For Project #2, I made a modular system with a Data Handler, a Naive Bayes Classifier, a K-Nearest Neighbors (KNN) classifier, and a Support Vector Machine (SVM) classifier to handle different parts of handling and sorting data. The Data Handler class was made to handle reading, splitting, and preparing data. This encapsulation made sure that the code was modular and easy to update by separating the complicated data preprocessing from the main algorithm implementations. By separating the data handling, the code became more organized and simpler to handle, which will make it easier to make changes or updates in the future.

I picked the Naive Bayes classifier for classification because it is easy to use and doesn't take a lot of time to run. Naive Bayes believes that features are independent, which makes it a great method for high-dimensional data. This algorithm uses feature values to figure out the chance of each class. It is famous for how well it works in text classification jobs and other situations. I built this classifier to take advantage of its probabilistic approach, which makes decisions clear and can handle big datasets with little computer power.The KNN method was added because it uses the most common class among the nearest neighbors to make classification easy to understand. KNN is a non-parametric method that works very well with datasets that are not linear because it doesn't assume anything about how the data is distributed. KNN can adapt to complex patterns in the data by looking at the closest data points in the feature space. This makes it useful for a wide range of classification jobs. I set up KNN as a starting point to compare with more complicated models. This way, I could be sure that the classifier I chose would work best with the data. The SVM was chosen because it is good at finding the best hyperplane to divide classes, even in spaces with a lot of dimensions. By using kernel functions to change the data, SVMs can work when the data can't be separated in a straight line. This change lets the SVM find a hyperplane in a space with more dimensions that makes the gap between classes as big as possible. I chose SVM because it can deal with errors and give a clear margin of separation, both of which are very important for getting accurate classification.

I had to deal with a number of problems during the job. Taking care of lost data was necessary to avoid mistakes. When values are missing, projections can be skewed, and the model as a whole works less well. In order to fix this, I added steps to the data preprocessing process to deal with missing numbers. This meant either adding the right statistical measures (mean, median) or getting rid of the rows that were missing information. This way, I made sure the dataset was clean and ready for training the model, which decreased the chance of mistakes happening during the learning process.

Another hard part was making sure that the train-test breaks were random. A random split is necessary to keep the model from becoming too perfect and to make sure it works well with data it hasn't seen before. This was done by mixing up the information before splitting it into two parts. This eliminated any biases and made sure that the sample was representative. This step was necessary to get a fair picture of how well the model worked and to avoid getting wrong results from the way the data was ordered.

Naive Bayes needed to store values in dictionaries so that they could be quickly found and calculations could be done more easily. This was needed to figure out class probabilities and feature statistics. By putting the data in this order, I was able to quickly calculate the probabilities I needed for the forecast, which cut down on the amount of work that had to be done and made the model work better. To change Naive Bayes to work with continuous features, we had to assume a Gaussian distribution and use the Gaussian probability density function to figure out the chance. By making this change, the classifier was able to handle continuous data well, which meant it could be used for more than just category data. I could get a better idea of the chances of something happening by using Gaussian distributions to model the continuous traits. This made the classifier better at predicting the future.

The main problem for KNN was making the computations as fast as possible. It can take a long time to run KNN on big datasets because it needs to figure out the distances between each point in the dataset and the query point. To solve this problem, I used optimized distance estimates and indexing techniques, like KD-trees or ball trees, to make the work easier on the computer. Because of these changes, the KNN algorithm could handle bigger datasets better and make estimates more quickly without losing accuracy.

When using SVM, it was very important to pick the right kernel functions and tune the hyperparameters. The success of the classifier is greatly affected by the kernel function that is chosen (linear, polynomial, radial basis function, etc.). In order to find the best values, I used grid search and cross-validation. Grid search explores a range of hyperparameter values in a planned way, and cross-validation checks how well the model works on different groups of data. This set of factors made sure that the SVM was perfectly tuned to the dataset, which led to the best results possible.

Each piece of code was carefully recorded to show how it was made and what ideas were behind it. Such as, in the Data Handler class, comments were added to make it clear what each method was supposed to do, like reading data, splitting the dataset, and dividing features from labels. This paperwork helped make sure that the process was clear and that the reasoning behind each choice was easy to understand.

Functions and loops were used a lot in the design to get the job done quickly. There were functions that were made to do specific jobs, like figuring out probabilities, distances, and hyperparameters. This made sure that the program could solve the problem with as little memory and computation as possible. The program's accuracy was checked by trying it thoroughly. Any bugs that were found were recorded and fixed right away. By pointing out problems and fixing them, I made sure that the final entry was correct and free of mistakes.

Overall, the design choices I made and the organized ways I dealt with problems led to a strong and flexible classification system. The contained data handling made sure that the preparation was clean, and the mix of Naive Bayes, KNN, and SVM gave a complete way to do classification tasks. I created a model that can make accurate and reliable predictions in a wide range of situations by addressing problems like missing data, randomness in splitting, computational efficiency, and hyperparameter tuning. This methodical and modular approach not only made each classifier work better, but it also taught us a lot about their strengths and weaknesses, which will help us make changes and find new uses for them in the future.