In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import math
student = pd.read_csv('Student_List_A2.csv')
```

A1: Data Wrangling

1.

In [4]:
```
#this allows me to see how big the database is
student.shape
```

Out[4]:
```
(2100, 7)
```

In [5]:
```
#shows me the data types for the column variables
student.dtypes
```

Out[5]:
```
StudentID            int64
Age                  int64
StudyTimeWeekly    float64
Absences             int64
ParentalSupport      int64
GPA                float64
GradeClass           int64
dtype: object
```

In [6]:
```
#allows me to see the columns
#and get an idea about what sort of values each column has
student
```

Out[6]:

	StudentID	Age	StudyTimeWeekly	Absences	ParentalSupport	GPA	GradeC
0	1002	18	15.408756	0	1	3.042915	
1	1003	15	4.210570	26	2	0.112602	
2	1004	17	10.028829	14	3	2.054218	
3	1005	17	4.672495	17	3	1.288061	
4	1006	18	8.191219	0	1	3.084184	
...	...	...	...	...	...	...	
2095	3386	16	1.445434	20	3	1.395631	
2096	3388	18	10.680555	2	4	3.455509	
2097	3390	16	6.805500	20	2	1.142333	
2098	3391	16	12.416653	17	2	1.803297	
2099	3392	16	17.819907	13	2	2.140014	

2100 rows × 7 columns

Column Names:

- StudentID
- Age
- StudyTimeWeekly
- Absences
- ParentalSupport
- GPA
- GradeClass

2.

In [9]:

```
#this replaces all the numerical values for the GradeClass with their
#corresponding letter grade in the DataFrame

student['GradeClass'] = student['GradeClass'].replace(
    {0:'A', 1:'B', 2:'C', 3:'D', 4:'F'})

#then prints the DataFrame to see the changes I have made
student
```

Out[9]:

	StudentID	Age	StudyTimeWeekly	Absences	ParentalSupport	GPA	GradeC
0	1002	18	15.408756	0	1	3.042915	
1	1003	15	4.210570	26	2	0.112602	
2	1004	17	10.028829	14	3	2.054218	
3	1005	17	4.672495	17	3	1.288061	
4	1006	18	8.191219	0	1	3.084184	
...	...	...	...	...	...	...	
2095	3386	16	1.445434	20	3	1.395631	
2096	3388	18	10.680555	2	4	3.455509	
2097	3390	16	6.805500	20	2	1.142333	
2098	3391	16	12.416653	17	2	1.803297	
2099	3392	16	17.819907	13	2	2.140014	

2100 rows × 7 columns

3.

In [11]:
```
#this gives the sum of all the null values in each column
student.isnull().sum()
```

Out[11]:
```
StudentID           0
Age                 0
StudyTimeWeekly    21
Absences            0
ParentalSupport     0
GPA                 0
GradeClass          0
dtype: int64
```

There are 21 missing values in the 'StudyTimeWeekly' column

In [13]:
```
#prints exactly those rows that have null values in the column StudyTimeWeekly

null = student[student['StudyTimeWeekly'].isnull()]
null
```

Out[13]:

	StudentID	Age	StudyTimeWeekly	Absences	ParentalSupport	GPA	GradeC
19	1021	16	NaN	2	3	2.778411	
23	1025	18	NaN	15	2	1.505156	
105	1107	16	NaN	18	0	0.842296	
126	1128	15	NaN	10	1	2.819922	
260	1262	16	NaN	20	1	1.265678	
388	1390	15	NaN	16	3	1.848866	
444	1446	17	NaN	29	4	0.869123	
492	1494	16	NaN	25	1	0.567237	
558	1560	16	NaN	5	3	3.366930	
599	1601	15	NaN	7	1	2.446157	
767	1769	18	NaN	14	2	1.736011	
965	1967	15	NaN	14	2	2.105309	
993	1995	18	NaN	26	2	0.743592	
1051	2053	17	NaN	12	1	2.254020	
1247	2249	16	NaN	17	3	1.907984	
1307	2309	15	NaN	10	3	2.759014	
1479	2481	17	NaN	0	1	3.323903	
1672	2674	15	NaN	11	2	1.858296	
1753	2755	15	NaN	19	2	1.537990	
1934	2936	18	NaN	3	3	3.471337	
2044	3276	15	NaN	15	0	2.284791	

In [14]:

```
#to see if there are students who have a weekly study time that is
#exactly the same the median for that column
equal = student[student['StudyTimeWeekly']==student['StudyTimeWeekly'].median()]
equal
```

Out[14]:

	StudentID	Age	StudyTimeWeekly	Absences	ParentalSupport	GPA	GradeC
1938	2940	17	9.513101	25	3	0.934943	

There is already a student whose weekly study time matches the median value of the weekly study time for all students. This median value will be visible when you try to print the rows where missing values from the 'WeeklyStudyTime' column have been replaced with the median.

In [16]:
```
#this replaces all the null values in the column StudyTimeWeekly with the
#median of StudyTimeWeekly
student['StudyTimeWeekly'] = student['StudyTimeWeekly'].fillna(
    student['StudyTimeWeekly'].median())
```

In [17]:
```
#this allows me to see the changes I just made
#To confirm that the null values have been replaced
null_median=student[student['StudyTimeWeekly']==student['StudyTimeWeekly'].media
null_median
```

Out[17]:

	StudentID	Age	StudyTimeWeekly	Absences	ParentalSupport	GPA	GradeC
19	1021	16	9.513101	2	3	2.778411	
23	1025	18	9.513101	15	2	1.505156	
105	1107	16	9.513101	18	0	0.842296	
126	1128	15	9.513101	10	1	2.819922	
260	1262	16	9.513101	20	1	1.265678	
388	1390	15	9.513101	16	3	1.848866	
444	1446	17	9.513101	29	4	0.869123	
492	1494	16	9.513101	25	1	0.567237	
558	1560	16	9.513101	5	3	3.366930	
599	1601	15	9.513101	7	1	2.446157	
767	1769	18	9.513101	14	2	1.736011	
965	1967	15	9.513101	14	2	2.105309	
993	1995	18	9.513101	26	2	0.743592	
1051	2053	17	9.513101	12	1	2.254020	
1247	2249	16	9.513101	17	3	1.907984	
1307	2309	15	9.513101	10	3	2.759014	
1479	2481	17	9.513101	0	1	3.323903	
1672	2674	15	9.513101	11	2	1.858296	
1753	2755	15	9.513101	19	2	1.537990	
1934	2936	18	9.513101	3	3	3.471337	
1938	2940	17	9.513101	25	3	0.934943	
2044	3276	15	9.513101	15	0	2.284791	

4.

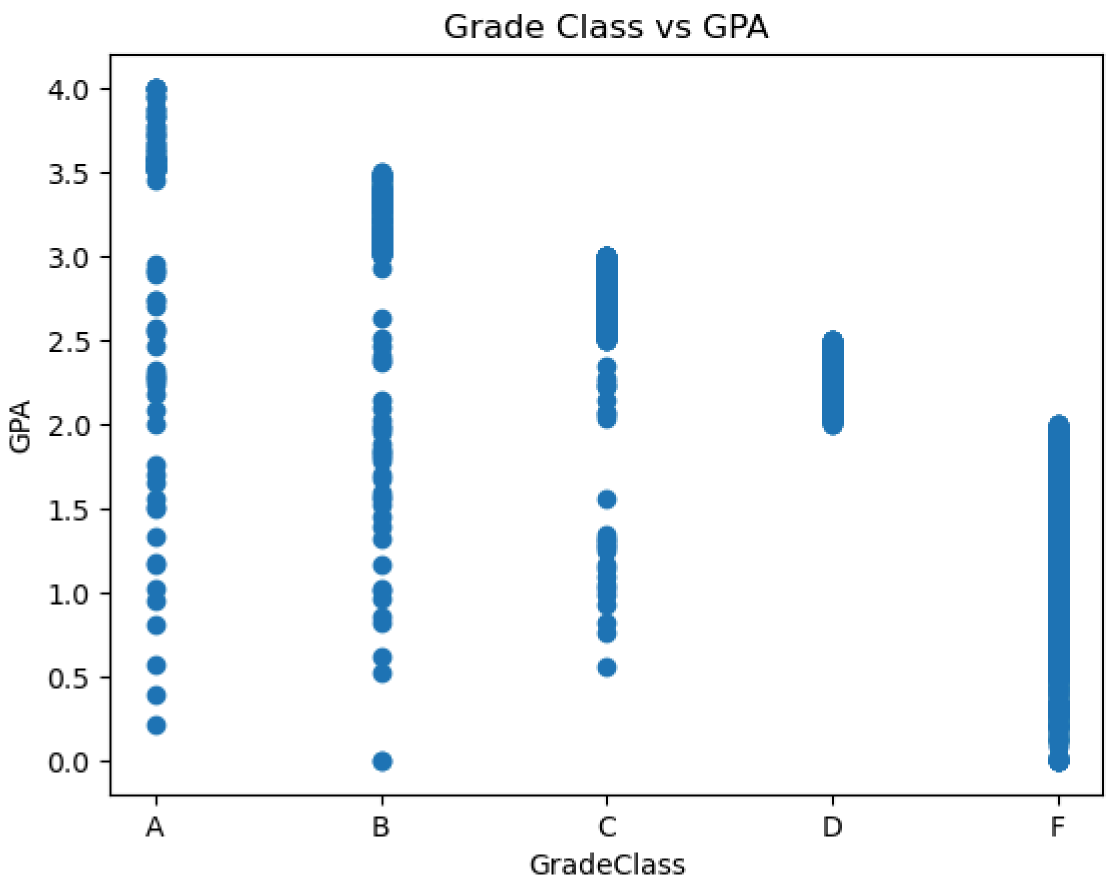
In [19]:
```
#this shows me the negative values.
#negative values don't make sense so I know what I need to eliminate and if
#there are values to eliminate
```

```
negative_absences = student[student['Absences'] < 0]

print("Rows with negative values in 'Absences':\n", negative_absences)

#this shows me the unreasonably high values
#in an academic year, a student goes to school for around 180 days.
#so anything higher than that wouldn't make sense
extreme_absences = student[student['Absences'] >= 180]

print("Rows with unreasonably high values in 'Absences':\n", extreme_absences)

#this filters out those negative value and extremely high values
student = student[(student['Absences'] >= 0) & (student['Absences'] < 180)]

print(
    "\nAfter removing rows that dont make sense:\n", student)
```

```
Rows with negative values in 'Absences':
       StudentID  Age  StudyTimeWeekly  Absences  ParentalSupport      GPA  \
1001       2003   15         0.806505      -122                3  3.20171

     GradeClass
1001          B
Rows with unreasonably high values in 'Absences':
      StudentID  Age  StudyTimeWeekly  Absences  ParentalSupport       GPA  \
112       1114   16        16.849282       320                1  1.919956

    GradeClass
112          F

After removing rows that dont make sense:
       StudentID  Age  StudyTimeWeekly  Absences  ParentalSupport       GPA  \
0          1002   18        15.408756         0                1  3.042915
1          1003   15         4.210570        26                2  0.112602
2          1004   17        10.028829        14                3  2.054218
3          1005   17         4.672495        17                3  1.288061
4          1006   18         8.191219         0                1  3.084184
...         ...  ...              ...       ...              ...       ...
2095       3386   16         1.445434        20                3  1.395631
2096       3388   18        10.680555         2                4  3.455509
2097       3390   16         6.805500        20                2  1.142333
2098       3391   16        12.416653        17                2  1.803297
2099       3392   16        17.819907        13                2  2.140014

     GradeClass
0             B
1             F
2             D
3             F
4             B
...         ...
2095          B
2096          A
2097          C
2098          B
2099          B

[2098 rows x 7 columns]
```

In [86]:
```
order = ['A', 'B', 'C', 'D', 'F']

# the line below converts 'GradeClass' to a categorical variable with a
#speciifc order that we defined above
student['GradeClass'] = pd.Categorical(
    student['GradeClass'], categories=order, ordered=True)

# creates a scatter plot with numeric codes for 'GradeClass'
plt.scatter(student['GradeClass'].cat.codes, student['GPA'])

# this sets plot title and labels
plt.title("Grade Class vs GPA")
plt.xlabel('GradeClass')
plt.ylabel('GPA')

# sets x-axis ticks to show the GradeClass categories
plt.xticks(ticks=range(len(order)), labels=order)

#displays the graph
plt.show()

#The SettingWithCopyWarning error is indicating to me that I am trying
#to modify a slice of a DF, which may lead to unexpected behavior
#but i can ignore this error in this case because I am sure I am working with a
#of the dataframe, not just a part of it, and my changes
#will be applied correctly
```

```
C:\Users\malav\AppData\Local\Temp\ipykernel_40140\627458026.py:5: SettingWithCopy
Warning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stabl
e/user_guide/indexing.html#returning-a-view-versus-a-copy
  student['GradeClass'] = pd.Categorical(
```



In [22]:

```
#this code is used to check if the graph is accurate or not
grades = ['A', 'B', 'C', 'D', 'F']

max_gpa = {}
min_gpa = {}

# this loops through each grade class
for grade in grades:
    #filters the DF for the current grade
    grade_data = student[student['GradeClass'] == grade]

    # then it finds the max and min GPA for the current grade
    max_gpa[grade] = grade_data['GPA'].max()
    min_gpa[grade] = grade_data['GPA'].min()

# prints the results
for grade in grades:
    print(f"{grade} - Max GPA: {max_gpa[grade]}, Min GPA: {min_gpa[grade]}")
```

```
A - Max GPA: 4.0, Min GPA: 0.214570091
B - Max GPA: 3.498257342, Min GPA: 0.0
C - Max GPA: 2.999544104, Min GPA: 0.557548887
D - Max GPA: 2.49968118, Min GPA: 2.00330798
F - Max GPA: 1.999571861, Min GPA: 0.0
```

The issue we are facing here is a data quality problem where the 'GradeClass' categories don't correspond to the appropriate GPA ranges. For instance, an 'A' grade should align with GPAs ranging between 3.5 and 4, but the current data reflects a mismatch, with GPAs ranging from 0.2 to 4. This discrepancy indicates that the grade classifications are not properly representing the intended GPA ranges.

To address the data quality issue where 'GradeClass' does not accurately reflect the corresponding GPA ranges, a solution involves defining correct GPA thresholds for each grade and reassigning the 'GradeClass' values based on these thresholds. For example, we can redefine the GPA range for 'A' grade to be 3.5 to 4. By applying a function to reassign 'GradeClass' based on these ranges, we ensure consistency between GPA values and their corresponding grade classifications. The following code snippet demonstrates this solution:

#this code has not been properly implemented (written format is raw) so doesn't affect the DF

#this is just an example of how I would implement logic to resolve the issue

def reassign_grade_class(gpa):

```
if 3.5 <= gpa <= 4.0:
    return 'A'
elif 3.0 <= gpa < 3.5:
    return 'B'
elif 2.5 <= gpa < 3.0:
    return 'C'
elif 2.0 <= gpa < 2.5:
    return 'D'
elif 0 <= gpa < 2.0:
    return 'F'
```

#Apply the function to the DataFrame student['GradeClass'] = student['GPA'].apply(assign_grade_class)

#Check the updated DataFrame print(student)

A2. Supervised Learning

1.

Supervised machine learning is a type of machine learning where a model is trained to make predictions or decisions based on labeled data. In this approach, each piece of input data, often called features, is paired with the correct output, referred to as the label. For instance, in a spam detection system, the input data might be an email, and the label would indicate whether the email is "spam" or "not spam". The

key idea is that the model learns from this labeled data to understand how different inputs are related to their corresponding outputs.

The process starts by providing the model with the training dataset containing numerous examples of input-output pairs. The model analyses these examples to find patterns or rules that link the inputs to the correct outputs. For instance, in image recognition, the model might learn that certain shapes or colors correspond to specific objects. During training, the model adjusts its internal parameters to minimize the number of errors in predicting the outputs from the inputs.

Once the model is trained, it is tested on a test dataset, which consists of new examples that the model has not seen before. The purpose of this test is to evaluate how well the model has learned and how accurately it can make predictions on unseen data. The test dataset is also labeled, but these labels are hidden from the model during prediction. By comparing the model's predictions to the actual labels in the test set, we can assess how well is generalizes beyond the training data.

2.

In [30]:
```
input_data = student.iloc[:, [1, 2, 3, 4]].values
#these are my features (index 1 to 4)-
#Age, StudyTimeWeekly, Absences and ParentalSupport

labeled_data = student.iloc[:, [6]].values
#GradeClass is index 6 so it is my labeled data
```

In [31]:
```
#this is to see if the right number of columns have been chosen
print("Shape of the whole dataset:")
print(student.shape)

print("Shape of input data:")
print(input_data.shape)

print("Shape of labeled data:")
print(labeled_data.shape)
```

```
Shape of the whole dataset:
(2098, 7)
Shape of input data:
(2098, 4)
Shape of labeled data:
(2098, 1)
```

3.

In [33]:
```
from sklearn.model_selection import train_test_split
```

In [34]:
```
#this line uses the train_test_split from the sklearn.model_selection to
#split the dataset into the required ratio
#in this case the split is 80% training data and 20% testing data
input_train, input_test, labeled_train, labeled_test = train_test_split(
    input_data,
    labeled_data,
```

```
    test_size = 0.20, #this line ensures that the testing size is 20% and
    #the remaining goes towards training data
    random_state = 0
)
```

In [35]:
```
import math

#this is to show the calculations of the testing size
print((80/100)*2098)
print((20/100)*2098)
```

```
1678.4
419.6
```

In [36]:
```
#This is to check if the shape of each data is accurate
#comparitively to the split size calculation
print("Training input shape:", input_train.shape)
print("Testing input shape:", input_test.shape)
print("Training label shape:", labeled_train.shape)
print("Testing label shape:", labeled_test.shape)
```

```
Training input shape: (1678, 4)
Testing input shape: (420, 4)
Training label shape: (1678, 1)
Testing label shape: (420, 1)
```

These dataset split sizes correspond accurately to the calcualtion done in the previous line

A3. Classification (Training)

1. Normalising the data

a.

The need to normalise data arises because features in a dataset can have different ranges, which can distort the way machine learning algorithms interpret and process them. When features with larger ranges dominate the calculations, especially in distance-based models like k-nearest Neighbours, they can overshadow smaller-range features, reducing the accuracy of the model. By normalising the data, we bring all features to comparable scale, ensuring that each one contributes proportionately to the algorithm;s performance.

b.

In [43]:
```
from sklearn.preprocessing import StandardScaler
```

In [44]:
```
#The mean and standard deviation values will be stores in the StandardScaler()
sc = StandardScaler()
input_train = sc.fit_transform(input_train)
#we only at fit_transform to training data bec
#we want it to have a mean of 0
#and a specific standard deviation
```

```
input_test = sc.transform(input_test)
#these means and standard deviations will be used for the testing
#the transform() function does that.
```

2. Using SVM to build a model

a.

Support Vecto Machines (SVM) are supervised learning methods used to tasks like classification, regression and outlier detection. They are particularly effective in high-dimensional spaces and can handle cases where the number of features exceeds the number of samples. SVMs use support vectors, which are a subset of training points, in decision-making process, making them memort effecient. They are versatile as various kernel functions can be applied to fit the data, with the option to custom kernels. However, SVMs may risk overfitting when dealing with very high-dimensional data, and they do not naturally provide probability estimates.

b.

In SVMs, a kernel is a function that transforms data into a higher-dimensional space to make it easier to classify or separate using linear boundary. This transformation allows SVM to handle complex, non-linear relationships between the data points. Kernels help find the optimal hyperplane that separates different classes by converting the data into a more separable form. Commonly used kernels include the linear kernel and polynomial kernel. Custome kernels can also be defined depending on the problem at hand. Essentially, the kernel trick allows SVM to effeciently perform the transformation without explicitly calculating the higher-dimensional space.

c.

In [51]:
```
from sklearn.svm import SVC
#this imports the support vector classifier

from sklearn.metrics import classification_report, accuracy_score
#these will help with evaluating
```

In [52]:
```
#this shows/investigates the class distribution for the testing
#and training data
unique_train, counts_train = np.unique(labeled_train, return_counts=True)
print("Training class distribution:", dict(zip(unique_train, counts_train)))

unique_test, counts_test = np.unique(labeled_test, return_counts=True)
print("Test class distribution:", dict(zip(unique_test, counts_test)))
```

```
Training class distribution: {'A': 71, 'B': 189, 'C': 287, 'D': 273, 'F': 858}
Test class distribution: {'A': 26, 'B': 52, 'C': 65, 'D': 71, 'F': 206}
```

In [53]:
```
classifier = SVC(kernel='rbf', random_state=0)
#this initalises the SVM classifier with an RBF kernel
#an RBF kernel transforms data into a higher-dimensional
#space based on the distance between data points.
```

```
#it is well-suited for non-linear problems with local patterns.

classifier.fit(input_train, labeled_train.ravel())
#this trains the SVM model on the training data
#.ravel() is used to convert labeled_train to a 1D array
```

Out[53]:

```
SVC

SVC(random_state=0)
```

3.

In [55]:

```
from sklearn.tree import DecisionTreeClassifier

#intialises and trains the DTC
classifier2 = DecisionTreeClassifier(criterion='entropy', random_state=0)
classifier2.fit(input_train, labeled_train.ravel())
#I got an error message that the DecisionTreeClassifier expects a 1D array
#for the target labels but recieved a 2D column vector instead
#ravel flattens the 2D column vector into a 1D array (which is what DTC required
```

Out[55]:

```
DecisionTreeClassifier

DecisionTreeClassifier(criterion='entropy', random_state=0)
```

A4. Classification (prediciton)

1.

In [58]:

```
#this is to make a prediciton using the SVM
labeled_prediciton = classifier.predict(input_test)
```

In [59]:

```
#this makes predicitons using the DecisionTree Model
labeled_prediction2 = classifier2.predict(input_test)
```

2.

In [61]:

```
from sklearn.metrics import confusion_matrix
```

In [62]:

```
cm_svm = confusion_matrix(labeled_test, classifier.predict(input_test))
cm_dt = confusion_matrix(labeled_test, classifier2.predict(input_test))
```

In [63]:

```
print("Confusiom Matrix for SVM Model:")
cm_svm
```

```
Confusiom Matrix for SVM Model:
```

Out[63]:

```
array([[  0,  17,   4,   1,   4],
       [  0,  15,  27,   3,   7],
       [  0,   4,  38,  13,  10],
       [  0,   0,  14,  40,  17],
       [  0,   0,   0,  10, 196]], dtype=int64)
```

In [64]:
```
print("Confusiom Matrix for DecisionTree Model:")
cm_dt
```

```
Confusiom Matrix for DecisionTree Model:
```

Out[64]:
```
array([[  8,   8,   5,   2,   3],
       [  3,  19,  16,   9,   5],
       [  6,  13,  22,  14,  10],
       [  2,   2,  14,  40,  13],
       [  3,   5,   3,  14, 181]], dtype=int64)
```

3.

The SVM confusion matrix indicates that the model is particularly effective in classifying instances within the lowest GPA (index 4, Grade F) category, where it correctly identified 196 instances. This suggests that the SVM has a strong capacity for recognizing low-performing students, which could be critical in educational assessments. However, the matrix also reveals some challenges. For instance, 17 instances were incorrectly classified as belonging to the second GPA category (index 1, Grade B), and there were additional misclassifications in higher categories, such as 4 instances misclassified as the third category (index 2, Grade C). These misclassifications suggest that while the SVM is adept at distinguishing the lower GPA classes, it may struggle with higher-performing students, potentially leading to an imbalance in its predictive accuracy across different GPA ranges.

The confusion matrix for the Decision Tree model shows a more varied distribution of predictions across GPA categories. While it successfully classified many instances, it also exhibited a higher number of misclassifications in the middle GPA categories. Specifically, it misclassified 19 instances as the second GPA category (index 1, Grade B) and 16 as the third (index 2, Grade C), indicating a lack of precision in distinguishing between these adjacent classes. However, the Decision Tree also shows some strengths, such as correctly identifying 40 instances in the fourth GPA category (index 3, Grade D) and 181 instances in the lowest GPA category (index 4, Grade F), which suggests that it has a solid grasp of the lower performance levels. This shows that both the SVM model and the Decision Tree Model have strengths in the same areas, however, the SVM model has higher accuracy levels than the Decision Tree model.

When comparing both models, the SVM exhibits a more pronounced capability in accurately identifying high-performing students, which may be particularly advantageous in contexts where recognizing struggling students is essential. The high number of correct classifications in the lower GPA category could imply that the SVM is more suitable for applications focused on predicting students who are struggling to achieve strong academic results. This capability can facilitate timely support and interventions for those who need it most. Conversely, the Decision Tree, while more prone to misclassifications in the middle ranges, may offer advantages in interpretability and the ability to capture non-linear relationships, which can be beneficial in understanding the underlying factors influencing GPA.

In summary, while both models have their merits, the SVM demonstrates superior performance in accurately identifying high GPA categories, making it the preferable choice for tasks where accurate recognition of academically poor students is critical. The Decision Tree's weaknesses in middle GPA classifications highlight a need for further refinement or additional features to improve its predictive capabilities. Therefore, the SVM model is justified as a more accurate or better performing classifier in this context.

Just a little side note/thought- We replaced the null values in Study Time Weeky with the average of that column at the very start and that may have affacted both the models and their accuracy (in a negative way). Because if most studnets whose study time wasnt recorded had an actual time lower than the mean, then teh precitors would have over-estimated their Grade. Wheres if most students whose study time wasnt recorded had an actual time higher than the mean, then the precitors would have under-estimated their Grade. Both of these cases would have caused the accuracy to go down. IF instead we just deleted those values, the the data might not have been affected as much and the predictors may do a better job predicitng the Grades.

A5. Independent Evaluation (Competition)

In [69]:
```
#this reads the new student list file and its data
student_new = pd.read_csv('Student_List_A2_Submission.csv')
```

In [70]:
```
#this allows me to see what the dataframe looks like
student_new
```

Out[70]:

	StudentID	Age	StudyTimeWeekly	Absences	ParentalSupport
0	5000	16	13.274090	27	1
1	5001	17	16.926360	6	2
2	5002	15	4.225258	15	3
3	5003	16	18.839829	17	3
4	5004	15	9.075075	6	2
...	...	...	...	...	...
156	5156	16	19.078416	15	4
157	5157	16	8.052229	24	1
158	5158	16	11.660373	27	1
159	5159	15	16.744383	8	2
160	5160	16	8.082197	16	2

161 rows × 5 columns

In [71]:
```
#this allows me to see if there are any missing values
#so I can work around them
student_new.isnull().sum()
```

Out[71]:
```
StudentID          0
Age                0
StudyTimeWeekly    0
Absences           0
ParentalSupport    0
dtype: int64
```

In [72]:
```
#this shows me the negative values.
#negative values don't make sense so I know what I need to eliminate and if
#there are values to eliminate
negative_absence = student_new[student_new['Absences'] < 0]
negative_study = student_new[student_new['StudyTimeWeekly'] < 0]
negative_support = student_new[student_new['ParentalSupport'] < 0]
negative_age = student_new[student_new['Age'] < 0]

print(negative_absence)
print(negative_study)
print(negative_support)
print(negative_age)

#these conditions are set to check if there are extreme values in the DF
#extreme values meaning they are too high (don't make sense with given context)
extreme_absence = student_new[student_new['Absences'] >= 180]
extreme_study = student_new[student_new['StudyTimeWeekly'] >= 60]
extreme_support = student_new[student_new['ParentalSupport'] >= 5]
extreme_age = student_new[student_new['Age'] >= 19]

print(extreme_absence)
print(extreme_study)
print(extreme_support)
print(extreme_age)
```

```
Empty DataFrame
Columns: [StudentID, Age, StudyTimeWeekly, Absences, ParentalSupport]
Index: []
Empty DataFrame
Columns: [StudentID, Age, StudyTimeWeekly, Absences, ParentalSupport]
Index: []
Empty DataFrame
Columns: [StudentID, Age, StudyTimeWeekly, Absences, ParentalSupport]
Index: []
Empty DataFrame
Columns: [StudentID, Age, StudyTimeWeekly, Absences, ParentalSupport]
Index: []
Empty DataFrame
Columns: [StudentID, Age, StudyTimeWeekly, Absences, ParentalSupport]
Index: []
Empty DataFrame
Columns: [StudentID, Age, StudyTimeWeekly, Absences, ParentalSupport]
Index: []
Empty DataFrame
Columns: [StudentID, Age, StudyTimeWeekly, Absences, ParentalSupport]
Index: []
Empty DataFrame
Columns: [StudentID, Age, StudyTimeWeekly, Absences, ParentalSupport]
Index: []
```

There are no values in the DataFrame that don't make sense. All the values are within appropriate range

In [74]:
```
#this step processed the data
#it extracts the relevant features (indices 1-4):
#Age, StudyTimeWeekly, Absences and ParentalSupport
input_new = student_new.iloc[:, [1, 2, 3, 4]].values
```

In [75]:
```
#this scales the new data using the previosuly fitted StandardScaler
#this uses the same scaler that was used with the training data
input_new_scaled = sc.transform(input_new)
```

In [76]:
```
#this makes predicitons using the best model that we trained before (SVM)
labeled_prediction_new= classifier.predict(input_new_scaled)
```

In [77]:
```
#this sort of prepares the results for submission like required
#it combines all the other rows with the predicted column to create a DataFrame
output = pd.DataFrame({
    'StudentID': student_new['StudentID'],
    'Age': student_new['Age'],
    'StudyTimeWeekly': student_new['StudyTimeWeekly'],
    'Absences': student_new['Absences'],
    'ParentalSupport': student_new['ParentalSupport'],
    #this extracts each of the exisitng column

    'GradeClass': labeled_prediction_new
    #uses the predicited labels
})
```

In [78]:
```
#this saved the predicitons to a CSV file
output.to_csv('Predicited_GradeClass.csv', index=False)
```