In [1]:
```
import pandas as pd
import numpy as np
df = pd.read_csv('Students Performance .csv')

df.head(10)
```

Out[1]:

	Student_ID	Student_Age	Sex	High_School_Type	Scholarship	Additional_Work	Spo
0	STUDENT1	19-22	Male	Other	50%	Yes	
1	STUDENT2	19-22	Male	Other	50%	Yes	
2	STUDENT3	19-22	Male	State	50%	No	
3	STUDENT4	18	Female	Private	50%	Yes	
4	STUDENT5	19-22	Male	Private	50%	No	
5	STUDENT6	19-22	Male	State	50%	No	
6	STUDENT7	18	Male	State	75%	No	
7	STUDENT8	18	Female	State	50%	Yes	
8	STUDENT9	19-22	Female	Other	50%	No	
9	STUDENT10	19-22	Female	State	50%	No	

In [2]:
```
print(df.columns.tolist())
```

```
['Student_ID', 'Student_Age', 'Sex', 'High_School_Type', 'Scholarship', 'Additional_
Work', 'Sports_activity', 'Transportation', 'Weekly_Study_Hours', 'Attendance', 'Rea
ding', 'Notes', 'Listening_in_Class', 'Project_work', 'Grade']
```

In [7]:
```
# student_performance_prediction_updated.py

import pandas as pd
import numpy as np
import joblib
import os
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler, OrdinalEncoder
from sklearn.linear_model import LinearRegression, Ridge, Lasso, LogisticRegression
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier, Gradien
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, accu
import warnings
warnings.filterwarnings('ignore')

class StudentPerformancePredictor:
    def __init__(self):
        self.model = None
        self.scaler = StandardScaler()
        self.label_encoders = {}
        self.grade_encoder = None
        self.feature_columns = None
```

```
        self.is_trained = False
        self.problem_type = None  # 'regression' or 'classification'

    def load_and_preprocess_data(self, filepath):
        """
        Load and preprocess the student performance dataset
        """
        print("Loading dataset...")
        df = pd.read_csv(filepath)
        print(f"Dataset loaded with shape: {df.shape}")

        # Display basic info
        print("\nDataset Info:")
        print(f"Total students: {len(df)}")
        print(f"Features: {list(df.columns)}")

        # Check for missing values
        print(f"\nMissing values:\n{df.isnull().sum()}")

        # Handle missing values if any
        df = df.dropna()  # Simple approach - drop rows with missing values

        # Analyze the Grade column to determine problem type
        print("\n" + "="*50)
        print("GRADE COLUMN ANALYSIS")
        print("="*50)

        grade_data = df['Grade']
        print(f"Grade data type: {grade_data.dtype}")
        print(f"Unique values: {grade_data.unique()[:20]}")  # Show first 20 unique
        print(f"Number of unique values: {grade_data.nunique()}")

        # Check if grades are numeric or categorical
        try:
            # Try to convert to numeric
            numeric_grades = pd.to_numeric(grade_data, errors='coerce')
            if numeric_grades.notna().all():
                self.problem_type = 'regression'
                print("✓ Grades are numeric - using regression approach")
            else:
                self.problem_type = 'classification'
                print("✓ Grades are categorical - using classification approach")
        except:
            self.problem_type = 'classification'
            print("✓ Grades are categorical - using classification approach")

        return df

    def preprocess_features(self, df, is_training=True):
        """
        Preprocess features: encode categorical variables and scale numerical featu
        """
        # Separate features and target
        X = df.drop(['Student_ID'], axis=1)
        y = df['Grade'] if 'Grade' in df.columns else None
```

```
        # Identify categorical and numerical columns in features
        categorical_cols = ['Sex', 'High_School_Type', 'Scholarship', 'Additional_W
                           'Sports_activity', 'Transportation']

        # Find which categorical columns actually exist in the data
        existing_categorical_cols = [col for col in categorical_cols if col in X.co

        # Find numerical columns (all remaining columns except Grade)
        potential_numerical = [col for col in X.columns if col != 'Grade' and col n
        numerical_cols = []

        for col in potential_numerical:
            if pd.api.types.is_numeric_dtype(X[col]):
                numerical_cols.append(col)
            else:
                # If it's not numeric, add to categorical
                existing_categorical_cols.append(col)

        print(f"\nCategorical features: {existing_categorical_cols}")
        print(f"Numerical features: {numerical_cols}")

        # Encode categorical variables
        X_encoded = X.copy()

        for col in existing_categorical_cols:
            if col in X.columns:
                if is_training:
                    self.label_encoders[col] = LabelEncoder()
                    # Handle any NaN values
                    X[col] = X[col].fillna('Unknown')
                    X_encoded[col] = self.label_encoders[col].fit_transform(X[col].
                else:
                    # Handle unseen categories during prediction
                    try:
                        X[col] = X[col].fillna('Unknown')
                        X_encoded[col] = self.label_encoders[col].transform(X[col].
                    except ValueError:
                        # If unseen category, set to -1
                        X_encoded[col] = -1

        # Handle target variable (Grade)
        if y is not None:
            if self.problem_type == 'classification':
                # Encode categorical grades
                if is_training:
                    self.grade_encoder = LabelEncoder()
                    y_encoded = self.grade_encoder.fit_transform(y.astype(str))
                    print(f"\nGrade categories: {list(self.grade_encoder.classes_)}
                else:
                    try:
                        y_encoded = self.grade_encoder.transform(y.astype(str))
                    except:
                        y_encoded = -1 * np.ones(len(y))
            else:
                # Convert to float for regression
                y_encoded = pd.to_numeric(y, errors='coerce').fillna(0).astype(floa
```

```
        else:
            y_encoded = None

        # Scale numerical features
        if numerical_cols:
            if is_training:
                X_encoded[numerical_cols] = self.scaler.fit_transform(X_encoded[num
            else:
                # Ensure columns match and handle missing columns
                for col in numerical_cols:
                    if col not in X_encoded.columns:
                        X_encoded[col] = 0
                X_encoded[numerical_cols] = self.scaler.transform(X_encoded[numeric

        # Store feature columns
        if is_training:
            self.feature_columns = X_encoded.columns.tolist()
        else:
            # Ensure all training columns exist
            for col in self.feature_columns:
                if col not in X_encoded.columns:
                    X_encoded[col] = 0
            X_encoded = X_encoded[self.feature_columns]

        return X_encoded, y_encoded

    def train_model(self, X, y, model_type='auto', test_size=0.2):
        """
        Train the appropriate model based on problem type
        """
        # Split the data
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=test_size, random_state=42
        )

        # Auto-detect best model if 'auto' is selected
        if model_type == 'auto':
            if self.problem_type == 'regression':
                # Try multiple models and pick the best
                models_to_try = {
                    'linear': LinearRegression(),
                    'ridge': Ridge(alpha=1.0),
                    'random_forest': RandomForestRegressor(n_estimators=100, random
                    'gradient_boosting': GradientBoostingRegressor(n_estimators=100
                }
            else:
                models_to_try = {
                    'logistic': LogisticRegression(max_iter=1000, random_state=42),
                    'random_forest': RandomForestClassifier(n_estimators=100, rando
                    'gradient_boosting': GradientBoostingClassifier(n_estimators=10
                }

            best_score = -np.inf
            best_model_name = None
            best_model = None
```

```
            print(f"\nTrying multiple {self.problem_type} models...")

            for name, model in models_to_try.items():
                model.fit(X_train, y_train)
                if self.problem_type == 'regression':
                    score = model.score(X_test, y_test)
                else:
                    score = accuracy_score(y_test, model.predict(X_test))

                print(f"{name}: {score:.4f}")

                if score > best_score:
                    best_score = score
                    best_model_name = name
                    best_model = model

            self.model = best_model
            print(f"\nSelected best model: {best_model_name} with score: {best_scor

        else:
            # Use specified model type
            if self.problem_type == 'regression':
                if model_type == 'linear':
                    self.model = LinearRegression()
                elif model_type == 'ridge':
                    self.model = Ridge(alpha=1.0)
                elif model_type == 'lasso':
                    self.model = Lasso(alpha=0.01)
                elif model_type == 'random_forest':
                    self.model = RandomForestRegressor(n_estimators=100, random_sta
                elif model_type == 'gradient_boosting':
                    self.model = GradientBoostingRegressor(n_estimators=100, random
                else:
                    self.model = LinearRegression()
            else:
                if model_type == 'logistic':
                    self.model = LogisticRegression(max_iter=1000, random_state=42)
                elif model_type == 'random_forest':
                    self.model = RandomForestClassifier(n_estimators=100, random_st
                elif model_type == 'gradient_boosting':
                    self.model = GradientBoostingClassifier(n_estimators=100, rando
                else:
                    self.model = LogisticRegression(max_iter=1000, random_state=42)

            print(f"\nTraining {model_type} {self.problem_type} model...")
            self.model.fit(X_train, y_train)

        # Make predictions
        y_pred_train = self.model.predict(X_train)
        y_pred_test = self.model.predict(X_test)

        # Evaluate model
        self.evaluate_model(y_train, y_pred_train, y_test, y_pred_test)

        self.is_trained = True
        return self.model
```

```
    def evaluate_model(self, y_train, y_pred_train, y_test, y_pred_test):
        """
        Evaluate model performance using appropriate metrics
        """
        print("\n" + "="*50)
        print(f"MODEL EVALUATION METRICS ({self.problem_type.upper()})")
        print("="*50)

        if self.problem_type == 'regression':
            # Regression metrics
            print("\nTRAINING SET:")
            print(f"MAE: {mean_absolute_error(y_train, y_pred_train):.4f}")
            print(f"RMSE: {np.sqrt(mean_squared_error(y_train, y_pred_train)):.4f}"
            print(f"R² Score: {r2_score(y_train, y_pred_train):.4f}")

            print("\nTESTING SET:")
            print(f"MAE: {mean_absolute_error(y_test, y_pred_test):.4f}")
            print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred_test)):.4f}")
            print(f"R² Score: {r2_score(y_test, y_pred_test):.4f}")

        else:
            # Classification metrics
            print("\nTRAINING SET:")
            print(f"Accuracy: {accuracy_score(y_train, y_pred_train):.4f}")

            print("\nTESTING SET:")
            print(f"Accuracy: {accuracy_score(y_test, y_pred_test):.4f}")

            # Get unique classes present in test set
            unique_classes = np.unique(y_test)

            if len(unique_classes) <= 10:  # Only show detailed report if not too m
                print("\nClassification Report (Test Set):")
                print(classification_report(y_test, y_pred_test, zero_division=0))

                print("\nConfusion Matrix (Test Set):")
                print(confusion_matrix(y_test, y_pred_test))

    def predict_grade(self, student_data):
        """
        Predict grade for a single student
        """
        if not self.is_trained:
            raise Exception("Model not trained yet. Please train the model first.")

        # Convert input to DataFrame
        if isinstance(student_data, dict):
            df_input = pd.DataFrame([student_data])
        else:
            df_input = student_data

        # Preprocess
        X_processed, _ = self.preprocess_features(df_input, is_training=False)

        # Predict
```

```
        prediction = self.model.predict(X_processed)[0]

        # Convert back to original grade format if classification
        if self.problem_type == 'classification' and self.grade_encoder is not None
            # Handle float predictions (round to nearest integer for class index)
            if isinstance(prediction, (np.floating, float)):
                prediction = int(round(prediction))
            # Ensure prediction is within valid range
            if 0 <= prediction < len(self.grade_encoder.classes_):
                prediction = self.grade_encoder.inverse_transform([prediction])[0]
            else:
                prediction = "Unknown"

        return prediction

    def save_model(self, filepath='student_performance_model.pkl'):
        """
        Save the trained model and preprocessors
        """
        if not self.is_trained:
            raise Exception("No trained model to save.")

        model_data = {
            'model': self.model,
            'scaler': self.scaler,
            'label_encoders': self.label_encoders,
            'grade_encoder': self.grade_encoder,
            'feature_columns': self.feature_columns,
            'problem_type': self.problem_type
        }

        joblib.dump(model_data, filepath)
        print(f"\nModel saved successfully to {filepath}")

    def load_model(self, filepath='student_performance_model.pkl'):
        """
        Load a trained model
        """
        if not os.path.exists(filepath):
            raise FileNotFoundError(f"Model file {filepath} not found.")

        model_data = joblib.load(filepath)
        self.model = model_data['model']
        self.scaler = model_data['scaler']
        self.label_encoders = model_data['label_encoders']
        self.grade_encoder = model_data.get('grade_encoder')
        self.feature_columns = model_data['feature_columns']
        self.problem_type = model_data.get('problem_type', 'classification')
        self.is_trained = True

        print(f"\nModel loaded successfully from {filepath}")
        print(f"Problem type: {self.problem_type}")

    def analyze_feature_importance(self):
        """
        Analyze feature importance
```

```
        """
        if self.feature_columns is None:
            print("\nFeature information not available.")
            return

        print("\n" + "="*50)
        print("FEATURE ANALYSIS")
        print("="*50)

        if hasattr(self.model, 'feature_importances_'):
            # Tree-based models
            importances = self.model.feature_importances_
            indices = np.argsort(importances)[::-1]

            print("\nFeature Importance (higher = more important):")
            for i, idx in enumerate(indices):
                if i < min(10, len(self.feature_columns)):
                    print(f"{i+1}. {self.feature_columns[idx]}: {importances[idx]:.

        elif hasattr(self.model, 'coef_'):
            # Linear models
            coefficients = self.model.coef_
            if len(coefficients.shape) > 1:
                # For multi-class, take mean absolute coefficient
                coefficients = np.mean(np.abs(coefficients), axis=0)

            indices = np.argsort(np.abs(coefficients))[::-1]

            print("\nFeature Coefficients (absolute value):")
            for i, idx in enumerate(indices):
                if i < min(10, len(self.feature_columns)):
                    print(f"{i+1}. {self.feature_columns[idx]}: {coefficients[idx]:
        else:
            print("Feature importance not available for this model type.")

def main():
    """
    Main CLI interface for the Student Performance Prediction System
    """
    predictor = StudentPerformancePredictor()

    print("="*60)
    print("STUDENT PERFORMANCE PREDICTION SYSTEM")
    print("="*60)

    while True:
        print("\n" + "-"*40)
        print("MAIN MENU")
        print("-"*40)
        print("1. Train new model")
        print("2. Load existing model")
        print("3. Predict student grade")
        print("4. Analyze feature importance")
        print("5. Exit")

        choice = input("\nEnter your choice (1-5): ").strip()
```

```
        if choice == '1':
            # Train new model
            filepath = input("Enter dataset path (default: Students Performance .cs
            if not filepath:
                filepath = "Students Performance .csv"

            try:
                # Load and preprocess data
                df = predictor.load_and_preprocess_data(filepath)

                # Preprocess features
                X, y = predictor.preprocess_features(df, is_training=True)

                if predictor.problem_type == 'regression':
                    print("\nSelect regression model:")
                    print("1. Linear Regression")
                    print("2. Ridge Regression")
                    print("3. Lasso Regression")
                    print("4. Random Forest")
                    print("5. Gradient Boosting")
                    print("6. Auto (try multiple and pick best)")
                else:
                    print("\nSelect classification model:")
                    print("1. Logistic Regression")
                    print("2. Random Forest")
                    print("3. Gradient Boosting")
                    print("4. Auto (try multiple and pick best)")

                model_choice = input("Enter choice (1-6): ").strip()

                model_types_reg = {
                    '1': 'linear',
                    '2': 'ridge',
                    '3': 'lasso',
                    '4': 'random_forest',
                    '5': 'gradient_boosting',
                    '6': 'auto'
                }

                model_types_clf = {
                    '1': 'logistic',
                    '2': 'random_forest',
                    '3': 'gradient_boosting',
                    '4': 'auto'
                }

                if predictor.problem_type == 'regression':
                    model_type = model_types_reg.get(model_choice, 'auto')
                else:
                    model_type = model_types_clf.get(model_choice, 'auto')

                # Train model
                predictor.train_model(X, y, model_type=model_type)

                # Save model
```

```
                save_choice = input("\nSave model? (y/n): ").strip().lower()
                if save_choice == 'y':
                    model_name = input("Enter model filename (default: student_mode
                    if not model_name:
                        model_name = "student_model.pkl"
                    predictor.save_model(model_name)

            except Exception as e:
                print(f"\nError: {e}")
                import traceback
                traceback.print_exc()

        elif choice == '2':
            # Load existing model
            model_name = input("Enter model filename (default: student_model.pkl):
            if not model_name:
                model_name = "student_model.pkl"

            try:
                predictor.load_model(model_name)
            except Exception as e:
                print(f"\nError loading model: {e}")

        elif choice == '3':
            # Predict student grade
            if not predictor.is_trained:
                print("\nPlease train or load a model first!")
                continue

            print("\n" + "-"*40)
            print("ENTER STUDENT DETAILS FOR PREDICTION")
            print("-"*40)
            print("(Press Enter to skip optional fields)")

            try:
                # Collect student data
                student_data = {}

                # Required fields
                student_data['Student_Age'] = int(input("Student Age: "))

                # Categorical fields
                sex = input("Sex (Male/Female): ").strip()
                if sex:
                    student_data['Sex'] = sex

                school_type = input("High School Type (Public/Private/Other): ").st
                if school_type:
                    student_data['High_School_Type'] = school_type

                scholarship = input("Scholarship (Yes/No): ").strip()
                if scholarship:
                    student_data['Scholarship'] = scholarship

                additional_work = input("Additional Work (Yes/No): ").strip()
                if additional_work:
```

```
            student_data['Additional_Work'] = additional_work

        sports = input("Sports Activity (Yes/No): ").strip()
        if sports:
            student_data['Sports_activity'] = sports

        transport = input("Transportation (Bus/Walk/Other): ").strip()
        if transport:
            student_data['Transportation'] = transport

        # Numerical fields
        study_hours = input("Weekly Study Hours: ").strip()
        if study_hours:
            student_data['Weekly_Study_Hours'] = float(study_hours)

        attendance = input("Attendance Percentage (0-100): ").strip()
        if attendance:
            student_data['Attendance'] = float(attendance)

        reading = input("Reading Score (0-100): ").strip()
        if reading:
            student_data['Reading'] = float(reading)

        notes = input("Notes Taking Score (0-100): ").strip()
        if notes:
            student_data['Notes'] = float(notes)

        listening = input("Listening in Class Score (0-100): ").strip()
        if listening:
            student_data['Listening_in_Class'] = float(listening)

        project = input("Project Work Score (0-100): ").strip()
        if project:
            student_data['Project_work'] = float(project)

        # Make prediction
        predicted_grade = predictor.predict_grade(student_data)

        print("\n" + "="*40)
        print("PREDICTION RESULT")
        print("="*40)
        print(f"Predicted Grade: {predicted_grade}")

        # Provide interpretation for numeric grades
        if predictor.problem_type == 'regression':
            try:
                grade_num = float(predicted_grade)
                if grade_num >= 90:
                    print("Performance: Excellent (A)")
                elif grade_num >= 80:
                    print("Performance: Very Good (B)")
                elif grade_num >= 70:
                    print("Performance: Good (C)")
                elif grade_num >= 60:
                    print("Performance: Satisfactory (D)")
                else:
```

```
                            print("Performance: Needs Improvement (F)")
                    except:
                        pass

            except Exception as e:
                print(f"\nError in prediction: {e}")
                import traceback
                traceback.print_exc()

        elif choice == '4':
            # Analyze feature importance
            if not predictor.is_trained:
                print("\nPlease train or load a model first!")
                continue

            predictor.analyze_feature_importance()

        elif choice == '5':
            print("\nThank you for using Student Performance Prediction System!")
            break

        else:
            print("\nInvalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

```
============================================================
STUDENT PERFORMANCE PREDICTION SYSTEM
============================================================

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Invalid choice. Please try again.

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
```

```
Invalid choice. Please try again.

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Invalid choice. Please try again.

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Invalid choice. Please try again.

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Invalid choice. Please try again.

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Invalid choice. Please try again.

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
```

```
Loading dataset...
Dataset loaded with shape: (145, 15)

Dataset Info:
Total students: 145
Features: ['Student_ID', 'Student_Age', 'Sex', 'High_School_Type', 'Scholarship', 'A
dditional_Work', 'Sports_activity', 'Transportation', 'Weekly_Study_Hours', 'Attenda
nce', 'Reading', 'Notes', 'Listening_in_Class', 'Project_work', 'Grade']

Missing values:
Student_ID            0
Student_Age           0
Sex                   0
High_School_Type      0
Scholarship           1
Additional_Work       0
Sports_activity       0
Transportation        0
Weekly_Study_Hours    0
Attendance            0
Reading               0
Notes                 0
Listening_in_Class    0
Project_work          0
Grade                 0
dtype: int64

==================================================
GRADE COLUMN ANALYSIS
==================================================
Grade data type: object
Unique values: ['AA' 'BA' 'CC' 'Fail' 'BB' 'CB' 'DD' 'DC']
Number of unique values: 8
✓ Grades are categorical - using classification approach

Categorical features: ['Sex', 'High_School_Type', 'Scholarship', 'Additional_Work',
'Sports_activity', 'Transportation', 'Student_Age', 'Attendance', 'Reading', 'Note
s', 'Listening_in_Class', 'Project_work']
Numerical features: ['Weekly_Study_Hours']

Grade categories: ['AA', 'BA', 'BB', 'CB', 'CC', 'DC', 'DD', 'Fail']

Select classification model:
1. Logistic Regression
2. Random Forest
3. Gradient Boosting
4. Auto (try multiple and pick best)
```

```
Traceback (most recent call last):
  File "C:\Users\ASIM ALI\AppData\Local\Temp\ipykernel_29208\2537903796.py", line 46
7, in main
    predictor.train_model(X, y, model_type=model_type)
  File "C:\Users\ASIM ALI\AppData\Local\Temp\ipykernel_29208\2537903796.py", line 23
5, in train_model
    self.model.fit(X_train, y_train)
  File "C:\Users\ASIM ALI\anaconda3\envs\fresh_env\lib\site-packages\sklearn\base.p
y", line 1365, in wrapper
    return fit_method(estimator, *args, **kwargs)
  File "C:\Users\ASIM ALI\anaconda3\envs\fresh_env\lib\site-packages\sklearn\ensembl
e\_forest.py", line 359, in fit
    X, y = validate_data(
  File "C:\Users\ASIM ALI\anaconda3\envs\fresh_env\lib\site-packages\sklearn\utils\v
alidation.py", line 2971, in validate_data
    X, y = check_X_y(X, y, **check_params)
  File "C:\Users\ASIM ALI\anaconda3\envs\fresh_env\lib\site-packages\sklearn\utils\v
alidation.py", line 1368, in check_X_y
    X = check_array(
  File "C:\Users\ASIM ALI\anaconda3\envs\fresh_env\lib\site-packages\sklearn\utils\v
alidation.py", line 1053, in check_array
    array = _asarray_with_order(array, order=order, dtype=dtype, xp=xp)
  File "C:\Users\ASIM ALI\anaconda3\envs\fresh_env\lib\site-packages\sklearn\utils\_
array_api.py", line 757, in _asarray_with_order
    array = numpy.asarray(array, order=order, dtype=dtype)
  File "C:\Users\ASIM ALI\anaconda3\envs\fresh_env\lib\site-packages\pandas\core\gen
eric.py", line 2168, in __array__
    arr = np.asarray(values, dtype=dtype)
ValueError: could not convert string to float: 'BA'
Training random_forest classification model...

Error: could not convert string to float: 'BA'

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Please train or load a model first!

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
```

```
Please train or load a model first!

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Invalid choice. Please try again.

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Invalid choice. Please try again.

----------------------------------------
MAIN MENU
----------------------------------------
1. Train new model
2. Load existing model
3. Predict student grade
4. Analyze feature importance
5. Exit
Thank you for using Student Performance Prediction System!
```

In [9]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter

def explore_dataset(filepath="Students Performance .csv"):
    """
    Explore and visualize the student performance dataset
    """
    # Load data
    df = pd.read_csv(filepath)

    print("="*60)
    print("STUDENT PERFORMANCE DATASET EXPLORATION")
    print("="*60)

    # Basic statistics
    print("\n1. DATASET OVERVIEW")
    print("-"*40)
    print(f"Shape: {df.shape}")
    print(f"Columns: {list(df.columns)}")
    print(f"\nData Types:\n{df.dtypes}")
```

```
# Check for missing values
print("\n2. MISSING VALUES")
print("-"*40)
print(df.isnull().sum())

# Grade analysis
print("\n3. GRADE DISTRIBUTION")
print("-"*40)

grade_data = df['Grade']

# Try to determine if grades are numeric or categorical
try:
    numeric_grades = pd.to_numeric(grade_data, errors='coerce')
    if numeric_grades.notna().all():
        print("Grades are NUMERIC")
        print(f"Mean Grade: {grade_data.mean():.2f}")
        print(f"Median Grade: {grade_data.median():.2f}")
        print(f"Std Deviation: {grade_data.std():.2f}")
        print(f"Min Grade: {grade_data.min()}")
        print(f"Max Grade: {grade_data.max()}")
    else:
        print("Grades are CATEGORICAL")
        grade_counts = grade_data.value_counts()
        print(f"Unique grade values: {grade_data.nunique()}")
        print("\nGrade frequency:")
        for grade, count in grade_counts.head(10).items():
            print(f"  {grade}: {count} ({count/len(df)*100:.1f}%)")
except:
    print("Grades are CATEGORICAL")
    grade_counts = grade_data.value_counts()
    print(f"Unique grade values: {grade_data.nunique()}")
    print("\nGrade frequency (top 10):")
    for grade, count in grade_counts.head(10).items():
        print(f"  {grade}: {count} ({count/len(df)*100:.1f}%)")

# Summary statistics for numerical columns
print("\n4. NUMERICAL FEATURES SUMMARY")
print("-"*40)
numerical_cols = df.select_dtypes(include=[np.number]).columns
if len(numerical_cols) > 0:
    print(df[numerical_cols].describe())
else:
    print("No numerical columns found (excluding Grade)")

# Categorical columns analysis
print("\n5. CATEGORICAL FEATURES SUMMARY")
print("-"*40)
categorical_cols = df.select_dtypes(include=['object']).columns
categorical_cols = [col for col in categorical_cols if col != 'Grade']

for col in categorical_cols:
    print(f"\n{col}:")
    value_counts = df[col].value_counts()
    for val, count in value_counts.head(5).items():
        print(f"  {val}: {count} ({count/len(df)*100:.1f}%)")
```

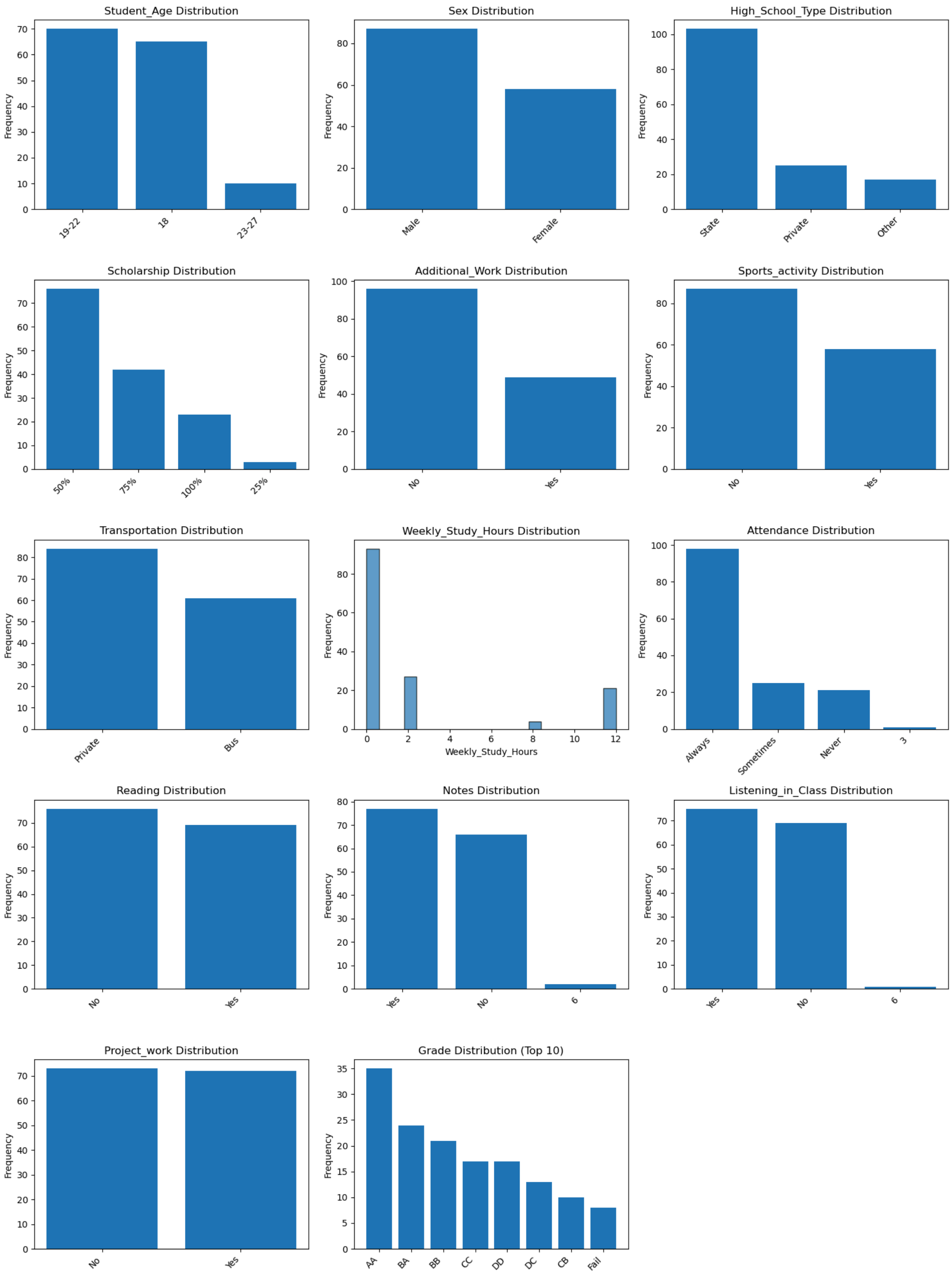
```
    # Create visualizations
    print("\n6. GENERATING VISUALIZATIONS...")

    # Determine grid size based on number of columns
    n_cols = min(3, len(df.columns))
    n_rows = (len(df.columns) + n_cols - 1) // n_cols

    fig, axes = plt.subplots(n_rows, n_cols, figsize=(5*n_cols, 4*n_rows))
    if n_rows == 1:
        axes = [axes]
    axes_flat = [ax for row in axes for ax in row]

    plot_idx = 0

    for col in df.columns:
        if col == 'Student_ID':
            continue

        ax = axes_flat[plot_idx]

        if col == 'Grade':
            # Special handling for Grade column
            try:
                numeric_grades = pd.to_numeric(df[col], errors='coerce')
                if numeric_grades.notna().all():
                    ax.hist(df[col].dropna(), bins=20, edgecolor='black', alpha=0.7
                    ax.set_title(f'{col} Distribution')
                    ax.set_xlabel(col)
                else:
                    # Categorical grade - show bar chart
                    grade_counts = df[col].value_counts().head(10)
                    ax.bar(range(len(grade_counts)), grade_counts.values)
                    ax.set_xticks(range(len(grade_counts)))
                    ax.set_xticklabels(grade_counts.index, rotation=45, ha='right')
                    ax.set_title(f'{col} Distribution (Top 10)')
            except:
                grade_counts = df[col].value_counts().head(10)
                ax.bar(range(len(grade_counts)), grade_counts.values)
                ax.set_xticks(range(len(grade_counts)))
                ax.set_xticklabels(grade_counts.index, rotation=45, ha='right')
                ax.set_title(f'{col} Distribution (Top 10)')

        elif pd.api.types.is_numeric_dtype(df[col]):
            # Numerical column - histogram
            ax.hist(df[col].dropna(), bins=20, edgecolor='black', alpha=0.7)
            ax.set_title(f'{col} Distribution')
            ax.set_xlabel(col)

        else:
            # Categorical column - bar chart
            value_counts = df[col].value_counts().head(8)
            ax.bar(range(len(value_counts)), value_counts.values)
            ax.set_xticks(range(len(value_counts)))
            ax.set_xticklabels(value_counts.index, rotation=45, ha='right')
            ax.set_title(f'{col} Distribution')
```

```
        ax.set_ylabel('Frequency')
        plot_idx += 1

    # Hide unused subplots
    for idx in range(plot_idx, len(axes_flat)):
        axes_flat[idx].set_visible(False)

    plt.tight_layout()
    plt.savefig('student_performance_analysis.png', dpi=300, bbox_inches='tight')
    plt.show()

    print("\nVisualizations saved to 'student_performance_analysis.png'")

    # Correlation analysis for numerical features
    print("\n7. CORRELATION ANALYSIS")
    print("-"*40)
    numerical_cols = df.select_dtypes(include=[np.number]).columns.tolist()
    if 'Student_ID' in numerical_cols:
        numerical_cols.remove('Student_ID')

    if len(numerical_cols) > 1:
        corr_matrix = df[numerical_cols].corr()
        print("Correlation matrix:")
        print(corr_matrix)

        # Plot correlation heatmap
        plt.figure(figsize=(10, 8))
        sns.heatmap(corr_matrix, annot=True, fmt='.2f', cmap='coolwarm', center=0)
        plt.title('Feature Correlation Heatmap')
        plt.tight_layout()
        plt.savefig('correlation_heatmap.png', dpi=300, bbox_inches='tight')
        plt.show()
        print("\nCorrelation heatmap saved to 'correlation_heatmap.png'")

    return df

if __name__ == "__main__":
    df = explore_dataset()
```

```
============================================================
STUDENT PERFORMANCE DATASET EXPLORATION
============================================================

1. DATASET OVERVIEW
----------------------------------------
Shape: (145, 15)
Columns: ['Student_ID', 'Student_Age', 'Sex', 'High_School_Type', 'Scholarship', 'Ad
ditional_Work', 'Sports_activity', 'Transportation', 'Weekly_Study_Hours', 'Attendan
ce', 'Reading', 'Notes', 'Listening_in_Class', 'Project_work', 'Grade']

Data Types:
Student_ID            object
Student_Age           object
Sex                   object
High_School_Type      object
Scholarship           object
Additional_Work       object
Sports_activity       object
Transportation        object
Weekly_Study_Hours     int64
Attendance            object
Reading               object
Notes                 object
Listening_in_Class    object
Project_work          object
Grade                 object
dtype: object

2. MISSING VALUES
----------------------------------------
Student_ID            0
Student_Age           0
Sex                   0
High_School_Type      0
Scholarship           1
Additional_Work       0
Sports_activity       0
Transportation        0
Weekly_Study_Hours    0
Attendance            0
Reading               0
Notes                 0
Listening_in_Class    0
Project_work          0
Grade                 0
dtype: int64

3. GRADE DISTRIBUTION
----------------------------------------
Grades are CATEGORICAL
Unique grade values: 8

Grade frequency:
  AA: 35 (24.1%)
  BA: 24 (16.6%)
```

```
  BB: 21 (14.5%)
  CC: 17 (11.7%)
  DD: 17 (11.7%)
  DC: 13 (9.0%)
  CB: 10 (6.9%)
  Fail: 8 (5.5%)

4. NUMERICAL FEATURES SUMMARY
----------------------------------------
       Weekly_Study_Hours
count          145.000000
mean             2.331034
std              4.249273
min              0.000000
25%              0.000000
50%              0.000000
75%              2.000000
max             12.000000

5. CATEGORICAL FEATURES SUMMARY
----------------------------------------

Student_ID:
  STUDENT1: 1 (0.7%)
  STUDENT74: 1 (0.7%)
  STUDENT94: 1 (0.7%)
  STUDENT95: 1 (0.7%)
  STUDENT96: 1 (0.7%)

Student_Age:
  19-22: 70 (48.3%)
  18: 65 (44.8%)
  23-27: 10 (6.9%)

Sex:
  Male: 87 (60.0%)
  Female: 58 (40.0%)

High_School_Type:
  State: 103 (71.0%)
  Private: 25 (17.2%)
  Other: 17 (11.7%)

Scholarship:
  50%: 76 (52.4%)
  75%: 42 (29.0%)
  100%: 23 (15.9%)
  25%: 3 (2.1%)

Additional_Work:
  No: 96 (66.2%)
  Yes: 49 (33.8%)

Sports_activity:
  No: 87 (60.0%)
  Yes: 58 (40.0%)
```

```
Transportation:
  Private: 84 (57.9%)
  Bus: 61 (42.1%)

Attendance:
  Always: 98 (67.6%)
  Sometimes: 25 (17.2%)
  Never: 21 (14.5%)
  3: 1 (0.7%)

Reading:
  No: 76 (52.4%)
  Yes: 69 (47.6%)

Notes:
  Yes: 77 (53.1%)
  No: 66 (45.5%)
  6: 2 (1.4%)

Listening_in_Class:
  Yes: 75 (51.7%)
  No: 69 (47.6%)
  6: 1 (0.7%)

Project_work:
  No: 73 (50.3%)
  Yes: 72 (49.7%)

6. GENERATING VISUALIZATIONS...
```




```
Visualizations saved to 'student_performance_analysis.png'

7. CORRELATION ANALYSIS
----------------------------------------
```