# Algorithm Descriptions

## Naive Bayes

**Assumptions:** The classifier works under the assumption that all of the features of the dataset are normally distributed. The two datasets meet this assumption after standardization. There is also the assumption that features are independent of each other given the output label, which is central to the naive bayes theorem.

**Training:** The training consists of calculating the probabilities of each class, and the mean and variance of each feature conditional on Y=c for each class c, which are stored as attributes in the NaiveBayes object. The distributions of each feature are assumed to be normal.

**Predicting:** In order to calculate a prediction, we apply bayes theorem, P(y|x) = P(x|y) * P(y) / P(x), where y is the class label and x is the entire input sample. We operate under the assumption that features are independent of each other given the output label. This allows us to find the probability of x given y, P(x|y) = P($x_1$|y)*P($x_2$|y)*...*P($x_n$|y). The P($x_i$|y) is calculated by using a normal PDF, using the mean and variance of features given Y=c, which are stored as attributes in the NaiveBayes object. We do this for each possible y. From this, P(y|x) can be calculated for each y, so that we know for each sample x, which y is it most likely to belong to. The sample is then assigned this most likely y as the predicted label. In practice, in order to get maximal P(y|x), we do not need to calculate the P(x) - we only need to maximize P(x|y) * P(y).

**Hyperparameter selection:** Because there are no parameters inherent to naive bayes, I am using the size of the train-test split as parameters. I tested a train set of size [.5, .6, .7, .8, .9]. For dataset 1, the best result was 0.5, and for dataset 2, the best result was 0.6.

## SVM

**Training:** First, the labels in y must be converted from 0 and 1 to -1 and 1. This is necessary for future calculations.This implementation utilizes CVXOPT library in python in order to solve the following quadratic programming problem (dual problem):

$$\max_{\lambda} L(\lambda) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{n} \lambda_i \lambda_j y_i y_j \mathbf{x_i} \bullet \mathbf{x_j} \quad \text{s.t. } \lambda \geq 0$$

Because we are implementing the radial basis function (RBF) kernel, we replace dot($x_i$, $x_j$) with the kernel matrix

$$k(\mathrm{x}, \mathrm{x}') = \exp\left(-||\mathrm{x} - \mathrm{x}'||^2 / 2\sigma^2\right)$$

By solving the dual problem, we are solving the constraints of SVM classification. In order to use the CVXOPT library, the equation must be reformatted to match the form

$$\begin{aligned} &\min \frac{1}{2} x^T P x + q^T x \\ s.t. \quad &Gx \leq h \\ &Ax = b \end{aligned}$$

So that we can call: `cvxopt_solvers.qp(P, q, G, h, A, b)` to retrieve the solution of the quadratic programming problem. The support vector coefficients are then extracted from the solution, and used to calculate the weights and bias of the SVM. Weights are calculated as a linear combination of the support vectors, weighted by their corresponding Lagrange multipliers. The bias is computed with the first support vector and its associated label. These are stored in the rbf_SVM object as attributes.

**Predicting:** The following formula is used for each sample x: $z = \mathbf{x}^T\mathbf{w} + b$.
The predicted label is assigned the sign of z. X is assigned 0 for $z < 0$ and is assigned 1 for $z >= 0$.

**Hyperparameter selection:** A grid search was performed for the parameters gamma and C(lambda), for gamma in [0.01, 0.1, 1, 10, 100] and C in [0.01, 0.1, 10, 100]. The best C and gamma for dataset1 were 0.1 and 0.01, respectively. The best C and gamma for dataset 2 were 0.1 and 1.0, respectively.

## k Nearest Neighbors

**Assumptions:** One core assumption of the k nearest neighbors algorithm is that the closer two data points are, the more related and similar they are. With this, we used Euclidean distance to calculate the distance between the two data points.

**Training:** The kNN algorithm doesn't require a training step like other algorithms do, since it takes the whole dataset and finds the nearest neighbors to the test data point.

**Predicting:** To predict the value for the test data point, we find the "k" nearest neighbors of the data point, and evaluate the class labels of those neighbors. The nearest neighbors, as mentioned earlier, are calculated using Euclidean distance. Based on the majority vote of the nearest neighbors, the test data point's class label is predicted.

**Hyperparameter selection:** I used a range of different numbers of neighbors as the parameter, testing values of [1, 3, 5, 7, 9] on each dataset. Based on the highest F1-Score, that was considered the "best performing configuration" and that was the value for the number of neighbors that we used in the model.

## Decision Tree and AdaBoost

**Assumptions**: Decision trees assume that the relationship between feature values and the outcome variable can be measured through a "tree" structure of if-else decision rules. AdaBoost, on the other hand, assumes that many weak learners like decision trees can be combined to create a strong learner. Both algorithms assume that the training data represents the underlying distribution.

**Training:** Training decision trees involves recursively partitioning the dataset based on feature values, eventually reaching a class label. This process includes finding the best feature and threshold for splitting the dataset. In our implementation, the "best_split" method is used to find the lowest entropy, and using that to split the dataset. AdaBoost trains decision trees by assigning higher weights to wrongly classified samples, thereby focusing on harder to classify samples. The decision tree and AdaBoost implementations both improve performances throughout training, as decision trees improve its decision rules and AdaBoost adjusting the weights of harder to classify samples.

**Predicting:** Predicting with decision tree occurs by traversing the decision tree based on the feature values and threshold until a leaf node is reached. In our implementation, the "predict_one_sample" method is used to describe this process. If there is no "left" or "right" to the node, this means it's a leaf node, and will then return the class probabilities (in this case, the class label). Otherwise, it will continue to traverse the tree to find the leaf node. Since AdaBoost is implemented on top of the decision tree, it works in a similar way, with the addition of using multiple decision trees, and adjusting the weights based on misclassified samples (as seen in the "calc_adaboost_weight" method).

## Result Evaluation and Hyperparameter Tuning

### Data Preprocessing

**Encoding:** The only non-numerical feature across both datasets is in dataset 2, which has a single categorical column with values "Absent" or "Present". This was converted to a binary column, with value 1 for "Present" and 0 for "Absent".

**Standardization:** Sklearn's StandardScaler was used to standardize every single feature across both datasets.

**Shuffling:** When the data is loaded in, it is randomly shuffled. This is important for training and cross validation.

### 10-Fold Cross Validation and Result Evaluation

The data was split into training and testing sets with an 80-20 ratio. Then, grid search was used, where each set of hyperparameters was tested using 10-fold cross validation. The highest performing hyperparameters were then applied to a model, which is then trained on the entire training set. This model then was used to predict on the testing set, in order to obtain the final performance metrics. The final performance metrics were: accuracy, precision, recall, and f1 score.

## Performance Comparison

All models performed far better on dataset1 compared to dataset 2. Because this is consistent across all models, this likely suggests that the relationship between the features and label in dataset2 is much weaker than the relationship between the features and label in dataset1. For dataset1, all models seem to have far lower recall compared to precision. This means that all of the models are more likely to underpredict the amount of 1's in the dataset. Again, because this is consistent across all models, this suggests that there is a property in the data that causes this. For dataset2, some models have better recall than precision, and some models have better precision than recall. This is likely due to the fact that no models perform exceptionally well on the dataset, so the precision and recall of models are not consistent with one another as their architectures lead them to predict differently, based on the weakly correlated data.

Across Dataset1, KNN demonstrates superior performance, boasting the highest accuracy, precision, recall, and F1 score among all models. This suggests that KNN's ability to capture complex relationships within the data is particularly effective in Dataset1's context. Conversely, on Dataset2, KNN's performance declines notably, indicating potential sensitivity to the characteristics of the data. AdaBoost, on the other hand, performs well on dataset1, but is the best performing model on dataset2, implying that it generalizes to different data well, and is particularly suited to deal with dataset2.

**Dataset1 Performance**

| | **Accuracy** | **Precision** | **Recall** | **F1** |
|---|---|---|---|---|
| **Naive Bayes** | 94.74 | 96.97 | 88.89 | 92.75 |
| **SVM** | 94.74 | 97.37 | 88.10 | 92.50 |
| **KNN** | 97.22 | 98.85 | 93.68 | 96.13 |
| **Decision Tree** | 93.86 | 1.0 | 83.33 | 90.91 |
| **AdaBoost** | 95.25 | 96.88 | 90.40 | 93.39 |

**Dataset2 Performance**

| | **Accuracy** | **Precision** | **Recall** | **F1** |
|---|---|---|---|---|
| **Naive Bayes** | 66.49 | 51.43 | 56.25 | 53.73 |
| **SVM** | 73.12 | 65.22 | 46.87 | 54.55 |
| **KNN** | 67.14 | 53.19 | 43.04 | 47.15 |
| **Decision Tree** | 57.14 | 33.33 | 50.0 | 40.0 |
| **AdaBoost** | 75.73 | 69.29 | 52.45 | 58.65 |

## Contributions:

Jerry

- Naive Bayes
- SVM
- Preprocessing and CV

Vishwa

- KNN
- Decision Tree
- AdaBoost