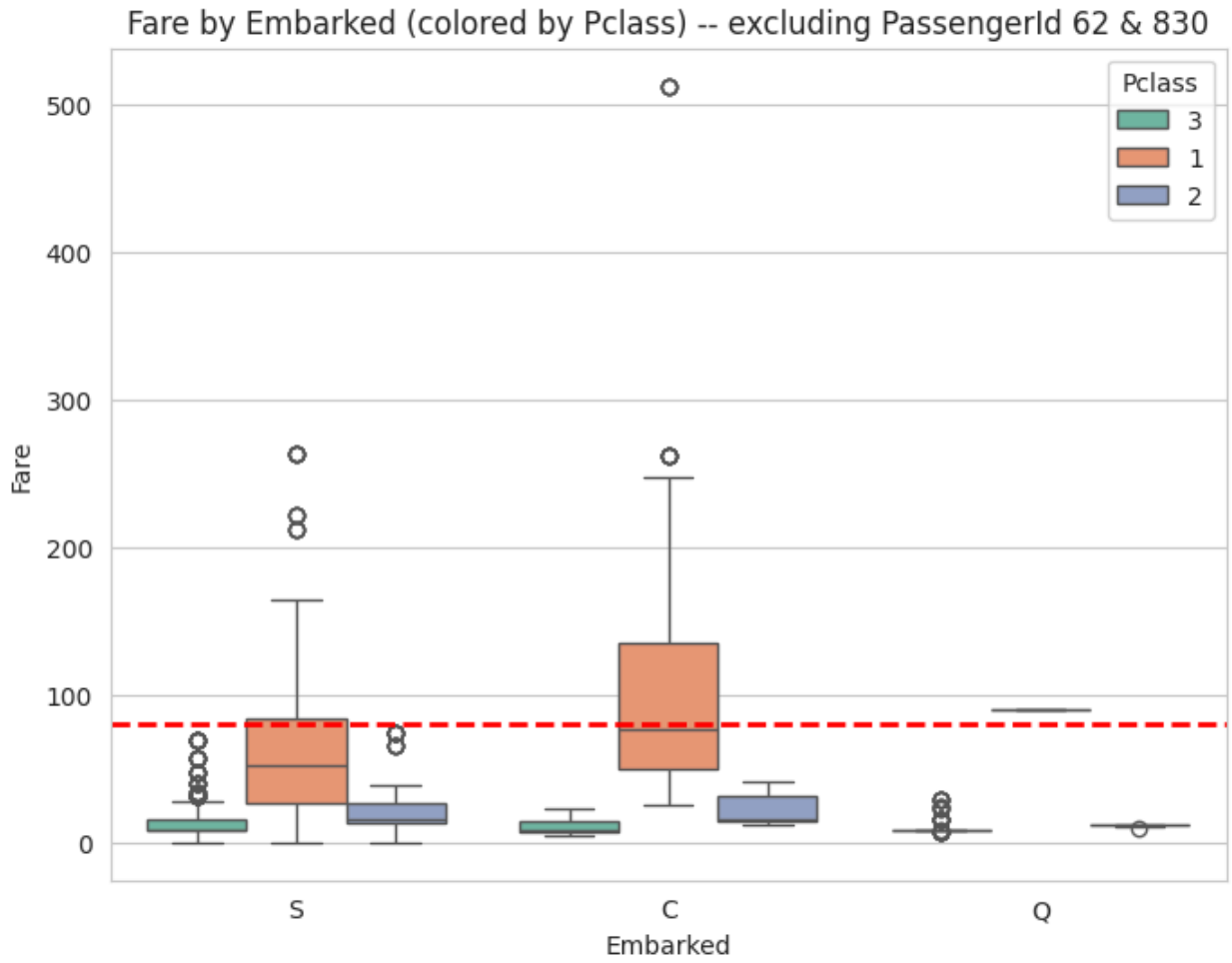
➤ A boxplot was created with: ➤ Embarked on the x-axis ➤ Fare on the y-axis ➤ Colors representing Passenger Class (Pclass)

➤ A horizontal red reference line at Fare = 80 was added to highlight fare patterns.

➤ The plot showed that passengers paying around this fare in 1st class most commonly embarked from Cherbourg (C).

➤ Based on this visual evidence, the missing Embarked values were filled as ‘C’.

In [ ]:
```
# Embarked vs Fare boxplot excluding PassengerId 62 & 830
embark_fare = full[~full['PassengerId'].isin([62, 830])]

plt.figure(figsize=(8,6))
sns.boxplot(data=embark_fare, x='Embarked', y='Fare', hue=embark_fare['Pclass'
plt.title("Fare by Embarked (colored by Pclass) -- excluding PassengerId 62 &
# draw horizontal line at 80 like R example
plt.axhline(80, color='red', linestyle='--', linewidth=2)
plt.legend(title='Pclass')
plt.show()

# Show the fares and Pclass for the two passengers to infer
print("Passenger 62 and 830 records (before fix):")
display(full.loc[full['PassengerId'].isin([62,830]), ['PassengerId','Pclass','
```



Passenger 62 and 830 records (before fix):

	PassengerId	Pclass	Fare	Embarked
61	62	1	80.0	NaN
829	830	1	80.0	NaN

Fix Missing Embarked Values

After checking the Fare vs Embarked boxplot, we saw that passengers who paid around $80 in 1st class typically departed from Cherbourg (C).

Passenger IDs 62 and 830 match this pattern, so we fill their missing Embarked values with "C".

This step ensures the dataset has no missing embarkation values before modeling.

In [ ]:
```
#  Fill Embarked for 62 and 830 with 'C' if missing
full.loc[full['PassengerId']==62, 'Embarked'] = full.loc[full['PassengerId']==
full.loc[full['PassengerId']==830, 'Embarked'] = full.loc[full['PassengerId']=
```

```
print("After filling:")
display(full.loc[full['PassengerId'].isin([62,830]), ['PassengerId','Pclass','
```

```
After filling:
```

	PassengerId	Pclass	Fare	Embarked
61	62	1	80.0	C
829	830	1	80.0	C

Fare Density Plot for Pclass 3 & Embarked S

Passenger 1044 has a missing Fare value. To fill it correctly, we analyze fares of passengers who are:

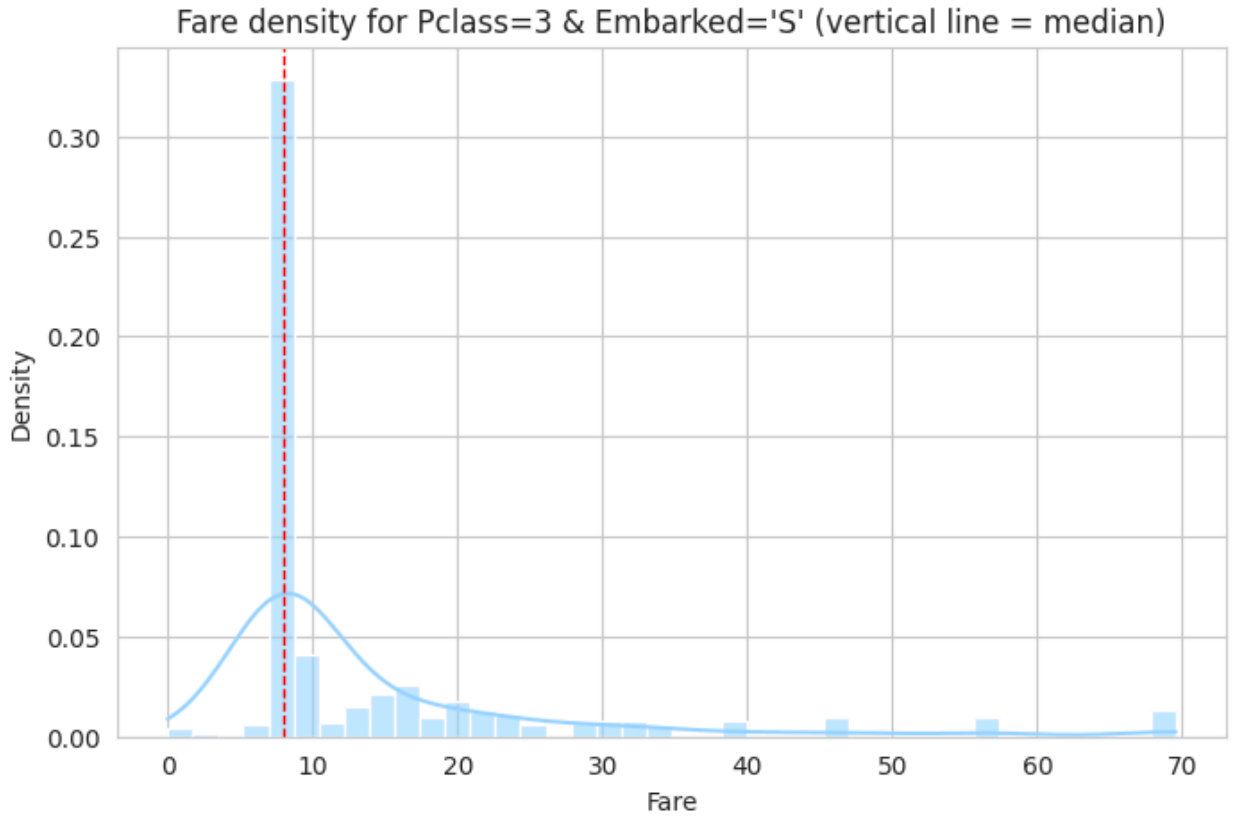
in 3rd class (Pclass = 3)

and embarked from Southampton (S)

We plot the fare distribution and mark the median fare with a red dashed line, just like the R version.

This median value is then used to fill the missing Fare.

In [ ]:
```
# Fare density for Pclass 3 & Embarked S (like R geom_density + vline median)
mask = (full['Pclass'] == 3) & (full['Embarked'] == 'S')
subset = full[mask]
plt.figure(figsize=(8,5))
sns.histplot(subset['Fare'], kde=True, stat='density', color='#99d6ff', alpha=
median_fare = subset['Fare'].median()
plt.axvline(median_fare, color='red', linestyle='--', linewidth=1)
plt.title("Fare density for Pclass=3 & Embarked='S' (vertical line = median)")
plt.xlabel("Fare")
plt.show()

print("Median fare (Pclass=3 & Embarked='S'):", round(median_fare, 2))
```




```
Median fare (Pclass=3 & Embarked='S'): 8.05
```

Fix Missing Fare Using Group Median

After analyzing fare patterns, we ensure that all missing Fare values are properly filled.

✔ For each missing Fare: We look at passengers with the same Pclass and same Embarked port, then compute their median fare, and fill it in.

This keeps the imputed value consistent with what similar passengers paid—

In [ ]:
```python
# Cell 8: Fix any missing Fare by group median (Pclass & Embarked)
if full['Fare'].isnull().sum() > 0:
    missing_fare_idx = full[full['Fare'].isnull()].index
    for idx in missing_fare_idx:
        p = full.loc[idx, 'Pclass']
        e = full.loc[idx, 'Embarked']
        med = full[(full['Pclass']==p) & (full['Embarked']==e)]['Fare'].median
        full.loc[idx, 'Fare'] = med
    print("Filled missing Fare values.")
else:
    print("No missing Fare values to fill.")
```

```
Filled missing Fare values.
```

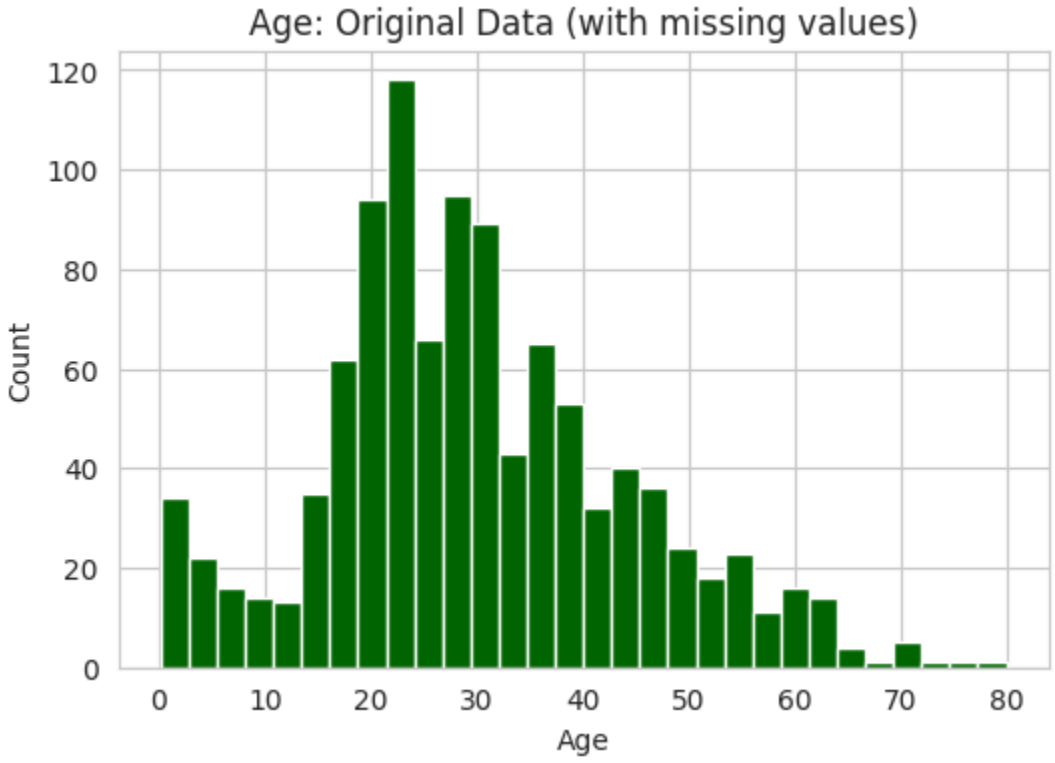
Checking Missing Age Values

➤ Before imputing missing ages, we first: ➤ Count how many Age values are missing ➤ Plot the original Age distribution

➤ This helps us understand: ➤ How incomplete the Age variable is ➤ Whether the distribution looks reasonable ➤ What changes occur after imputation

In [ ]:
```
#  Show missing age count and histogram before imputation
print("Missing Ages before:", full['Age'].isnull().sum())
plt.figure(figsize=(6,4))
full['Age'].hist(bins=30, color='darkgreen')
plt.title("Age: Original Data (with missing values)")
plt.xlabel("Age")
plt.ylabel("Count")
plt.show()
```

```
Missing Ages before: 263
```



Predicting Missing Ages Using Random Forest

Age has many missing values, so instead of filling with a simple mean or median, we build a Random Forest regression model to predict missing ages more

accurately.

✔ Steps performed:

Select useful features (Pclass, Sex, Fare, etc.) to help predict Age.

Encode categorical variables (Sex, Embarked, Title) using LabelEncoder.

Split data into:

Rows with Age → used for training

Rows without Age → used for predicting

Train a RandomForestRegressor on the known ages.

Predict missing ages and fill them back into the dataset.

This matches the R notebook's MICE imputation approach, but uses Random Forest in Python.

In [ ]:
```
#  Prepare features for age prediction
age_vars = ['Age','Pclass','Sex','SibSp','Parch','Fare','Embarked','Title']
age_df = full[age_vars].copy()

# Encode categorical cols for RF
le_age = {}
for c in ['Sex','Embarked','Title']:
    le = LabelEncoder()
    age_df[c] = le.fit_transform(age_df[c].astype(str))
    le_age[c] = le  # store encoders

# Split known / unknown
known_age = age_df[age_df['Age'].notnull()]
unknown_age = age_df[age_df['Age'].isnull()]

X_known = known_age.drop('Age', axis=1)
y_known = known_age['Age']

print("Training RF regressor on", X_known.shape[0], "rows to predict", unknown
rf_age = RandomForestRegressor(n_estimators=200, random_state=129)
rf_age.fit(X_known, y_known)

# Predict
if not unknown_age.empty:
    X_unknown = unknown_age.drop('Age', axis=1)
    pred_ages = rf_age.predict(X_unknown)
    full.loc[full['Age'].isnull(), 'Age'] = pred_ages
    print("Age imputation complete.")
else:
    print("No missing Age values to impute.")
```

```
Training RF regressor on 1046 rows to predict 263 missing ages.
Age imputation complete.
```

Age Imputation Using Random Forest

The Age column has many missing values, so instead of filling them with simple averages, we train a Random Forest regression model to predict missing ages more accurately.

What this code does:

Select useful features that help predict age (Pclass, Sex, Fare, Embarked, Title, etc.)

Convert categorical features to numeric using LabelEncoder (since RF models need numbers)

Split the data into:

Rows with age → for training

Rows missing age → for prediction

Train a RandomForestRegressor on known ages

Predict missing ages and fill them back into the dataset

This produces a smarter, more realistic Age distribution.

In [ ]:
```python
#  Prepare features for age prediction
age_vars = ['Age','Pclass','Sex','SibSp','Parch','Fare','Embarked','Title']
age_df = full[age_vars].copy()

# Encode categorical cols for RF
le_age = {}
for c in ['Sex','Embarked','Title']:
    le = LabelEncoder()
    age_df[c] = le.fit_transform(age_df[c].astype(str))
    le_age[c] = le  # store encoders

# Split known / unknown
known_age = age_df[age_df['Age'].notnull()]
unknown_age = age_df[age_df['Age'].isnull()]

X_known = known_age.drop('Age', axis=1)
y_known = known_age['Age']

print("Training RF regressor on", X_known.shape[0], "rows to predict", unknown
rf_age = RandomForestRegressor(n_estimators=200, random_state=129)
rf_age.fit(X_known, y_known)
```

```
# Predict
if not unknown_age.empty:
    X_unknown = unknown_age.drop('Age', axis=1)
    pred_ages = rf_age.predict(X_unknown)
    full.loc[full['Age'].isnull(), 'Age'] = pred_ages
    print("Age imputation complete.")
else:
    print("No missing Age values to impute.")
```

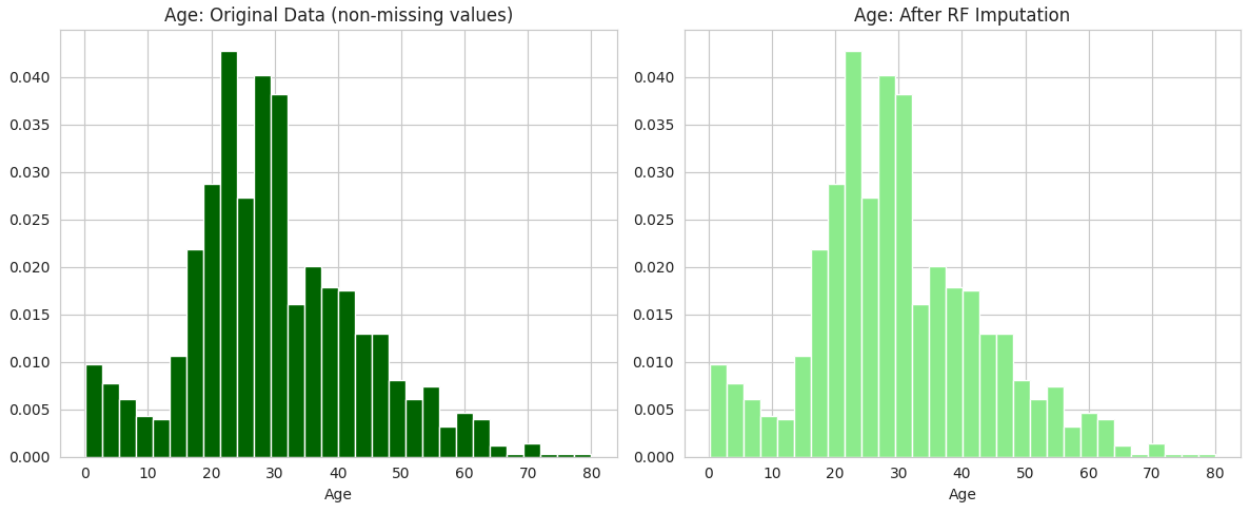
```
Training RF regressor on 1309 rows to predict 0 missing ages.
No missing Age values to impute.
```

Comparing Age Distributions (Before vs After Imputation)

➤ After predicting missing ages using the Random Forest model, we compare: ➤ ✔ Original Age distribution ➤ (using only rows where Age was not missing) ➤ ✔ Imputed Age distribution ➤ (the new, complete Age column after filling predictions)

➤ This side-by-side comparison lets us check: ➤ Whether the imputed ages follow a realistic pattern ➤ Whether the distribution looks similar to the original data ➤ Whether any distortions were introduced

In [ ]:
```
#  Age histograms side-by-side (original vs imputed)
plt.figure(figsize=(12,5))

plt.subplot(1,2,1)
# We can't show original after overwrite, so reload original ages by re-readin
# To simulate R flow properly, we should have stored original Age histogram ea
# We'll show a histogram of ages before imputation using the saved known_age d
plt.hist(known_age['Age'], bins=30, color='darkgreen', density=True)
plt.title("Age: Original Data (non-missing values)")
plt.xlabel("Age")

plt.subplot(1,2,2)
plt.hist(full['Age'], bins=30, color='lightgreen', density=True)
plt.title("Age: After RF Imputation")
plt.xlabel("Age")

plt.tight_layout()
plt.show()
```



In [ ]:
```
print("Missing Ages after imputation:", full['Age'].isnull().sum())
```

```
Missing Ages after imputation: 0
```

Creating Child and Mother Features

Two new features help capture survival patterns based on age and family role:

1. Child

Passengers younger than 18 → "Child"

Others → "Adult" Children often had higher survival chances.

2. Mother

Defined using four conditions (same as the R script): A passenger is labeled “Mother” if she is:

female

older than 18

has at least one child onboard (Parch > 0)

and does not have the title "Miss"

We then print cross-tab tables to compare survival rates for:

Child vs Adult

Mother vs Not Mother

In [ ]:
```
#  Child and Mother
full['Child'] = np.where(full['Age'] < 18, 'Child', 'Adult')

full['Mother'] = 'Not Mother'
# condition: female & Parch > 0 & Age > 18 & Title != 'Miss'
cond = (full['Sex']=='female') & (full['Parch']>0) & (full['Age']>18) & (full[
full.loc[cond, 'Mother'] = 'Mother'

print("Counts (Child vs Survived) in train:")
print(pd.crosstab(full.iloc[:891]['Child'], full.iloc[:891]['Survived']))

print("\nCounts (Mother vs Survived) in train:")
print(pd.crosstab(full.iloc[:891]['Mother'], full.iloc[:891]['Survived']))
```

```
Counts (Child vs Survived) in train:
Survived  0.0  1.0
Child
Adult     487  274
Child      62   68

Counts (Mother vs Survived) in train:
Survived    0.0  1.0
Mother
Mother       16   39
Not Mother  533  303
```

Age Distribution by Survival and Sex

This visualization shows how Age relates to Survival, separately for males and females.

The data is faceted by Sex (male / female)

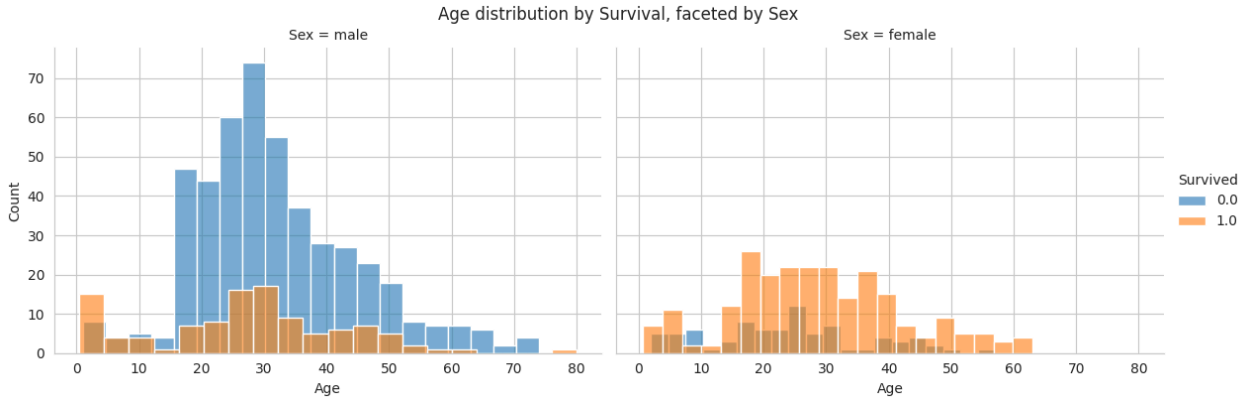
Bars are colored by Survival status

Helps reveal patterns such as:

Higher survival among women

Different survival trends across age groups

In [ ]:
```
#  Age distribution faceted by Sex and colored by Survived (like ggplot facet_
g = sns.FacetGrid(full.iloc[:891], col="Sex", hue="Survived", height=4, aspect
g.map(sns.histplot, "Age", bins=20, alpha=0.6).add_legend()
g.fig.suptitle("Age distribution by Survival, faceted by Sex", y=1.02)
plt.show()
```



In [ ]:
```
factor_vars = ['PassengerId','Pclass','Sex','Embarked','Title','Surname','Fami
# Convert specified columns to 'category' dtype in Python
for col in factor_vars:
    full[col] = full[col].astype('category')
```

Encoding Categorical Variables

Machine learning models like Random Forest require numeric input. Since our dataset contains several categorical features (text values), we convert them into numeric form using Label Encoding.

What this step does:

Ensures the Deck feature exists (extracted from Cabin)

Converts categorical columns such as:

Sex, Embarked, Title

Family, FsizeD

Child, Mother

Stores encoders so labels remain consistent

In [ ]:
```
#  Label encode categorical features across the whole dataset
cat_cols = ['Pclass','Sex','Embarked','Title','Surname','Family','FsizeD','Chi
# Ensure Deck exists (we created Cabin earlier, Deck initial may be missing)
if 'Deck' not in full.columns:
    full['Cabin'] = full['Cabin'].fillna('Unknown')
    full['Deck'] = full['Cabin'].apply(lambda x: x[0])

le_dict = {}
for c in cat_cols:
    le = LabelEncoder()
    full[c] = le.fit_transform(full[c].astype(str))
```

```
    le_dict[c] = le
print("Categorical columns encoded.")
```

```
Categorical columns encoded.
```

➤Random Forest Training & Error Analysis

We now train a Random Forest classifier to predict passenger survival.

What this step does:

Selects the most important engineered features

Trains the Random Forest incrementally using warm_start

Uses Out-of-Bag (OOB) error to evaluate model performance

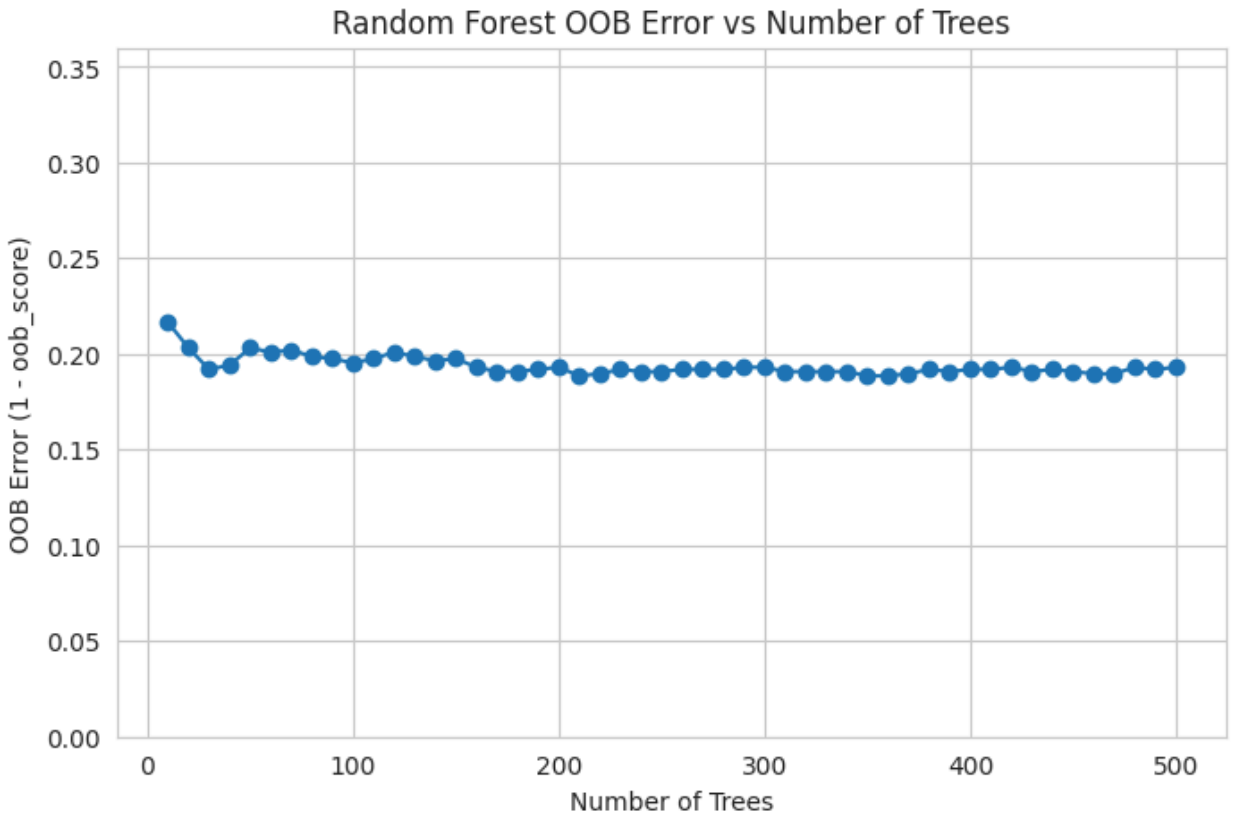
Plots OOB error vs number of trees, similar to the R randomForest error plot

OOB error gives an unbiased estimate of model accuracy without a separate validation set.

In [ ]:
```
#  Train RF with warm_start and record OOB error per step
features = ['Pclass','Sex','Age','SibSp','Parch','Fare','Embarked','Title','Fs
X = train_df[features]
y = train_df['Survived'].astype(int)

# We'll train RF progressively and record OOB error
oob_errors = []
n_trees_range = list(range(10, 501, 10))

# Initialize RF with warm_start
rf_ws = RandomForestClassifier(warm_start=True, oob_score=True, random_state=7

for n_t in n_trees_range:
    rf_ws.n_estimators = n_t
    rf_ws.fit(X, y)
    oob_err = 1 - rf_ws.oob_score_
    oob_errors.append(oob_err)

# Plot OOB error vs n_trees (this is equivalent to plotting rf_model error cur
plt.figure(figsize=(8,5))
plt.plot(n_trees_range, oob_errors, marker='o')
plt.xlabel("Number of Trees")
plt.ylabel("OOB Error (1 - oob_score)")
plt.title("Random Forest OOB Error vs Number of Trees")
plt.ylim(0, 0.36)
plt.grid(True)
plt.show()
```

```
# We'll keep rf_ws as final model with last n_estimators
rf_model = rf_ws
print("Final RF model trained with n_estimators =", rf_ws.n_estimators)
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/ensemble/_forest.py:612: UserWa
rning: Some inputs do not have OOB scores. This probably means too few trees we
re used to compute any reliable OOB estimates.
  warn(
```




```
Final RF model trained with n_estimators = 500
```

Variable Importance

This step shows which features contributed most to predicting survival in the Random Forest model.

Random Forest computes feature importance based on how much each variable reduces impurity

This is the Python equivalent of Mean Decrease in Gini used in the R notebook

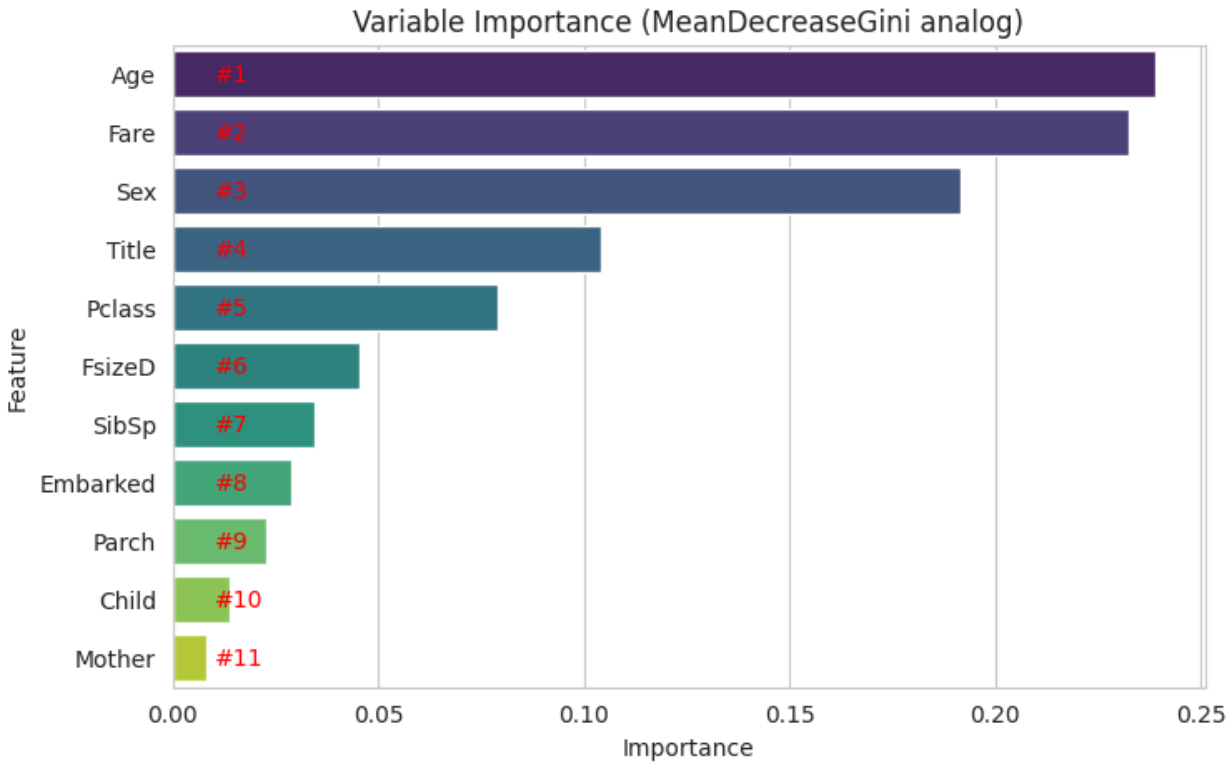
Features are ranked from most important to least important

Rank labels (#1, #2, ...) help interpret relative importance easily

This confirms the impact of engineered features such as Title, Age, and Family-related variables.

In [ ]:
```
#  Variable importance plotting
importances = rf_model.feature_importances_
varimp = pd.DataFrame({'Feature': features, 'Importance': importances}).sort_v

# Add rank (like R's dense_rank)
varimp['Rank'] = ['#'+str(i+1) for i in range(len(varimp))]

plt.figure(figsize=(8,5))
sns.barplot(x='Importance', y='Feature', data=varimp, palette='viridis')
# add rank labels to left of bars
for i, (imp, feat) in enumerate(zip(varimp['Importance'], varimp['Feature'])):
    plt.text(0.01, i, varimp.iloc[i]['Rank'], color='red', fontsize=10, va='ce
plt.title("Variable Importance (MeanDecreaseGini analog)")
plt.show()

display(varimp)
```

```
/tmp/ipython-input-1781104081.py:9: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same e
ffect.

  sns.barplot(x='Importance', y='Feature', data=varimp, palette='viridis')
```



	Feature	Importance	Rank
2	Age	0.238982	#1
5	Fare	0.232482	#2
1	Sex	0.191505	#3
7	Title	0.104303	#4
0	Pclass	0.079075	#5
8	FsizeD	0.045521	#6
3	SibSp	0.034525	#7
6	Embarked	0.028896	#8
4	Parch	0.022676	#9
9	Child	0.013810	#10
10	Mother	0.008224	#11

Final Prediction & Submission File

In this final step, we use the trained Random Forest model to predict survival for passengers in the test dataset.

➤ What this step does:

Selects the same feature set used during training

Predicts Survived (0 or 1) for each test passenger

Creates a submission-style DataFrame with:

PassengerId

Survived

Saves the results to a CSV file (rf_mod_solution_python.csv)

In [ ]:
```
#  Prepare test features and predict
X_test = test_df[features]
predictions = rf_model.predict(X_test)

submission = pd.DataFrame({
    'PassengerId': test_df['PassengerId'].astype(int),
    'Survived': predictions.astype(int)
})
```

```
# Save CSV
submission.to_csv('rf_mod_solution_python.csv', index=False)
print("Saved rf_mod_solution_python.csv (first 10 rows):")
display(submission.head(10))
```

```
Saved rf_mod_solution_python.csv (first 10 rows):
```

	PassengerId	Survived
891	892	0
892	893	0
893	894	0
894	895	1
895	896	0
896	897	0
897	898	0
898	899	0
899	900	1
900	901	0

Conclusion

➤The Titanic dataset was explored using Python in Google Colab, closely following the structure of the original R analysis.

➤Passenger names were used to extract Titles and Surnames, which provided important social and family-related information.

➤A Family Size feature was created, showing that:

➤Single passengers had lower survival rates

➤Passengers in small families (2–4 members) had higher survival chances

➤Very large families showed reduced survival

➤Missing Embarked values were inferred using Fare and Passenger Class distributions.

➤Missing Fare values were filled using the median fare of corresponding Pclass and Embarked groups.

➤Missing Age values were imputed using a Random Forest regressor, producing

realistic age distributions.

➤Additional features such as Child and Mother were engineered, improving survival pattern understanding.

➤A Random Forest classifier was trained using engineered features like Title, Sex, Age, Pclass, Fare, and family-related variables.

➤Out-of-Bag (OOB) error analysis showed stable model performance as the number of trees increased.

➤Feature importance analysis revealed that:

➤Title, Sex, Age, and Pclass were the strongest predictors of survival

➤The final model was used to predict survival on the test dataset and generate a valid submission file.

Overall, this project demonstrates how feature engineering, predictive imputation, and ensemble learning improve model performance and insight.