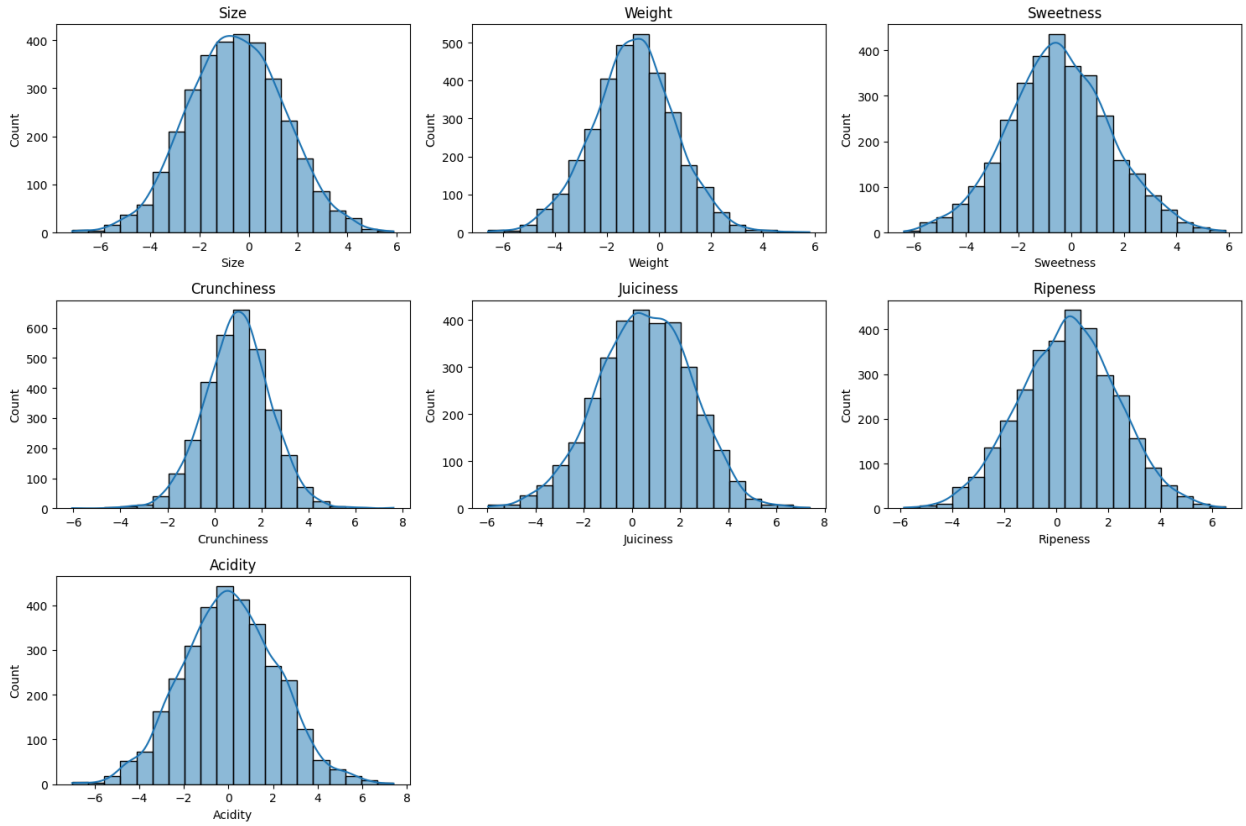
# Case Study 1

***Authors:*** *Lam Nguyen (ltn18) and Jerry Xiao (jxx419)*

## Data Preprocessing

### Standardization

We first check the distribution of all of the features of the dataset. All of the features of the dataset are normally distributed:




We see that each feature is normally distributed, and we decide to standardize the data using sklearn's standard scalar, mainly in order to proceed with principal component analysis and linear discriminant analysis for feature selection.

### Train Test Split

We chose an 80-20 train test split for our data. When splitting the dataset, we utilized the stratify property to ensure that both the training and the test set have similar distribution of the output labels.

### Feature Selection: PCA, LDA, and SFS

For feature selection, principal component analysis (PCA), linear discriminant analysis (LDA), and sequential feature selection (SFS) were all tested. For each of the three, the entire possible range for the number of selected features were also tested. For PCA and SFS, this means that 1 through 7 selected features were tested. In the case of LDA, because there are only 2 classes, there could only be 1 feature selected. Unfortunately, due to the longer runtime of SFS, we decided to continue our analysis without it, since when added to the grid search, the runtime was prohibitively high. The results of the feature selection are discussed further in the *Model Selection* section.

# Models and Hyperparameters

Logistic regression, SVM, and decision trees were all implemented through sklearn. In addition to all of the different data preprocessing steps tested, hyperparameters were also tested. All of the hyperparameters tested for each model is are outlined below:

## Logistic Regression

The type of regularization (l1, l2, elasticnet, or none) were tested. The type of regularization refers to the penalty term that is added to the loss function of the logistic regression. The L1 regularization term is proportional to the absolute values of the weights in logistic regression. This term encourages sparsity in the coefficients by shrinking some of them to zero (feature selection). The L2 regularization term is proportional to the square of the coefficients. This term penalizes large coefficients but does not typically lead to feature selection. Elasticnet regularization combines both the L1 and L2 terms.

The size of the regularization parameter C was tested with values of C in (0.001, 0.01, 0.1, 1.0, 10.0, 100.0). C is the inverse of the regularization strength, meaning that the higher the value for C, the weaker the influence of the regularization term on the bias, and the inverse is true for smaller values of C.

## Support Vector Machines

As it was the type of SVM we studied in class, we decided to use the radial basis function (RBF) kernel for SVM.

The regularization parameter C was tested with values of C in (0.001, 0.01, 0.1, 1.0, 10.0, 100.0). A smaller C allows for more misclassification in the training, encouraging a larger margin, while a larger C penalizes misclassifications more heavily, encouraging a smaller margin.

The kernel coefficient gamma was tested, with gamma being wither “scale” or “auto”. In sklearn implementation of SVM, when gamma is set to “scale”, gamma is set to 1 / (n_features * X.var()). This means that gamma is inversely proportional to the number of features and the variance of features. When gamma is set to “auto”, gamma is set to 1 / n_features, omitting the

consideration of the variance of features. Because all of our features are standardized, there should not be a significant difference between “scale” and “auto” settings of gamma.

### Decision Tree

Max depth of the decision tree was tested with values max_depth in (None, 2, 4, 6, 8, 10). A smaller max_depth can prevent overfitting by restricting complexity of the tree. A larger max_depth can lead to a more complex decision boundary, at the risk of overfitting.

Different split quality measuring criteria were tested, with criterion in (gini, entropy). These are the functions that are used to measure the quality of a node split during the formation of a decision tree. Gini employs the formula below:

$$I_G = 1 - \sum_{j=1}^{c} p_j^2$$

and entropy employs this formula:

$$I_H = -\sum_{j=1}^{c} p_j log_2(p_j)$$

Both aim to choose splits that result in more homogenous subsets.

## Hyperparameter Tuning Process

We implemented a grid search, including preprocessing methods, number of features selected from preprocessing, different models, and the different hyperparameters that pertain to each model. This whole process was pipelined, so that for each raw data parsed into the search, it will first be standardized using StandardScaler. Then it would be dimensionally reduced by either PCA or LDA with the associated n_components parameter. Lastly, it will then be fit into one of the three classifier models with its specific hyperparameters determined by the parameters grid.

Each of the mentioned parameters is discussed in further detail in the *Data Preprocessing* and *Models and Hyperparameters* sections above. In summary, here are all of the possible variables for each model:

1) **Type of feature selection and parameters:**
    a) PCA:
        i) Features selected: 1, 2, 3, 4, 5, 6, 7
    b) LDA:
        i) Features selected: 1
    c) SFS (removed)
2) **Type of model and parameters:**
    a) Logistic regression
        i) Regularization term: L1, L2, elastic-net, none
        ii) Regularization parameter C: 0.001, 0.01, 0.1, 1.0, 10.0, 100.0

b) SVM
    i) Regularization parameter C: 0.001, 0.01, 0.1, 1.0, 10.0, 100.0
    ii) Gamma: auto, scale
c) Decision trees
    i) Max depth: None, 2, 4, 6, 8, 10
    ii) Criterion: gini, entropy

Every single combination from the above list was tested by creating a parameter grid consisting of a total of 528 candidates, where each candidate is a different combination of the above variables. We fit 5 folds for each of the candidates, resulting in 2640 fits.

## Model Selection

Based on the model scoring of each combination of parameters in the tuning process, we were able to achieve the configuration that obtained the highest accuracy. The configuration being an SVM model utilizing PCA having 7 components. The parameters of the model are {C: 10.0, gamma: 'scale'}.

In this case, we can see that as n_components = 7, we are not removing any of the components in the PCA process. This signifies that all the features are needed to well explain the variance of the dataset. We thus retain all the features to fit in our model.

The 'C' parameter is to represent the regularization parameter, in which the strength of regularization is inversely proportional to C. The penalty in this case is a squared L2 penalty.

The 'gamma' parameter is set to 'scale'. When 'gamma' is set to 'scale', it uses $1 / (n\_features * X.var())$ as the value of gamma.

We were able to capture each and every combination of configurations and write them in the 'tune_output.txt' file, which contains each of the scores of each candidate. From this file, we saw that the best accuracy that we were able to obtain was 90.16%.

However, we were still skeptical about our model's performance as it is 90.16%. So we took a step further to test on the original Kaggle dataset (which contains 800 more examples than Data/train.csv). In this case, when we fit the model and scored it, we were surprised by the performance: accuracy_score: 90.75%, recall_score: 91.27%, f1_score: 90.82%, and precision_score: 90.37%. This states that our model has great generalizability on unseen data.

In conclusion, regarding the data contained in the Data/train.csv path, our SVM (C=10.0, gamma='scale') + PCA (n_components=7) achieved 90.16% in accuracy.

*Our best accuracy is located in the 'submission.txt' file.*

# Discussion

### Preprocessing

PCA consistently outperformed LDA as a feature selection method. This makes sense, as because there are only 2 output classes, LDA can have a maximum of 1 feature as output, which is likely not enough to capture the full complexity of the dataset.

Regarding PCA, we found that a higher number of features selected performed best for all models, when all else about a model was held equal. As stated in model selection, this means that all of the information contained in the features were useful in creating an output.

### Model Selection

SVM performed the best, followed by decision trees, then logistic regression. Logistic regression may have performed the worst out of the three because it is most powerful with linear relationships, suggesting that the features may not have had a linear relationship with the labels. SVM and decision trees may have been more effective in capturing the more intricate decision boundary in the data. SVM may have performed better than decision trees because SVMs inherently have regularization through the margin parameter, while decision trees are more prone to overfitting.

### Scoring Metric

We obtained a nice observation about the SVM + PCA combination in terms of using various scoring metrics in hyperparameters tuning. We saw that for accuracy, f1, and precision, the model's score stays at around 90.16%. However, if we were to use recall, the accuracy experienced a drastic drop of 50.47%. This can be explained by the fact that if we were to optimize the recall metrics, we were trying to correctly predict all the positive classes, making the model overfit our data, thus making it sensitive to negative classes.

# How To Run

### Training

To train our model, make sure that the 'Data' folder contains 'kaggle.csv' and 'train.csv'. Run the script 'python3 -m 19_classifier_apple' to start the training and evaluating process. The output will be written in files 'output.txt', 'submission.txt', and 'tune_output.txt'.

### Training

To test our model, add the data to the path 'Data/test.csv'. Use the file 'test.py' to run the model by executing the script 'python3 -m test' in the terminal. The 'test.py' file contains our model with the best parameter configuration. I have added a sample 'test.csv' file in the Data folder but you can replace it with your own 'test.csv' test data.