## Introduction:

The “Student Performance Mathematics dataset” used in this report contains information about student's academic outcomes. It is an area where data analytics, artificial intelligence (AI), and machine learning (ML) techniques can offer transformative solutions. By leveraging data about student's behaviors, study patterns, and performance levels, we can gain deeper insights into the key factors that influence success and create strategies.

Specifically, we employ a Markov Decision Process (MDP) framework to simulate how various student actions such as increasing study time or reducing distractions impact their academic outcomes. The goal is to identify an optimal set of actions that students can take to maximize their performance over time.

## Methodology:

Problem Definition and Data Preparation:

It appears that the dataset is not correctly parsed, all columns are combined into a single column separated by semicolons. We need to split these columns based on the semicolon delimiter and then analyze the dataset properly.

The dataset has now been correctly parsed, with 395 entries and 33 columns.

### Key Attributes:

- **Demographics**: school, sex, age, address, famsize, Pstatus
- **Parental Information**: Medu (mother's education), Fedu (father's education), Mjob (mother's job), Fjob (father's job)
- **Educational Support**: schoolsup (extra educational support), famsup (family support), paid (extra paid classes)
- **Study & Leisure**: traveltime, studytime, activities, freetime, goout
- **Behavior**: Dalc (workday alcohol consumption), Walc (weekend alcohol consumption), health, romantic.
- **Performance**: G1, G2, G3 (grades in three periods)

### Summary Statistics:

- **Age**: The students' average age is 16.7 years, ranging from 15 to 22.
- **Education of Parents**: On a scale from 0 to 4 (where 4 is the highest education level), the average education level for mothers is 2.75 and for fathers is 2.52.
- **Studytime**: The average study time is about 2 hours per week (scale from 1 to 4).

- **Failures**: Most students have no previous failures, with an average failure rate of 0.33.
- **Grades**: The average final grade (G3) is 10.4, on a scale of 0 to 20.

The target variable we want to predict is likely **G3** (the final grade). The task could be to predict student performance based on various features such as family background, study habits.

The data was loaded into Jupyter Notebook and stored in a data frame “df_mat”.
The info function was run on the df_mat data frame which showed that the data frame has 395 (rows) * 32 (columns) which are of types int and object.

```
[409]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 395 entries, 0 to 394
Data columns (total 33 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   school      395 non-null    object
 1   sex         395 non-null    object
 2   age         395 non-null    int64
 3   address     395 non-null    object
 4   famsize     395 non-null    object
 5   Pstatus     395 non-null    object
 6   Medu        395 non-null    int64
 7   Fedu        395 non-null    int64
 8   Mjob        395 non-null    object
 9   Fjob        395 non-null    object
 10  reason      395 non-null    object
 11  guardian    395 non-null    object
 12  traveltime  395 non-null    int64
 13  studytime   395 non-null    int64
 14  failures    395 non-null    int64
 15  schoolsup   395 non-null    object
 16  famsup      395 non-null    object
 17  paid        395 non-null    object
 18  activities  395 non-null    object
 19  nursery     395 non-null    object
 20  higher      395 non-null    object
 21  internet    395 non-null    object
 22  romantic    395 non-null    object
 23  famrel      395 non-null    int64
 24  freetime    395 non-null    int64
 25  goout       395 non-null    int64
 26  Dalc        395 non-null    int64
 27  Walc        395 non-null    int64
 28  health      395 non-null    int64
 29  absences    395 non-null    int64
 30  G1          395 non-null    int64
 31  G2          395 non-null    int64
 32  G3          395 non-null    int64
dtypes: int64(16), object(17)
memory usage: 102.0+ KB
```

The first 5 rows were displayed using head (5). Each column was crosschecked with its type and all the columns are not of the correct type and need encoding for the purpose of Exploratory Data Analysis.

```
[465]: df_mat.head()
```

[465]:

| | school | sex | age | address | famsize | Pstatus | Medu | Fedu | Mjob | Fjob | reason | guardian | traveltime | studytime | failures | schoolsup | famsup | paid | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | GP | F | 18 | U | GT3 | A | 4 | 4 | at_home | teacher | course | mother | 2 | 2 | 0 | yes | no | no | |
| 1 | GP | F | 17 | U | GT3 | T | 1 | 1 | at_home | other | course | father | 1 | 2 | 0 | no | yes | no | |
| 2 | GP | F | 15 | U | LE3 | T | 1 | 1 | at_home | other | other | mother | 1 | 2 | 3 | yes | no | yes | |
| 3 | GP | F | 15 | U | GT3 | T | 4 | 2 | health | services | home | mother | 1 | 3 | 0 | no | yes | yes | |
| 4 | GP | F | 16 | U | GT3 | T | 3 | 3 | other | other | home | father | 1 | 2 | 0 | no | yes | yes | |

There are no missing values in the dataset, which is great.

```
[469]: null_values = df.isnull().sum()
print("Null Values:\n", null_values)
```

```
Null Values:
 school       0
sex           0
age           0
address       0
famsize       0
Pstatus       0
Medu          0
Fedu          0
Mjob          0
Fjob          0
reason        0
guardian      0
traveltime    0
studytime     0
failures      0
schoolsup     0
famsup        0
paid          0
activities    0
nursery       0
higher        0
internet      0
romantic      0
famrel        0
freetime      0
goout         0
Dalc          0
Walc          0
health        0
absences      0
G1            0
G2            0
G3            0
dtype: int64
```

**Encode categorical variables**: We will apply encoding (like one-hot encoding for nominal variables).

```
[471…
# Find categorical variables
categorical_columns = df.select_dtypes(include=['object']).columns

# Display categorical columns
print('Categorical_columns',categorical_columns)
```

```
Categorical_columns Index(['school', 'sex', 'address', 'famsize', 'Pstatus', 'Mjob', 'Fjob',
       'reason', 'guardian', 'schoolsup', 'famsup', 'paid', 'activities',
       'nursery', 'higher', 'internet', 'romantic'],
      dtype='object')
```

```
[473]:
# One-hot encode the categorical columns
df_encode = pd.get_dummies(df, columns=categorical_columns, drop_first=True)
pd.set_option('display.max_columns', None)
df_encode.head()
```

[473]:

| | G3 | school_MS | sex_M | address_U | famsize_LE3 | Pstatus_T | Mjob_health | Mjob_other | Mjob_services | Mjob_teacher | Fjob_health | Fjob_other | Fjob_services |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | False | False | True | False | False | False | False | False | False | False | False | False |
| 5 | 6 | False | False | True | False | True | False | False | False | False | False | True | False |
| 3 | 10 | False | False | True | True | True | False | False | False | False | False | True | False |
| 4 | 15 | False | False | True | False | True | True | False | False | False | False | False | True |
| 1 | 10 | False | False | True | False | True | False | True | False | False | False | True | False |

While doing categorical conversion we had to face an issue, so we must convert it into Boolean

```
[475]: bool_features = df_encode.select_dtypes(include=['boolean']).columns.tolist()
```

```
[477]:
bool_maps = {True:1, False: 0}
for column in bool_features:
    df_encode[column] = df_encode[column].map(bool_maps)

df_encode.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 395 entries, 0 to 394
Data columns (total 42 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   age                395 non-null    int64
 1   Medu               395 non-null    int64
 2   Fedu               395 non-null    int64
 3   traveltime         395 non-null    int64
 4   studytime          395 non-null    int64
 5   failures           395 non-null    int64
 6   famrel             395 non-null    int64
 7   freetime           395 non-null    int64
 8   goout              395 non-null    int64
 9   Dalc               395 non-null    int64
 10  Walc               395 non-null    int64
 11  health             395 non-null    int64
 12  absences           395 non-null    int64
 13  G1                 395 non-null    int64
 14  G2                 395 non-null    int64
 15  G3                 395 non-null    int64
 16  school_MS          395 non-null    int64
 17  sex_M              395 non-null    int64
 18  address_U          395 non-null    int64
```

Calling out correct data by df_encode:

```
[479]: df_encode.head()
```

[479]:

| G2 | G3 | school_MS | sex_M | address_U | famsize_LE3 | Pstatus_T | Mjob_health | Mjob_other | Mjob_services | Mjob_teacher | Fjob_health | Fjob_other | Fjob_servic |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 6 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 5 | 6 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | |
| 8 | 10 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | |
| 14 | 15 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | |
| 10 | 10 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | |

## Summary Statistics

Start by calculating the basic summary statistics for numerical columns, which will give insights of the mean, standard deviation and max.

```
[481]: df.describe()
```

[481]:

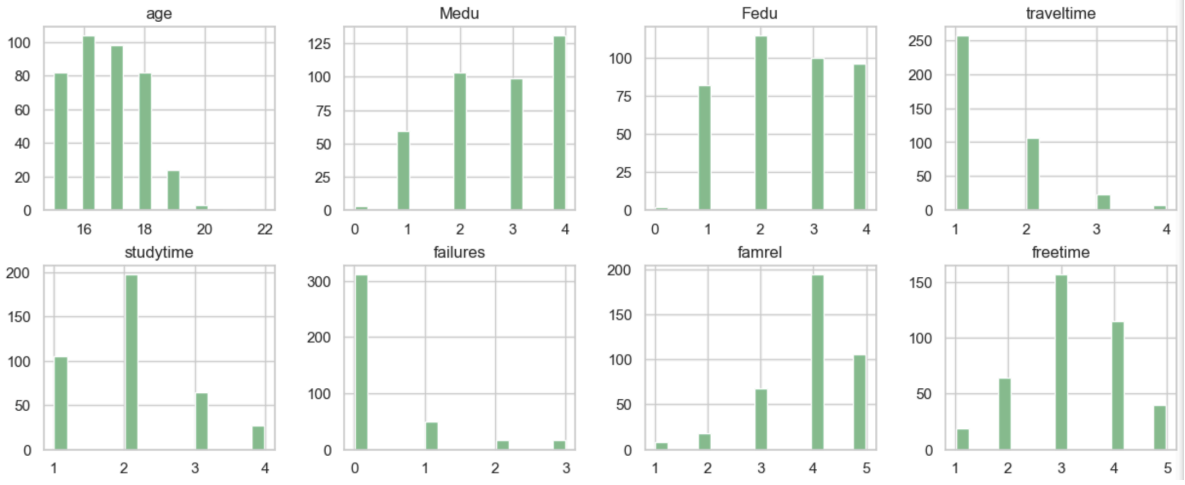
| | age | Medu | Fedu | traveltime | studytime | failures | famrel | freetime | goout | Dalc | Walc | healtl |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 395.000000 | 395.000000 | 395.000000 | 395.000000 | 395.000000 | 395.000000 | 395.000000 | 395.000000 | 395.000000 | 395.000000 | 395.000000 | 395.00000 |
| mean | 16.696203 | 2.749367 | 2.521519 | 1.448101 | 2.035443 | 0.334177 | 3.944304 | 3.235443 | 3.108861 | 1.481013 | 2.291139 | 3.55443 |
| std | 1.276043 | 1.094735 | 1.088201 | 0.697505 | 0.839240 | 0.743651 | 0.896659 | 0.998862 | 1.113278 | 0.890741 | 1.287897 | 1.39030 |
| min | 15.000000 | 0.000000 | 0.000000 | 1.000000 | 1.000000 | 0.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.00000 |
| 25% | 16.000000 | 2.000000 | 2.000000 | 1.000000 | 1.000000 | 0.000000 | 4.000000 | 3.000000 | 2.000000 | 1.000000 | 1.000000 | 3.00000 |
| 50% | 17.000000 | 3.000000 | 2.000000 | 1.000000 | 2.000000 | 0.000000 | 4.000000 | 3.000000 | 3.000000 | 1.000000 | 2.000000 | 4.00000 |
| 75% | 18.000000 | 4.000000 | 3.000000 | 2.000000 | 2.000000 | 0.000000 | 5.000000 | 4.000000 | 4.000000 | 2.000000 | 3.000000 | 5.00000 |
| max | 22.000000 | 4.000000 | 4.000000 | 4.000000 | 4.000000 | 3.000000 | 5.000000 | 5.000000 | 5.000000 | 5.000000 | 5.000000 | 5.00000 |

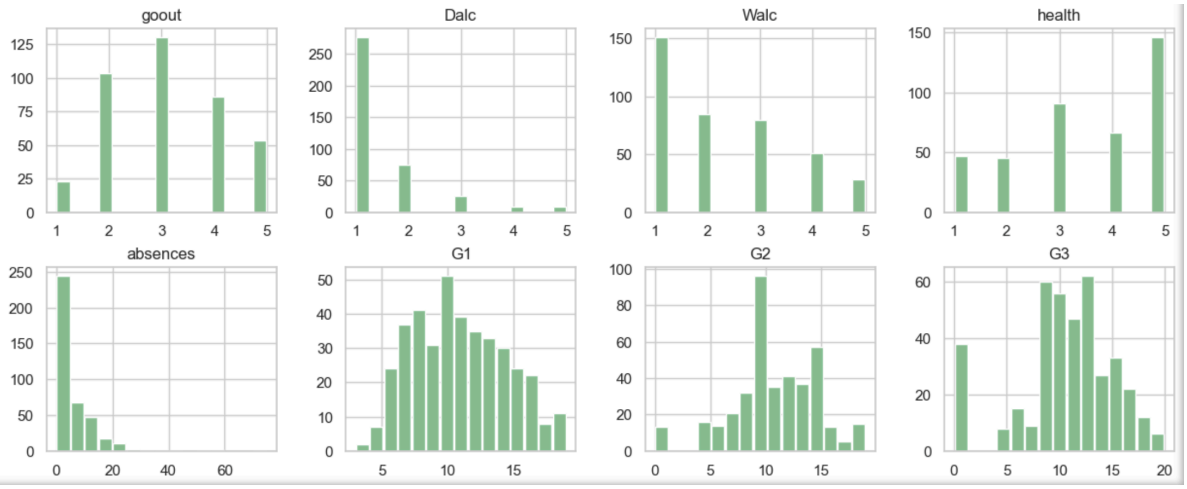
## Distribution of Numerical Features

Plotting the distribution of numerical variables like age, absences, and grades (G1, G2, G3) can show whether these variables are normally distributed, skewed, or have outliers.

```
[431]:
import matplotlib.pyplot as plt
numerical_columns = df.select_dtypes(include=['int64']).columns
print('Numerical_columns',numerical_columns)

# Histogram for all numerical features
df[numerical_columns].hist(bins=15, figsize=(15, 12), color='#86bf91', zorder=2, rwidth=0.9)
plt.show()
```

```
Numerical_columns Index(['age', 'Medu', 'Fedu', 'traveltime', 'studytime', 'failures', 'famrel',
       'freetime', 'goout', 'Dalc', 'Walc', 'health', 'absences', 'G1', 'G2',
       'G3'],
      dtype='object')
```

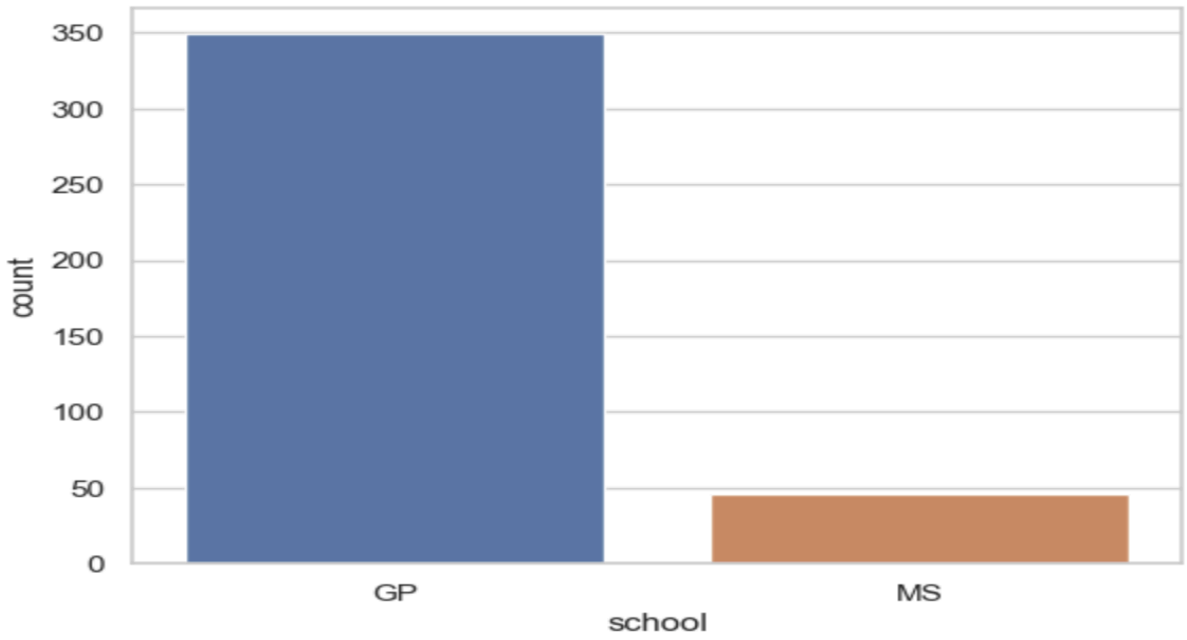


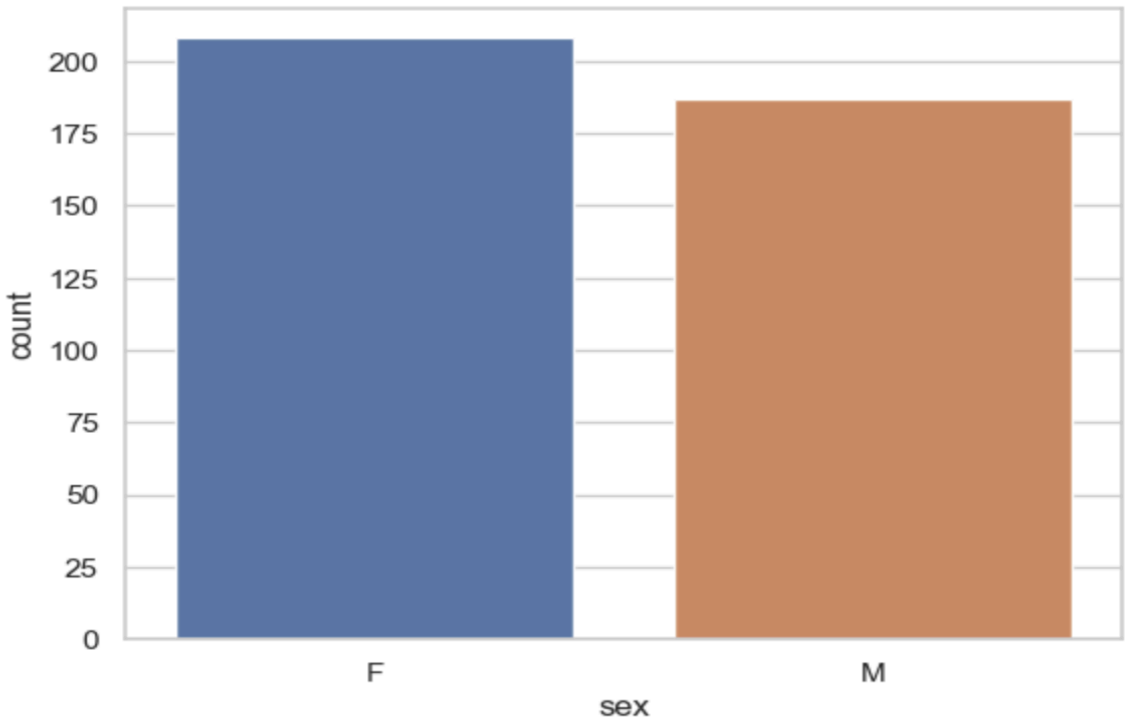



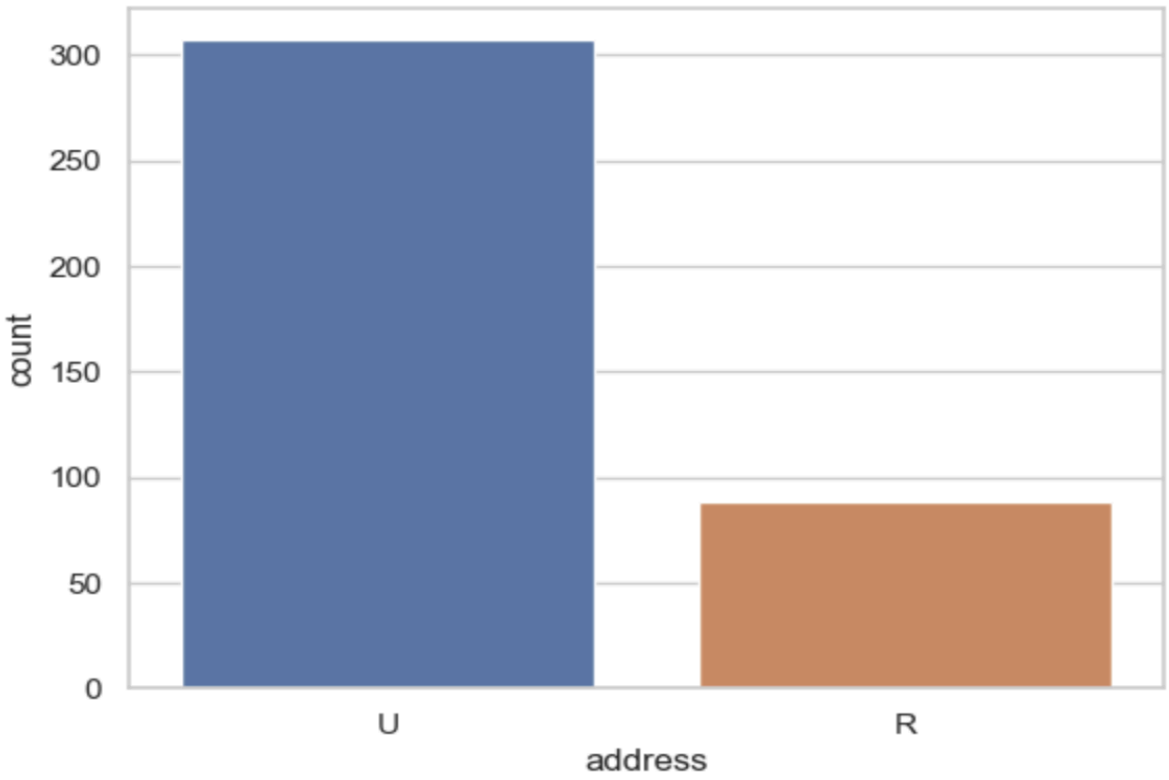
## Distribution of Categorical Variables

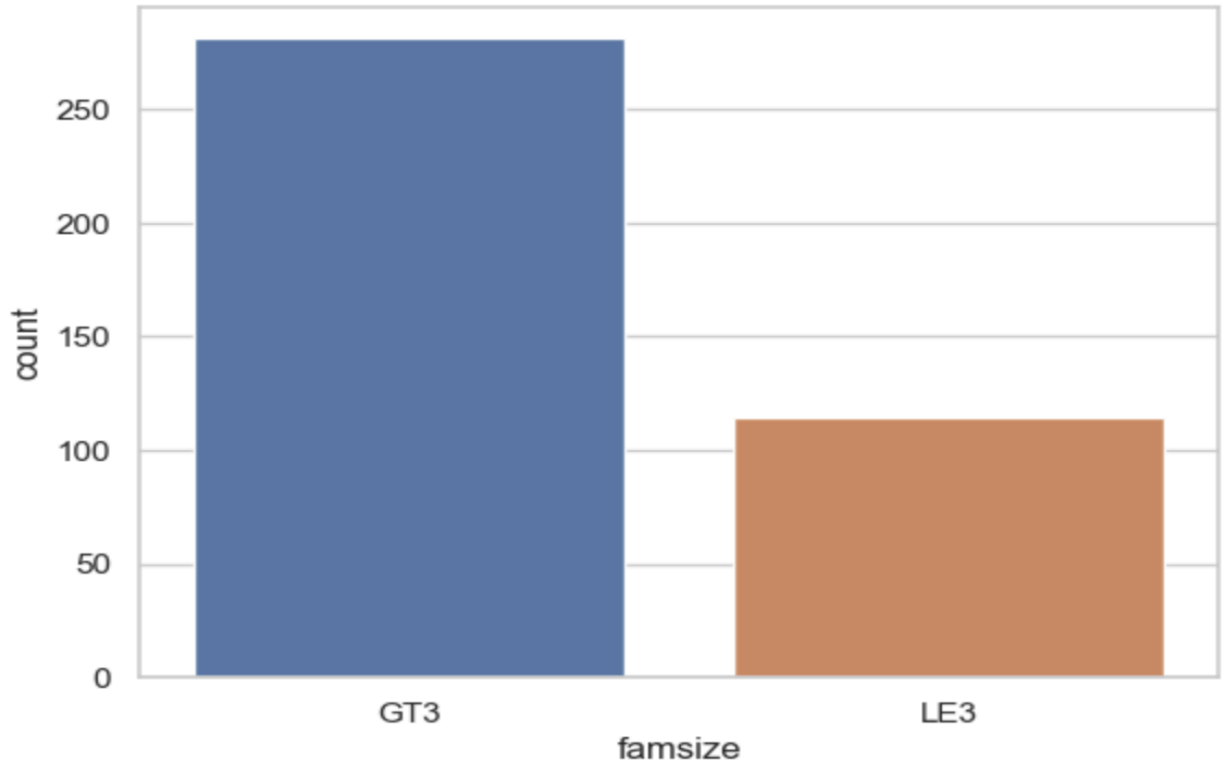
For categorical variables (like sex, school, address, Mjob, etc.), you can use bar plots to visualize the counts of each category.

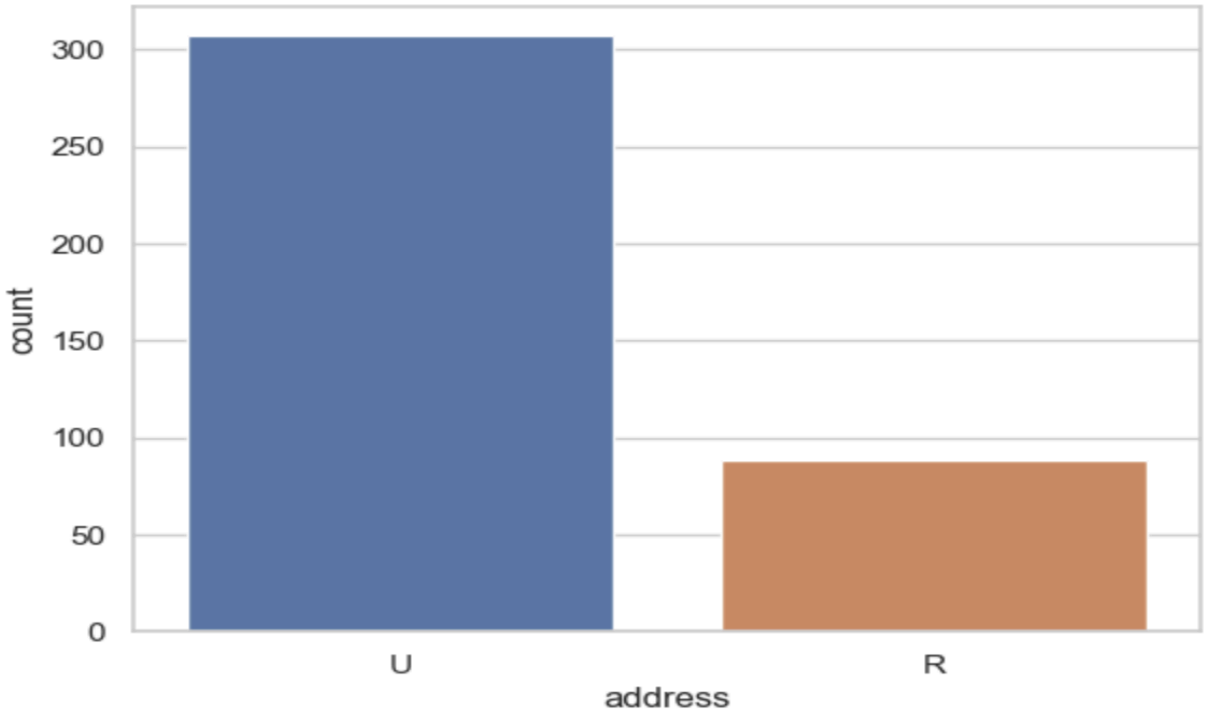
[432]:
```
# Count plots for categorical features
import seaborn as sns
for col in categorical_columns:
    sns.countplot(x=col, data=df, palette='deep')
    plt.show()
```

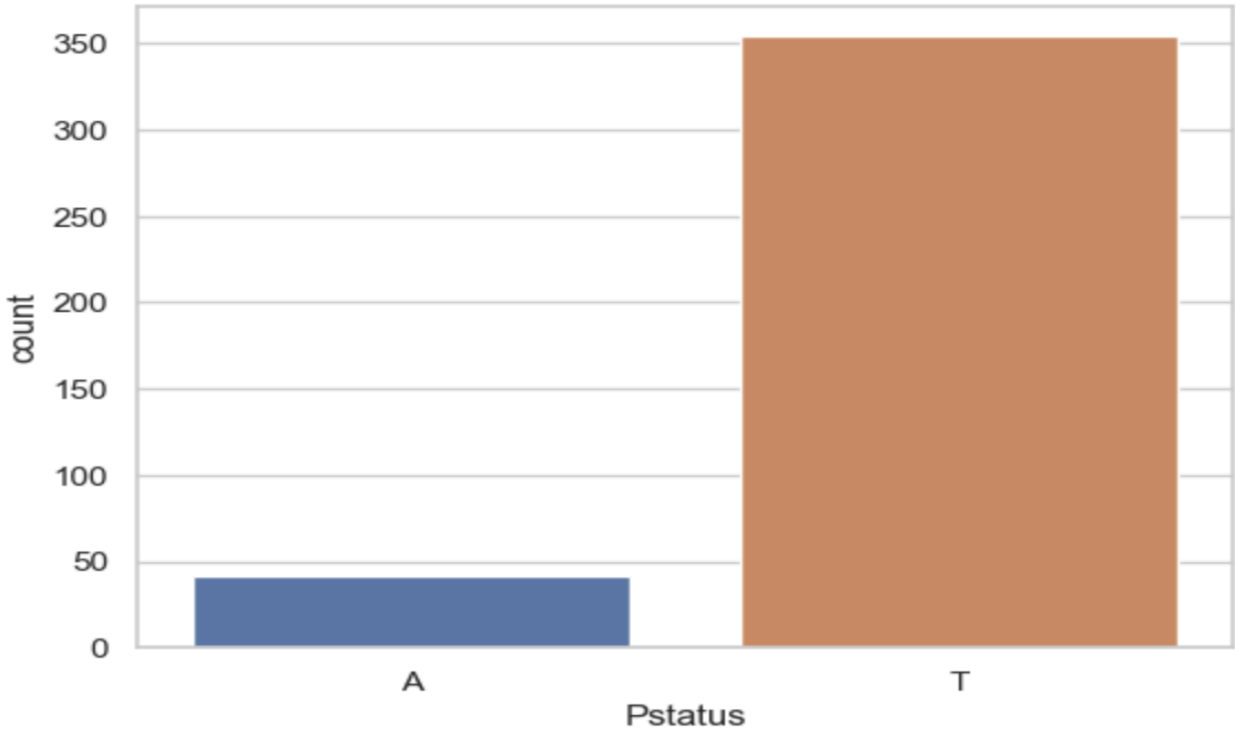


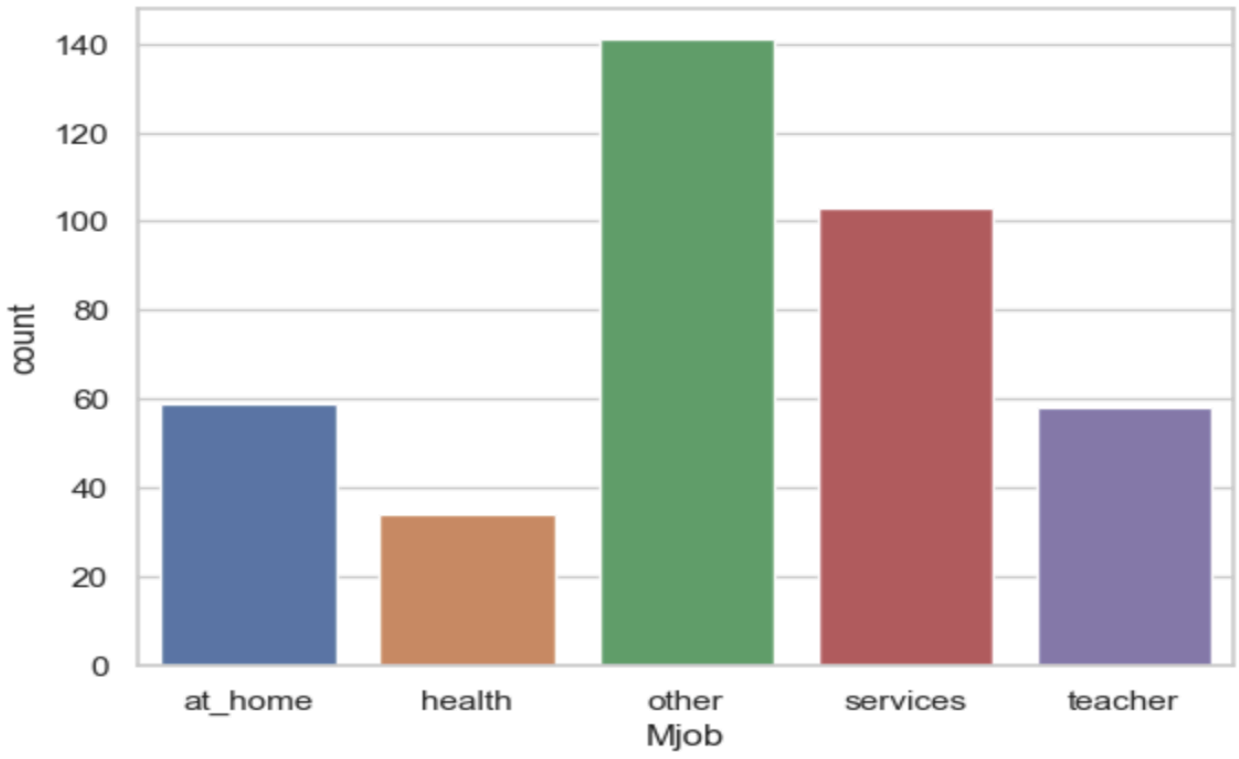

200
175
150
125
100
75
50
25
0
count
F
M
sex

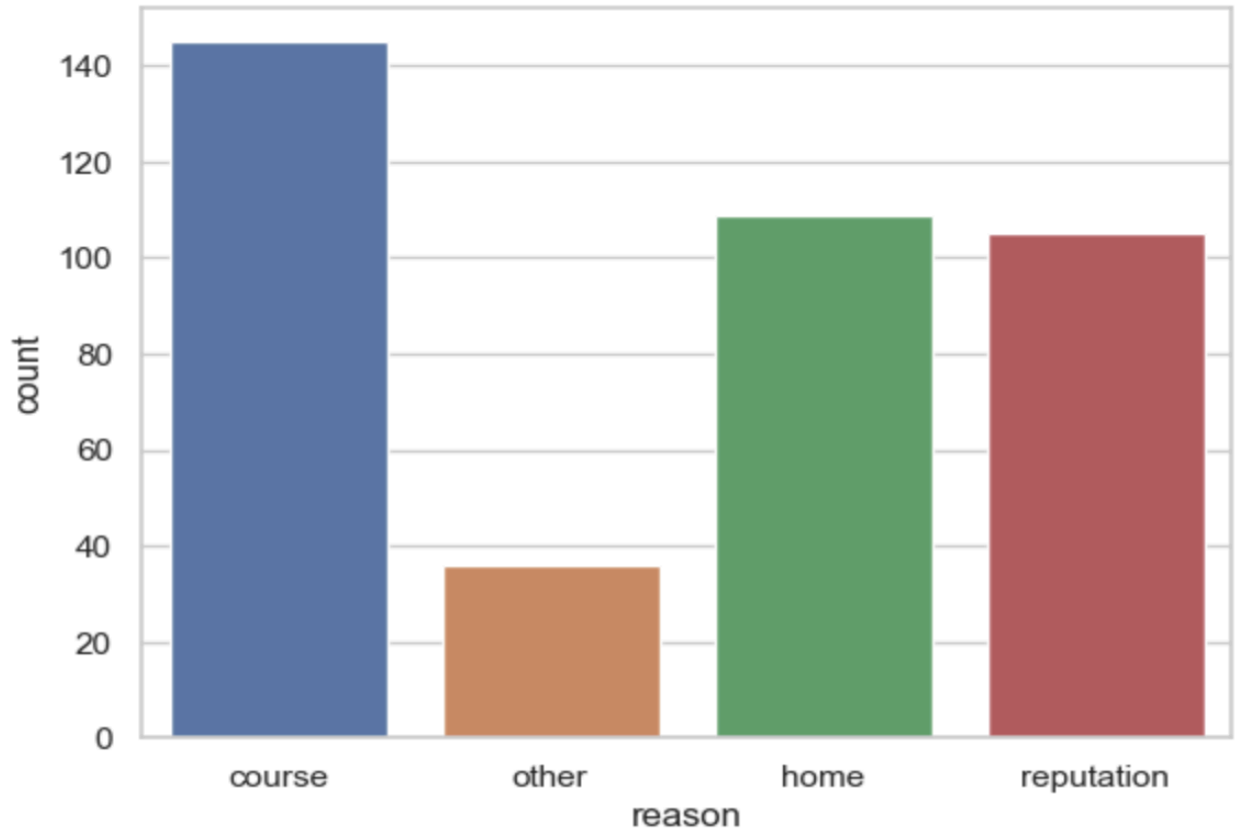

300
250
200
150
100
50
0
count
U
R
address

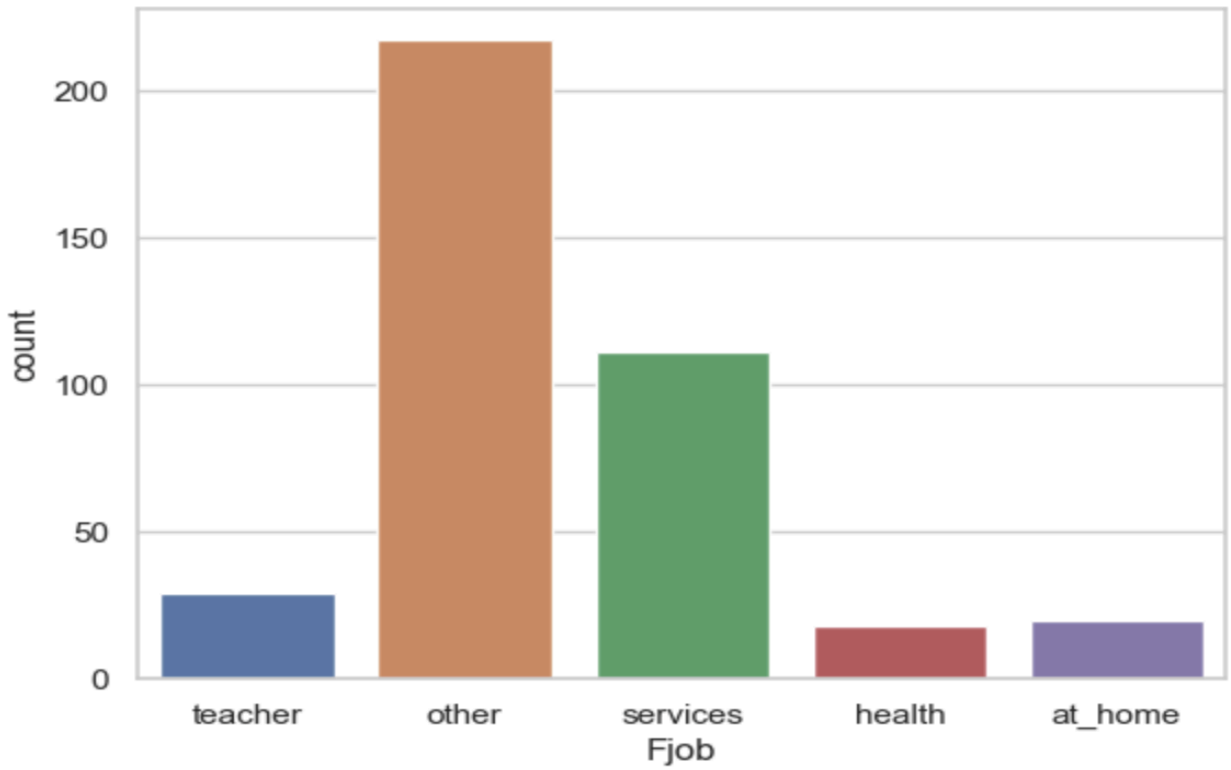

250
200
150
100
50
0
count
GT3
LE3
famsize

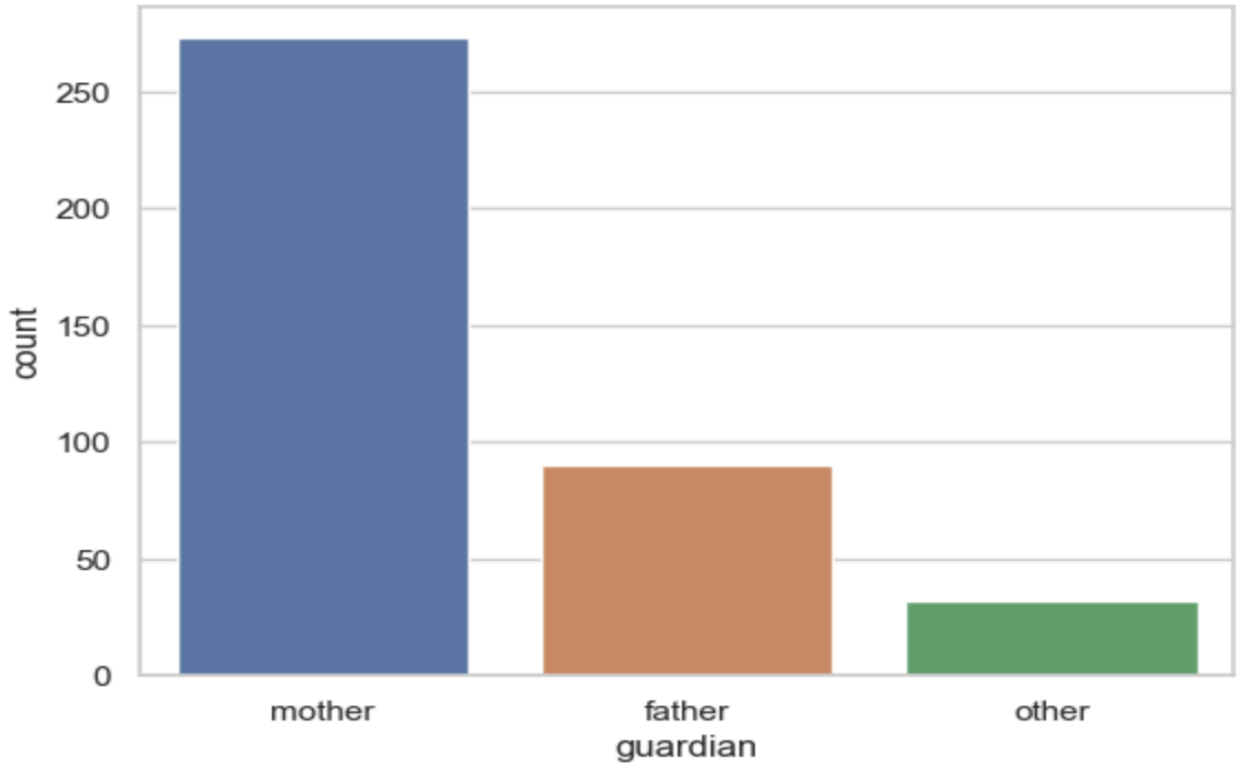

300
250
200
150
100
50
0
count
U
R
address


350
300
250
200
150
100
50
0
count
A
T
Pstatus


140
120
100
80
60
40
20
0
count
at_home
health
other
services
teacher
Mjob


140
120
100
80
60
40
20
0
count
course
other
home
reputation
reason

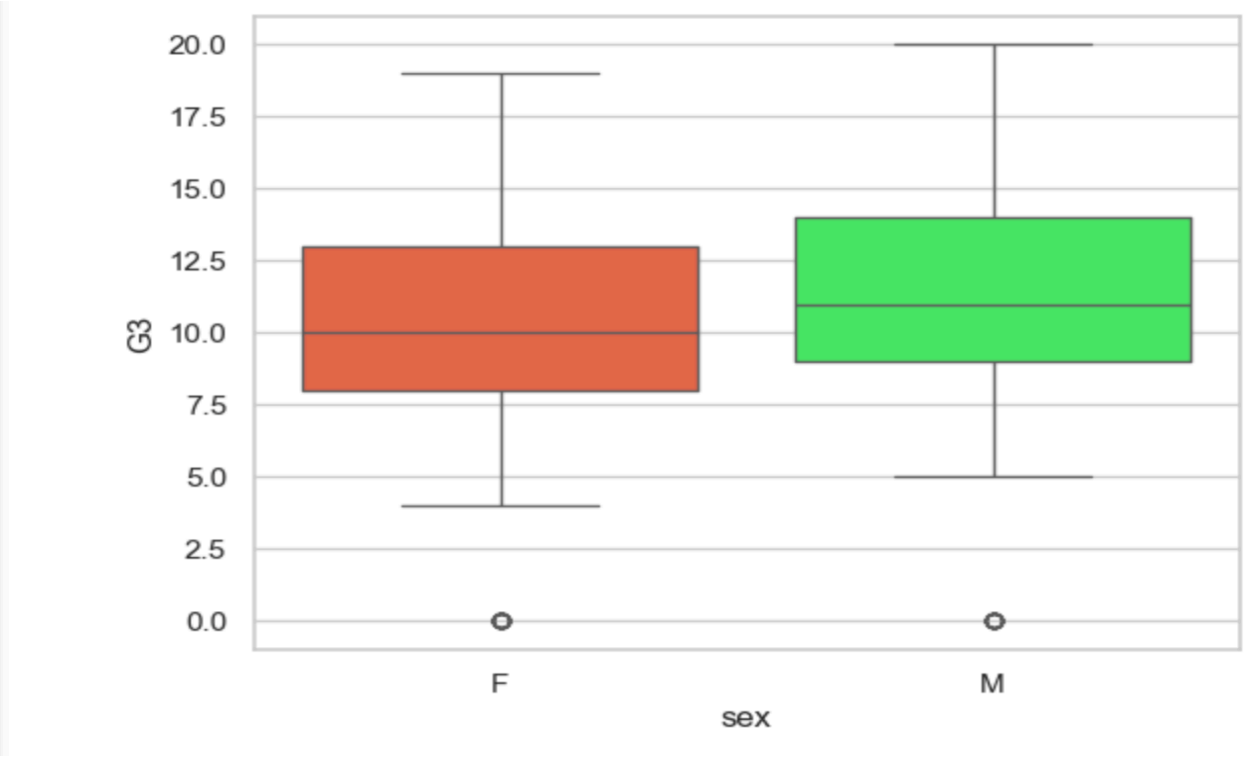

200
150
100
50
0
count
teacher
other
services
health
at_home
Fjob

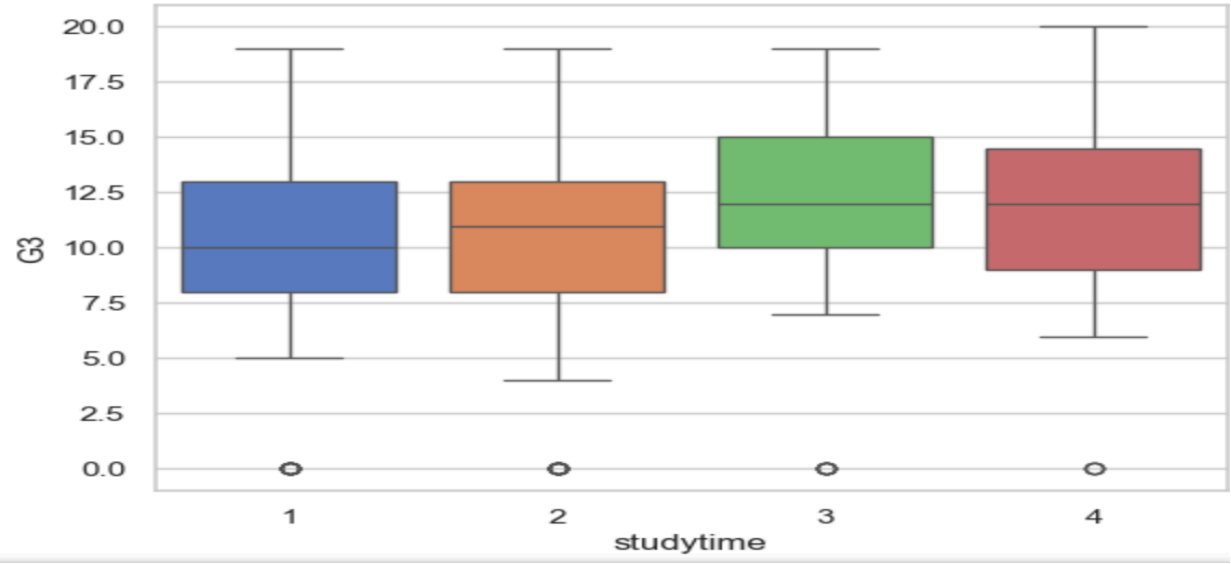

250
200
150
100
50
0
count
mother
father
other
guardian

## Boxplots

Boxplots are useful for identifying outliers and understanding the spread of the data. You can plot boxplots for different variables against the target variable (e.g., G3).

```
[435]: # Boxplot of grades (G3) by categorical features like sex
sns.boxplot(x='sex', y='G3', data=df, palette=['#FF5733', '#33FF57'])
plt.show()

# Boxplot of grades by study time
sns.boxplot(x='studytime', y='G3', data=df,palette='muted')
plt.show()
```



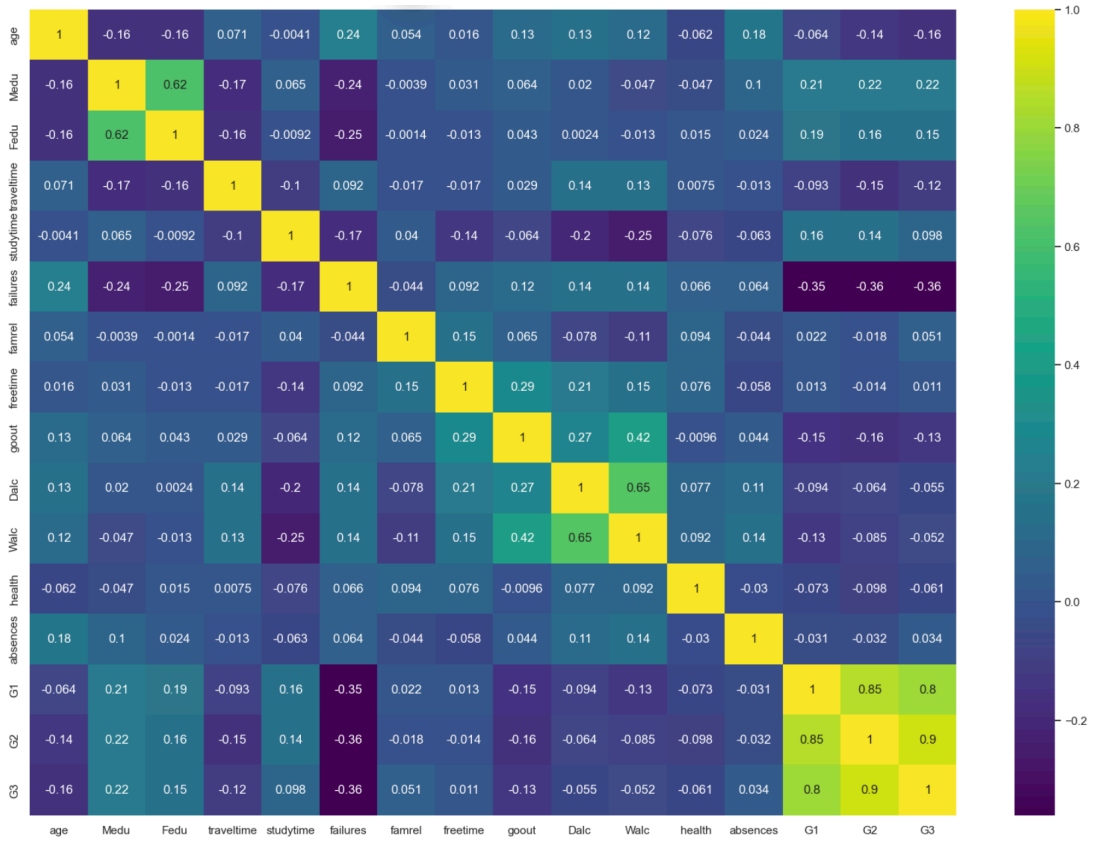



## Correlation Analysis:

- Studytime vs. Final Grade (G3): There is a weak positive correlation (0.10), suggesting that more study time is mildly associated with better grades.
- Failures vs. Final Grade (G3): There is a stronger negative correlation (-0.36), indicating that more failures significantly decrease the final grade.
- Alcohol consumption:
    - Weekday (Dalc) has a slight negative correlation (-0.05) with the final grade.
    - Weekend (Walc) also shows a minor negative correlation (-0.05) with the final grade.

**(PCC):**

[438]:
```
# Correlation matrix
corr_matrix = df.select_dtypes(include=['number']).corr()

# Visualize the correlation matrix
plt.figure(figsize=(20, 14))
sns.heatmap(corr_matrix, annot=True, cmap='viridis')
plt.show()
```




Interpreting the Correlation Coefficients:

- +1: Perfect positive correlation
- 0: No correlation
- -1: Perfect negative correlation

You can focus on the correlation between **G3** (final grade) and other features to explore what influences final grades.

## Hypothesis Testing

You can perform hypothesis testing to check if certain factors (like gender, study time, etc.) significantly affect student performance.

### Example 1: T-test to compare the means of two groups (e.g., does gender affect final grades?)

(*H0*): The mean final grade is the same for male and female students.

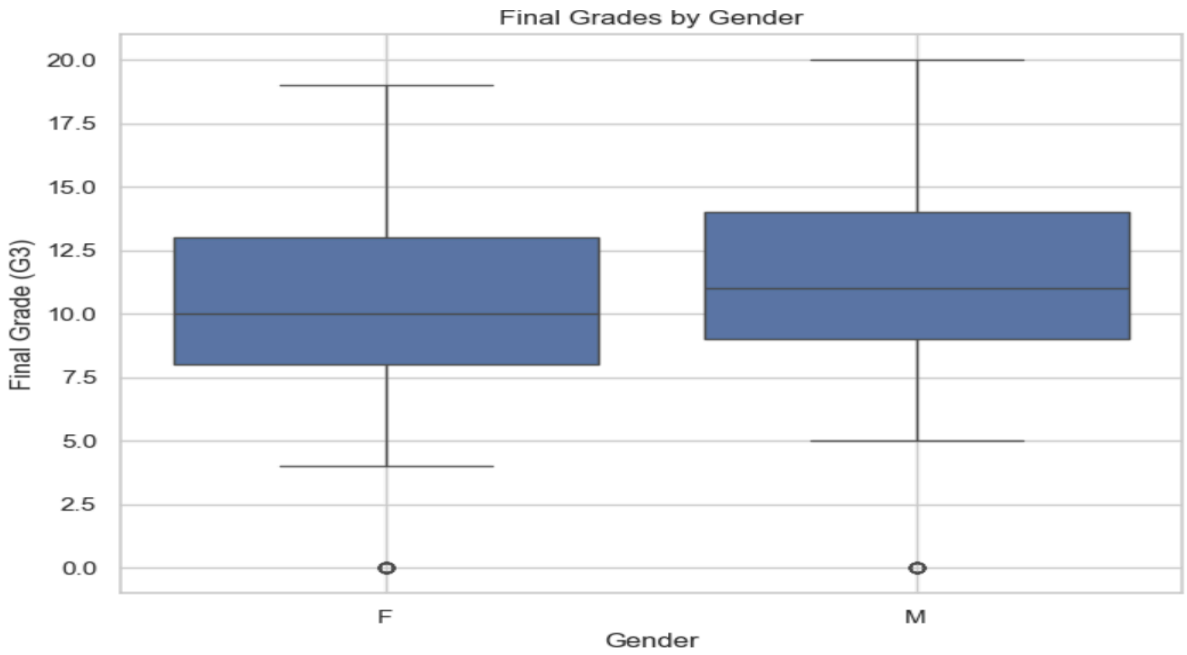
(*H1*): The mean final grade is different between male and female students.

**Interpreting the results**:

- If the p-value is > 0.05, you fail to reject the null hypothesis, no difference.

[441]:
```
from scipy.stats import ttest_ind

# Divide the data based on gender
male_students = df[df['sex'] == 'M']['G3']
female_students = df[df['sex'] == 'F']['G3']

# Set the significance level (alpha)
alpha = 0.05


# Perform t-test
t_stat, p_value = ttest_ind(male_students, female_students)

# Display results
print(f"T-statistic: {t_stat}, P-value: {p_value}")

# Interpret the results
if p < alpha:
    print("There is a significant difference between males and females for the final grades(G3).")
else:
    print("TThere is no significant difference between males and females for the final grades(G3).")


plt.figure(figsize=(8, 6))
sns.boxplot(x='sex', y='G3', data=df)
plt.title('Final Grades by Gender')
plt.xlabel('Gender')
plt.ylabel('Final Grade (G3)')
plt.grid(True)
plt.show()
```

```
T-statistic: 2.061992815503971, P-value: 0.039865332341527636
There is a significant difference between males and females for the final grades(G3).
```




**Example 2: Chi-square Test for Categorical Variables (e.g., does the schoolsup affect final grade categories?)**

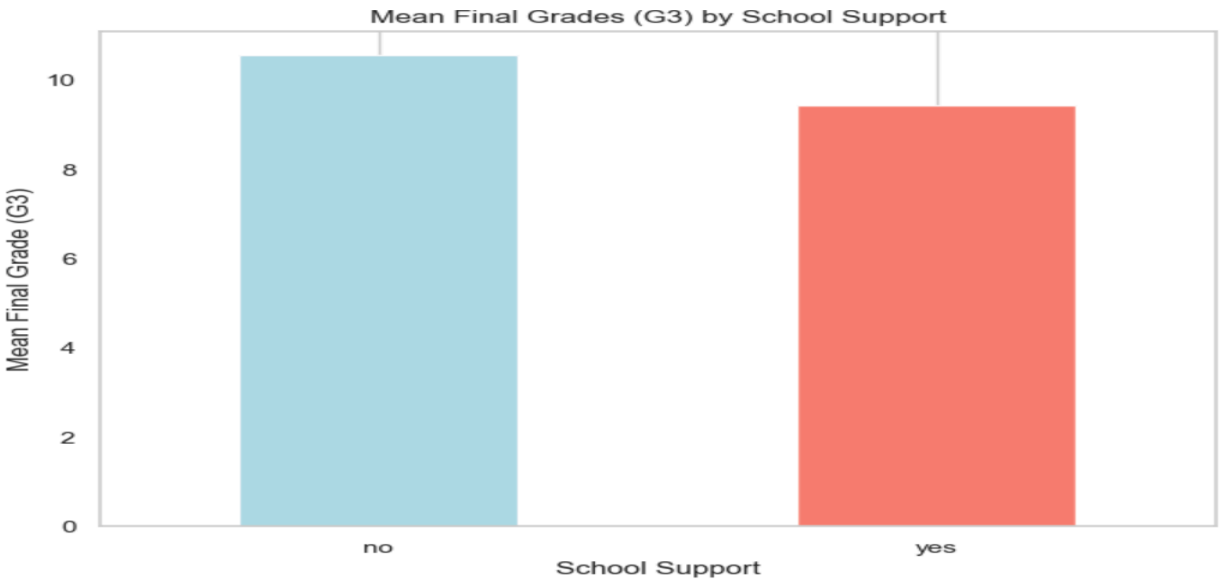
You can use the chi-square test to check if there is an association between two categorical variables.

Interpreting the results:

- If the p-value is < 0.05, there is a relation between receiving school support and final grades.
- If the p-value is > 0.05, there is no relation between.

[443]:
```
from scipy.stats import chi2_contingency

# Create a contingency table
contingency_table = pd.crosstab(df['schoolsup'], df['G3'] > 10)

# Perform chi-square test
chi2, p, dof, expected = chi2_contingency(contingency_table)

# Set the significance level (alpha)
alpha = 0.05

# Display results
print("Chi-Square Statistic:", chi2)
print("P-Value:", p)
print("Degrees of Freedom:", dof)
print("Expected Frequencies Table:")
print(expected)

# Interpret the results
if p < alpha:
    print("There is a significant association between school support(schoolsup) and final grades(G3).")
else:
    print("There is no significant association between school support(schoolsup) and final grades(G3).")



# Calculate mean grades based on school support
mean_grades = df.groupby('schoolsup')['G3'].mean()

# Create a bar plot
plt.figure(figsize=(8, 6))
mean_grades.plot(kind='bar', color=['lightblue', 'salmon'])
plt.title('Mean Final Grades (G3) by School Support')
plt.xlabel('School Support')
plt.ylabel('Mean Final Grade (G3)')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()
```

```
Chi-Square Statistic: 6.50560505010095
P-Value: 0.01075349659103107
Degrees of Freedom: 1
Expected Frequencies Table:
[[161.98481013 182.01518987]
 [ 24.01518987  26.98481013]]
There is a significant association between school support(schoolsup) and final grades(G3).
```



## Probability Analysis

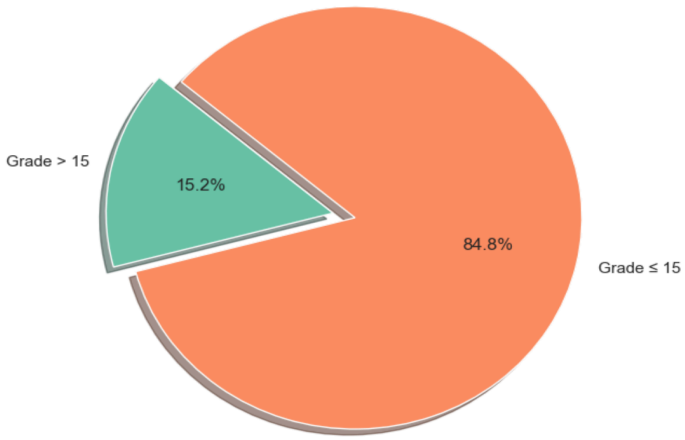
You can also calculate probabilities related to various outcomes, you can compute the probability that a student with a certain level of study time achieves a high final grade.

### Example: Probability of getting high final grade based on study time

```
[445…
# Probability of students with study time less than 2 hours getting a final grade > 15
high_study_time = df[df['studytime'] > 2]
prob_high_grade = len(high_study_time[high_study_time['G3'] > 15]) / len(high_study_time)

[445]: t(f"Probability of getting a high grade with high study time: {prob_high_grade}")

# data for pie chart
labels = ['Grade > 15', 'Grade ≤ 15']
# Count the number of students with grades > 15 and ≤ 15
count_high_grade = len(high_study_time[high_study_time['G3'] > 15])
count_low_grade = len(high_study_time[high_study_time['G3'] <= 15])
sizes = [count_high_grade, count_low_grade]  # Use counts instead of DataFrame
colors = ['#66c2a5', '#fc8d62']
explode = (0.1, 0)  # explode the first slice

# Create a pie chart
plt.figure(figsize=(8, 6))
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%1.1f%%', shadow=True, startangle=140)
plt.title('Probability of High Grades (G3 > 15) with Study Time4 > 2')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```

```
Probability of getting a high grade with high study time: 0.15217391304347827
```

Toggle output scrolling



## Singular Value Decomposition (SVD)

SVD the dataset and analyze the components for target prediction, follow these steps. SVD is a technique that decomposes a matrix into three other matrices, often used for dimensionality reduction and capturing the most important information.

[447]:
```python
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import TruncatedSVD

# Already df_encoded is the preprocessed dataset with one-hot encoded categorical features

# Separate features and

X = df_encode.drop(['G3'], axis=1)  # Features
y = df_encode['G3']  # Target (final grade)

# Standardize the features (important for SVD)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply SVD
n_components = 10  # You can set this based on the number of components you want
svd = TruncatedSVD(n_components=n_components)
X_svd = svd.fit_transform(X_scaled)

# Explained variance ratio for each component
explained_variance = svd.explained_variance_ratio_

# Print the explained variance by each component
for i, var in enumerate(explained_variance):
    print(f"Component {i+1} explains {var:.2%} of the variance.")
```

```
Component 1 explains 9.10% of the variance.
Component 2 explains 6.45% of the variance.
Component 3 explains 5.18% of the variance.
Component 4 explains 4.84% of the variance.
Component 5 explains 4.38% of the variance.
Component 6 explains 4.13% of the variance.
Component 7 explains 3.93% of the variance.
Component 8 explains 3.63% of the variance.
Component 9 explains 3.49% of the variance.
Component 10 explains 3.41% of the variance.
```

## Interpretation of Components:

> The explained variance tells you how much information each component captures. The first component typically captures the most variance in the dataset, with subsequent components capturing progressively less.
> If a few components capture most of the variance (e.g., 90%), this suggests that the dataset can be effectively represented in fewer dimensions, which can help with predictive models.

## Analyzing the Components:

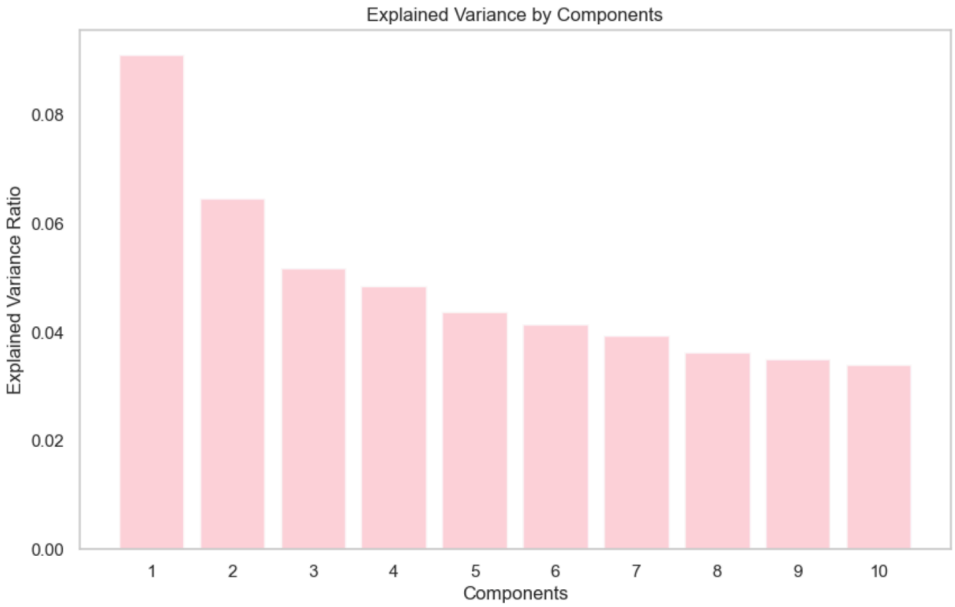
You can analyze the components by looking at their coefficients (loadings) on the original features. These coefficients tell you how much each feature contributes to the component.

```
# Get the components (loadings) for each feature
components = svd.components_

# Create a DataFrame for the loadings
loadings_df = pd.DataFrame(components.T, index=X.columns, columns=[f'Component_{i+1}' for i in range(n_components)])

# Display loadings for the first few components
loadings_df.head()
```

| | Component_1 | Component_2 | Component_3 | Component_4 | Component_5 | Component_6 | Component_7 | Component_8 | Component_9 | Compone |
|---|---|---|---|---|---|---|---|---|---|---|
| **age** | 0.232348 | 0.072844 | 0.246474 | 0.109539 | 0.248493 | 0.235608 | 0.043430 | 0.058619 | -0.002055 | -0.1 |
| **Medu** | -0.332323 | 0.213682 | -0.086817 | -0.116469 | 0.112138 | 0.165715 | 0.117389 | 0.102174 | 0.033415 | -0.05 |
| **Fedu** | -0.301978 | 0.204838 | -0.009428 | -0.126721 | 0.015059 | 0.149065 | 0.187224 | 0.172586 | 0.043320 | 0.03 |
| **traveltime** | 0.179012 | 0.028539 | -0.036667 | 0.097426 | -0.133100 | 0.282715 | -0.063890 | 0.047439 | 0.005451 | -0.03 |
| **studytime** | -0.157647 | -0.235733 | 0.158506 | -0.037522 | 0.100121 | 0.134007 | -0.194080 | 0.013152 | -0.111891 | -0.07 |

Each component represents a combination of original features. The magnitude of the loading (positive or negative) indicates how much that feature contributes to the component.
You can analyze which features are most important in the first few components. eg if the first component has high positive loadings for studytime and negative loadings for goout, it might suggest that students who study more and socialize less perform better in terms of grades.

```python
import matplotlib.pyplot as plt

# Plot explained variance
plt.figure(figsize=(10, 6))
plt.bar(range(1, n_components + 1), explained_variance, alpha=0.7, color='pink')
plt.ylabel('Explained Variance Ratio')
plt.xlabel('Components')
plt.title('Explained Variance by Components')
plt.xticks(range(1, n_components + 1))
plt.grid()
plt.show()
```

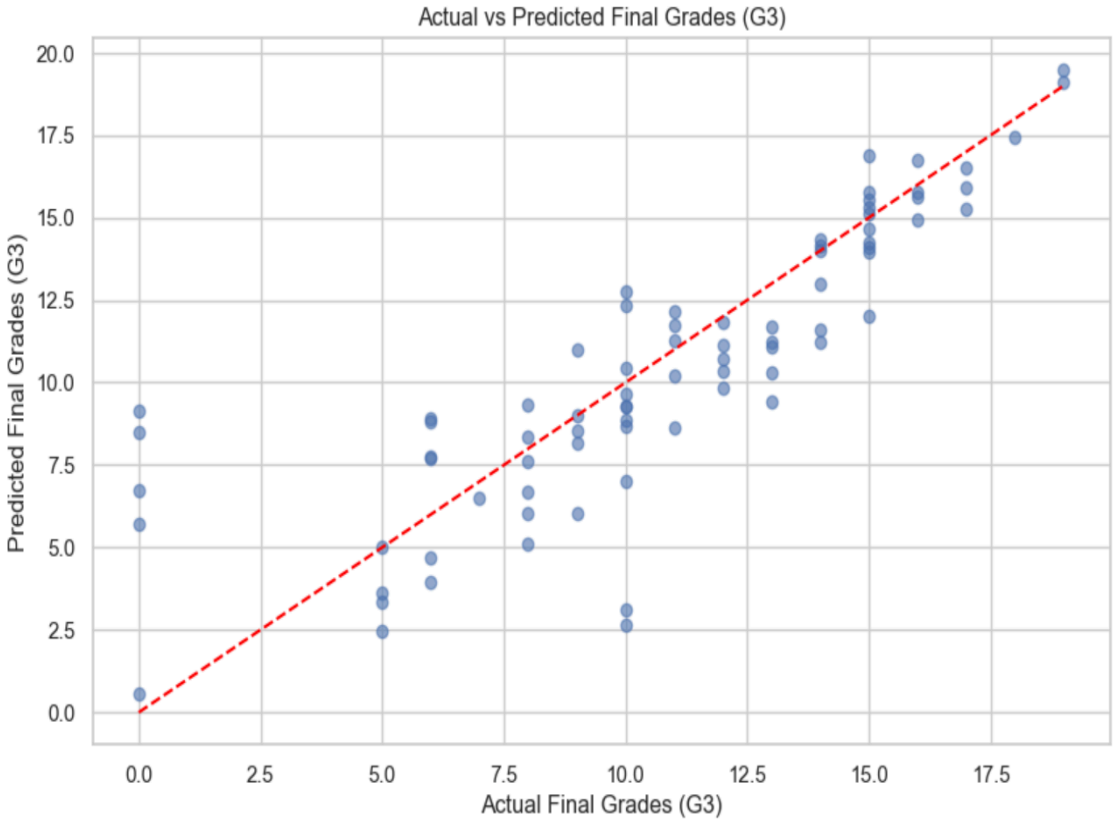


## Using Components for Prediction:

You can use the transformed dataset (SVD components) to build a predictive model for the target variable (final grade). For instance, using a linear regression model on the transformed data:

```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Split the SVD-transformed data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_svd, y, test_size=0.2, random_state=42)

# Fit a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")

# Make predictions
y_pred = model.predict(X_test)

# Calculate R-squared
r2 = r2_score(y_test, y_pred)

# Display the R-squared value
print(f"R-squared: {r2:.4f}")

from scipy.stats import spearmanr
spearman = spearmanr(y_test, y_pred)
print(spearman)

from scipy.stats import pearsonr
pearson = pearsonr(y_test, y_pred)
print(pearson)

# Create a scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', linestyle='--')  # Diagonal line
plt.title('Actual vs Predicted Final Grades (G3)')
plt.xlabel('Actual Final Grades (G3)')
plt.ylabel('Predicted Final Grades (G3)')
plt.grid(True)
plt.show()
```

```
Mean Squared Error: 6.508193828916252
R-squared: 0.6826
SignificanceResult(statistic=0.9005286356373527, pvalue=1.4110929585529583e-29)
PearsonRResult(statistic=0.8332353157316543, pvalue=1.65591021742108e-21)
```




## Insights from the SVD Components:

**Explained Variance:** Components that explain more variance are more important for capturing the underlying structure of the data.

**Loadings:** By looking at the loadings of each component, you can interpret the key factors affecting student performance (e.g., whether time spent studying, family support, or extracurricular activities are most significant).

**Prediction:** Using a few SVD components instead of all original features can improve model efficiency and performance, especially if the first few components explain most of the variance.

## Model Optimization

The objective is to optimize the parameters or hyperparameters of a predictive model to minimize the error (e.g., Mean Squared Error) or maximize accuracy. A common example is optimizing hyperparameters of a machine learning model like **Random Forest**, **SVM.**

Example Scenario:

Let's take **Linear Regression** or **Random Forest** for predicting student grades (G3) and optimize hyperparameters using a grid search or random search.

## Formulate the Optimization Problem

Optimization Goal:

- Model: Linear Regression / Random Forest (you can replace this with any model of choice).
- Optimization Method: Gradient descent or Grid search for hyperparameter tuning.

## Optimization Approach:

**Gradient Descent for Linear Regression:** For Linear Regression, you can implement **gradient descent** to optimize the parameters (weights) to minimize the cost function (MSE).

```
•[455…
```

```python
import numpy as np

# Gradient Descent for Linear Regression
def gradient_descent(X, y, learning_rate=0.01, epochs=1000):
    m, n = X.shape
    theta = np.zeros(n)
    cost_history = []

    for epoch in range(epochs):
        # Prediction
        y_pred = X.dot(theta)
        error = y_pred - y

        # Compute gradients
        gradients = X.T.dot(error) / m

        # Update parameters
        theta -= learning_rate * gradients

        # Compute and store cost (MSE)
        cost = (1 / (2 * m)) * np.sum(error**2)
        cost_history.append(cost)

    return theta, cost_history

# Example usage:
X_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]
theta_opt, cost_hist = gradient_descent(X_b, y_train, learning_rate=0.01, epochs=1000)

# Optimal parameters (theta)
print("Optimized parameters:", theta_opt)
```

```
Optimized parameters: [10.35564079 -1.27775908  0.14720086  0.07989541  1.86253542  0.57883003
 -0.46049146 -0.4473704  -0.31090476  0.51103261  0.15338489]
```

## Hyperparameter Optimization

## Random Search for Faster Hyperparameter Tuning

**Random Search**, which samples a fixed number of parameter settings from the specified hyperparameter space and can be computationally cheaper compared to grid search.

```
from sklearn.model_selection import RandomizedSearchCV

# Define the Random Search
random_search = RandomizedSearchCV(estimator=rf, param_distributions=param_grid, n_iter=10, cv=3, scoring='neg_mean_squared_error', random_s

# Fit the random search to the data
random_search.fit(X_train, y_train)

# Best parameters and the best score
best_params_random = random_search.best_params_
best_score_random = random_search.best_score_

print(f"Best Parameters from Random Search: {best_params_random}")
print(f"Best Cross-Validated Score (MSE) from Random Search: {-best_score_random}")
```

```
Best Parameters from Random Search: {'n_estimators': 100, 'min_samples_split': 5, 'min_samples_leaf': 2, 'max_depth': 10}
Best Cross-Validated Score (MSE) from Random Search: 8.725319984179034
```

After finding the best hyperparameters, evaluate the optimized model on the test set.

```
from sklearn.metrics import mean_squared_error

# Predict using the best estimator from grid search
y_pred = grid_search.best_estimator_.predict(X_test)

# Compute the Mean Squared Error
mse_test = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Set: {mse_test}")
```

```
Mean Squared Error on Test Set: 8.422668194549399
```

A **MDP** is a mathematical framework used to model decision-making processes where outcomes are partly random and partly under the control of a decision-maker. In the context of optimizing students' academic outcomes, we can model students' actions (such as study time, extracurricular activities, etc.) and transitions between performance levels (e.g., improving or degrading academic performance).

**MDP for Student Performance Optimization**

***State (S): Represents different levels of student academic performance (e.g., grade bands).***

We can simplify it into three performance levels:

- **S1**: Low performance (G3 < 10)
- **S2**: Medium performance (10 ≤ G3 < 15)
- **S3**: High performance (G3 ≥ 15)

***Actions (A): Represents actions students can take to influence their performance.***

The possible actions are:

**A1**: Increase study (studying more hours per week).
**A2**: Reduce distractions (e.g., decrease socializing time or screen time).
**A3**: Engage in extracurricular activities

**Transition Probabilities (P): Defines the likelihood of moving from one state to another given an action. These probabilities reflect the uncertainty involved in how actions affect academic performance.**

For simplicity, the transition probabilities might be something like:

**P (S2 | S1, A1)**: Probability low to medium performance by increasing study time.
**P (S3 | S2, A2)**: Probability medium to high performance by balancing extracurricular activities and study.
**P (S1 | S3, A3)**: Probability of falling back to low performance due to increased distractions despite initially being in high performance.
**Rewards (R): Represents the reward associated with transitioning between performance levels. The rewards can be designed to encourage actions that improve academic outcomes:**

- **R (S1, A1) → S2**: Positive reward from low to medium performance.

- **R (S2, A2) → S3**: Higher reward  from medium to high performance.
- **R (S3, A3) → S1**: Negative reward for dropping back to low performance from high performance.

```python
# Define states, actions, and transition probabilities
states = ['Low', 'Average', 'High']
actions = ['StudyMore','DecreaseAbsences', 'Extra Activities']

# Define rewards matrix
rewards = {
    ('Low', 'StudyMore', 'Average'): 10,
    ('Low', 'Extra Activities', 'Average'): 5,
    ('Average', 'StudyMore', 'High'): 20,
    ('Average', 'DecreaseAbsences', 'High'): 15,
    ('High', 'Extra Activities', 'Low'): -15
}

# Transition probabilities
transition_probs = {
    ('Low', 'StudyMore', 'Average'): 0.7,
    ('Low', 'Extra Activities', 'Average'): 0.5,
    ('Average', 'StudyMore', 'High'): 0.6,
    ('Average', 'DecreaseAbsences', 'High'): 0.5,
    ('High', 'Extra Activities', 'Low'): 0.3
}

# Initialize value function
values = {state: 0 for state in states}
gamma = 0.9

# Value iteration
for _ in range(1000):
    new_values = values.copy()
    for state in states:
        new_values[state] = max(
            sum(
                transition_probs.get((state, action, next_state), 0) *
                (rewards.get((state, action, next_state), 0) + gamma * values[next_state])
                for next_state in states
            )
            for action in actions
        )
    values = new_values

# Print the optimal values
print("Optimal Values for each state:", values)

optimal_policy = {}
for state in states:
    optimal_action = max(actions, key=lambda action: sum(
        transition_probs.get((state, action, next_state), 0) *
        (rewards.get((state, action, next_state), 0) + gamma * values[next_state])
        for next_state in states
    ))
    optimal_policy[state] = optimal_action

# Print the optimal policy
print("Optimal Policy for each state:", optimal_policy)
```

```
Optimal Values for each state: {'Low': 14.559999999999999, 'Average': 12.0, 'High': 0.0}
Optimal Policy for each state: {'Low': 'StudyMore', 'Average': 'StudyMore', 'High': 'StudyMore'}
```

## Optimization Objective

The objective of the MDP is to find the **optimal policy** (i.e., the best set of actions for each state) that maximizes the student's long-term expected rewards. In this case, this would involve finding the actions (e.g., studying more or reducing distractions) that are most likely to result in an improved academic performance over time.

## Conclusion

This report highlights the potential of using AI techniques application of such AI-driven optimization methods in education can offer personalized guidance, helping students to make data-informed decisions and ultimately leading to better academic performance. This work also demonstrates the broader potential for AI to be leveraged in other educational settings, providing adaptive learning and decision-making frameworks that enhance student success. Future work could explore more sophisticated models, additional factors influencing academic performance, and real-time implementation in learning platforms.

Word Count

2994

**References**

Brownlee, Jason. (2020). "One-Hot Encodings for Categorical Data". Machinelearningmastery. https://machinelearningmastery.com/one-hot-encoding-for-categorical-data//

Hyperparameter tuning https://www.geeksforgeeks.org/hyperparameter-tuning/

MDP
Lecture 12 – Foundation of AI. https://canvas.qub.ac.uk/courses/28565/pages/learning-materials-lecture-slides-and-videos-12?module_item_id=1338216

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder, StandardScaler
import matplotlib.pyplot as plt

df_mat= pd.read_csv('student-mat.csv', delimiter = ';')
```

```
df_mat.head()
df_mat.info()
null_values = df.isnull().sum()
print("Null Values:\n", null_values)
# Find categorical variables
categorical_columns = df.select_dtypes(include=['object']).columns

# Display categorical columns
print('Categorical_columns',categorical_columns)
# One-hot encode the categorical columns
df_encode = pd.get_dummies(df, columns=categorical_columns, drop_first=True)
pd.set_option('display.max_columns', None)
df_encode.head()

bool_features = df_encode.select_dtypes(include=['boolean']).columns.tolist()
bool_maps = {True:1, False: 0}
for column in bool_features:
    df_encode[column] = df_encode[column].map(bool_maps)

df_encode.info()
df_encode.head()
df.describe()



numerical_columns = df.select_dtypes(include=['int64']).columns
print('Numerical_columns',numerical_columns)

# Histogram for all numerical features
df[numerical_columns].hist(bins=15, figsize=(15, 12), color='#86bf91', zorder=2, rwidth=0.9)
plt.show()



# Count plots for categorical features
import seaborn as sns
for col in categorical_columns:
    sns.countplot(x=col, data=df, palette='deep')
```

```
    plt.show()

# Boxplot of grades (G3) by categorical features like sex
sns.boxplot(x='sex', y='G3', data=df, palette=['#FF5733', '#33FF57'])
plt.show()

# Boxplot of grades by study time
sns.boxplot(x='studytime', y='G3', data=df,palette='muted')
plt.show()

# Correlation matrix
corr_matrix = df.select_dtypes(include=['number']).corr()

# Visualize the correlation matrix
plt.figure(figsize=(20, 14))
sns.heatmap(corr_matrix, annot=True, cmap='viridis')
plt.show()

from scipy.stats import ttest_ind

# Divide the data based on gender
male_students = df[df['sex'] == 'M']['G3']
female_students = df[df['sex'] == 'F']['G3']

# Set the significance level (alpha)
alpha = 0.05



# Perform t-test
t_stat, p_value = ttest_ind(male_students, female_students)

# Display results
print(f"T-statistic: {t_stat}, P-value: {p_value}")

# Interpret the results
if p < alpha:
    print("There is a significant difference between males and females for the final grades(G3).")
```

```
else:
    print("TThere is no significant difference between males and females for the final
grades(G3).")



plt.figure(figsize=(8, 6))
sns.boxplot(x='sex', y='G3', data=df)
plt.title('Final Grades by Gender')
plt.xlabel('Gender')
plt.ylabel('Final Grade (G3)')
plt.grid(True)
plt.show()



from scipy.stats import chi2_contingency

# Create a contingency table
contingency_table = pd.crosstab(df['schoolsup'], df['G3'] > 10)

# Perform chi-square test
chi2, p, dof, expected = chi2_contingency(contingency_table)

# Set the significance level (alpha)
alpha = 0.05

# Display results
print("Chi-Square Statistic:", chi2)
print("P-Value:", p)
print("Degrees of Freedom:", dof)
print("Expected Frequencies Table:")
print(expected)

# Interpret the results
if p < alpha:
    print("There is a significant association between school support(schoolsup) and final
grades(G3).")
else:
```

```
    print("There is no significant association between school support(schoolsup) and final grades(G3).")



# Calculate mean grades based on school support
mean_grades = df.groupby('schoolsup')['G3'].mean()

# Create a bar plot
plt.figure(figsize=(8, 6))
mean_grades.plot(kind='bar', color=['lightblue', 'salmon'])
plt.title('Mean Final Grades (G3) by School Support')
plt.xlabel('School Support')
plt.ylabel('Mean Final Grade (G3)')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()

# Probability of students with study time less than 2 hours getting a final grade > 15
high_study_time = df[df['studytime'] > 2]
prob_high_grade = len(high_study_time[high_study_time['G3'] > 15]) / len(high_study_time)

print(f"Probability of getting a high grade with high study time: {prob_high_grade}")

# data for pie chart
labels = ['Grade > 15', 'Grade ≤ 15']
# Count the number of students with grades > 15 and ≤ 15
count_high_grade = len(high_study_time[high_study_time['G3'] > 15])
count_low_grade = len(high_study_time[high_study_time['G3'] <= 15])
sizes = [count_high_grade, count_low_grade]  # Use counts instead of DataFrame
colors = ['#66c2a5', '#fc8d62']
explode = (0.1, 0)  # explode the first slice

# Create a pie chart
plt.figure(figsize=(8, 6))
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%1.1f%%', shadow=True, startangle=140)
```

```
plt.title('Probability of High Grades (G3 > 15) with Study Time4 > 2')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import TruncatedSVD


# Already df_encoded is the preprocessed dataset with one-hot encoded categorical features

# Separate features and

X = df_encode.drop(['G3'], axis=1)  # Features
y = df_encode['G3']  # Target (final grade)

# Standardize the features (important for SVD)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply SVD
n_components = 10  # You can set this based on the number of components you want
svd = TruncatedSVD(n_components=n_components)
X_svd = svd.fit_transform(X_scaled)

# Explained variance ratio for each component
explained_variance = svd.explained_variance_ratio_

# Print the explained variance by each component
for i, var in enumerate(explained_variance):
    print(f"Component {i+1} explains {var:.2%} of the variance.")
# Get the components (loadings) for each feature
components = svd.components_

# Create a DataFrame for the loadings
loadings_df = pd.DataFrame(components.T, index=X.columns, columns=[f'Component_{i+1}'
for i in range(n_components)])
```

```
# Display loadings for the first few components
loadings_df.head()

import matplotlib.pyplot as plt

# Plot explained variance
plt.figure(figsize=(10, 6))
plt.bar(range(1, n_components + 1), explained_variance, alpha=0.7, color='pink')
plt.ylabel('Explained Variance Ratio')
plt.xlabel('Components')
plt.title('Explained Variance by Components')
plt.xticks(range(1, n_components + 1))
plt.grid()
plt.show()

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Split the SVD-transformed data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_svd, y, test_size=0.2, random_state=42)

# Fit a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")

# Make predictions
y_pred = model.predict(X_test)

# Calculate R-squared
```

```
r2 = r2_score(y_test, y_pred)

# Display the R-squared value
print(f"R-squared: {r2:.4f}")

from scipy.stats import spearmanr
spearman = spearmanr(y_test, y_pred)
print(spearman)

from scipy.stats import pearsonr
pearson = pearsonr(y_test, y_pred)
print(pearson)

# Create a scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', linestyle='--')  # Diagonal line
plt.title('Actual vs Predicted Final Grades (G3)')
plt.xlabel('Actual Final Grades (G3)')
plt.ylabel('Predicted Final Grades (G3)')
plt.grid(True)
plt.show()



import numpy as np

# Gradient Descent for Linear Regression
def gradient_descent(X, y, learning_rate=0.01, epochs=1000):
    m, n = X.shape
    theta = np.zeros(n)
    cost_history = []

    for epoch in range(epochs):
        # Prediction
        y_pred = X.dot(theta)
        error = y_pred - y
```

```
        # Compute gradients
        gradients = X.T.dot(error) / m

        # Update parameters
        theta -= learning_rate * gradients

        # Compute and store cost (MSE)
        cost = (1 / (2 * m)) * np.sum(error**2)
        cost_history.append(cost)

    return theta, cost_history

# Example usage:
X_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]
theta_opt, cost_hist = gradient_descent(X_b, y_train, learning_rate=0.01, epochs=1000)

# Optimal parameters (theta)
print("Optimized parameters:", theta_opt)

from sklearn.model_selection import RandomizedSearchCV



# Define the Random Search
random_search = RandomizedSearchCV(estimator=rf, param_distributions=param_grid,
n_iter=10, cv=3, scoring='neg_mean_squared_error', random_state=42, n_jobs=-1)

# Fit the random search to the data
random_search.fit(X_train, y_train)

# Best parameters and the best score
best_params_random = random_search.best_params_
best_score_random = random_search.best_score_

print(f"Best Parameters from Random Search: {best_params_random}")
```

```
print(f"Best Cross-Validated Score (MSE) from Random Search: {-best_score_random}")

from sklearn.metrics import mean_squared_error

# Predict using the best estimator from grid search
y_pred = grid_search.best_estimator_.predict(X_test)

# Compute the Mean Squared Error
mse_test = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Set: {mse_test}")

# Define states, actions, and transition probabilities
states = ['Low', 'Average', 'High']
actions = ['StudyMore','DecreaseAbsences', 'Extra Activities']

# Define rewards matrix
rewards = {
    ('Low', 'StudyMore', 'Average'): 10,
    ('Low', 'Extra Activities', 'Average'): 5,
    ('Average', 'StudyMore', 'High'): 20,
    ('Average', 'DecreaseAbsences', 'High'): 15,
    ('High', 'Extra Activities', 'Low'): -15
}


# Transition probabilities
transition_probs = {
    ('Low', 'StudyMore', 'Average'): 0.7,
    ('Low', 'Extra Activities', 'Average'): 0.5,
    ('Average', 'StudyMore', 'High'): 0.6,
    ('Average', 'DecreaseAbsences', 'High'): 0.5,
    ('High', 'Extra Activities', 'Low'): 0.3
}

# Initialize value function
values = {state: 0 for state in states}
gamma = 0.9
```

```
# Value iteration
for _ in range(1000):
    new_values = values.copy()
    for state in states:
        new_values[state] = max(
            sum(
                transition_probs.get((state, action, next_state), 0) *
                (rewards.get((state, action, next_state), 0) + gamma * values[next_state])
                for next_state in states
            )
            for action in actions
        )
    values = new_values

# Print the optimal values
print("Optimal Values for each state:", values)

optimal_policy = {}
for state in states:
    optimal_action = max(actions, key=lambda action: sum(
        transition_probs.get((state, action, next_state), 0) *
        (rewards.get((state, action, next_state), 0) + gamma * values[next_state])
        for next_state in states
    ))
    optimal_policy[state] = optimal_action

# Print the optimal policy
print("Optimal Policy for each state:", optimal_policy)
```