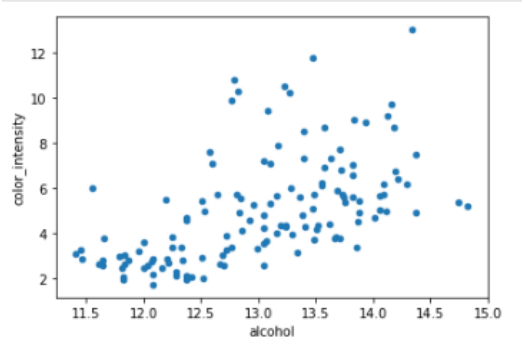
Introduction to Machine Learning: Final Paper

I. THE DATA SET, THE TARGET VARIABLE, VISUALIZATIONS AND TRANSFORMATIONS

For the three projects, I chose to work with a wine data set that is built for wine recognition from UC Irvine. I found the data set on Kaggle.com, although the original source is from the Institute of Pharmaceutical and Food Analysis and Technologies in Genoa, Italy.[1] The data set looks at three different classes of wines grown in the same region in Italy, but derived from three different cultivars. Each cultivar equates to Class 1, 2, or 3 in the data set. The data set has a mix of float or integer measures for 13 chemical elements in the wine, *including Alcohol, Malic acid, Ash, Alcalinity of ash, Magnesium, Total phenols, Flavanoids, Nonflavanoid phenols, Proanthocyanins, Color intensity, Hue, OD280/OD315 of diluted wines, and Proline.* The original data set is composed of 178 rows and 14 columns.

I chose this wine data set after exploring working with other data sets. First, I wanted to work with a data set with information on missing migrants. However, the type of prediction task that was possible with the data in the end was regression, which is not the focus of this class. I also considered working with a mushroom variety data set, however it required more complex transformations of each variable to pre-process it for machine learning. Since I am newer to Python, I decided to go with the wine data set, which is already in a solid state for running machine learning models. **The predictive y variable from the 13 wine component features is the wine class (1, 2 or 3).**

In Project 1, when I first started exploring the data set, I only changed the data type of the 'class' variable to be a category instead of an object. The data set had no missing data or null values, I used the .info() and .describe() to check variable names and for missing values. I then set the target variable (y) to be predicted equal to the 'class variable' in the data set (*y = wine[['class']]*) and set the X or features variables as equal to the rest of the predictive variables in the data set (*X = wine.drop(['class'], axis=1]*). I then split my data set into the training and test sets with the Sklearn *train_test_split* function, stratifying on y and assuring for proportionate distribution of features between the training and test sets.



Through visualizing my data with mostly histograms and scatter plots, I learned a few things. First, I was able to check the distribution of the data and check for normality. I did not detect any major outliers or bizarre data in my data set. I was able to see the relative distribution of wine types between classes, with the most values in class 2 (53), followed by class 1 (44), and class 3 (36). I was also able to explore the visual relationship between different variables. For example, running a scatter plot on my X training set and comparing alcohol vs. color intensity, there appears to be a correlation between higher color intensity and greater alcohol content.

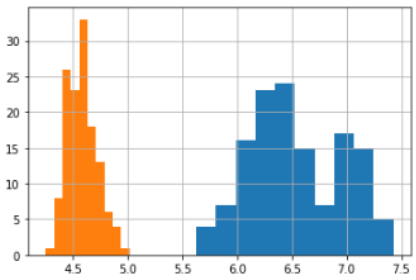
I further applied squaring, cubing, logarithmic and exponential transformations to the proline and

[1] 'Classifying wine varieties.' (2017). Kaggle.com. <https://(/www.kaggle.com/datasets/brynja/wineuci>

magnesium features in my data sets. I plotted the scatter plots and histograms of the resultant data to see how these transformations changed the distribution of my data. Applying for example the log transformation when preprocessing the data set can help to unskew the data if there are major outliers. Standardization, or making my data have a zero-mean, was not necessary on my data at this stage.

```
#PLOT THE TRANSFORMED DATA
#HISTOGRAMS
#Log
X_train["proline_log"].hist()
X_train["magnesium_log"].hist()
```

```
<AxesSubplot:>
```



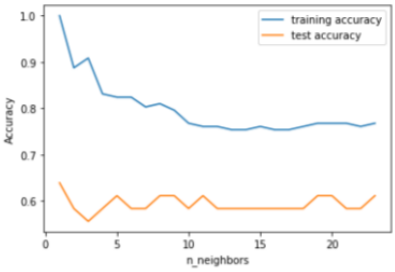
Before applying the logarithmic transformations to my data, for example, the variance of magnesium was among the highest at 192.196 and proline at 105507.02 (in their respective units). The variation (.var) after applying the log transformation was 0.1769 for proline, and 0.018 for magnesium, thus drastically reducing the variance and bringing it closer to the variance found in the other features of the data set.

II. SUPERVISED LEARNING ALGORITHMS

For the supervised learning models, I chose to do K Nearest Neighbors and Random Forest Classifier models.

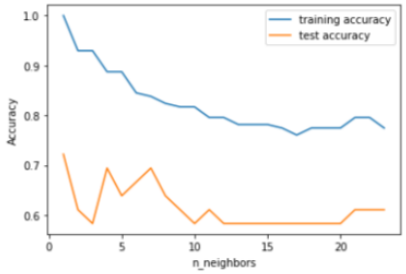
K-Nearest Neighbors

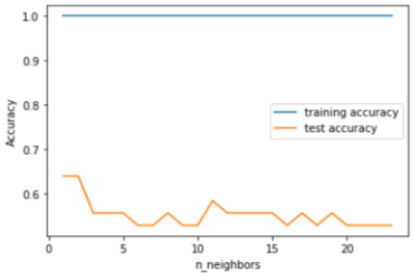
The K Nearest Neighbors is a non-parametric model that does not make assumptions about normality. The model classifies data through basing a label on the nearest data points and their assigned classes. It can be based on 1, 2, 3, or *k* neighbors depending on the adjusted parameters. The model will attempt to predict the model, or in this case 'wine class', by looking at the *k* closest labeled data points and taking a majority vote on what label the unclassified point should have.



When running this model on the wine data set, I first manually adjusted the *k* (or n_neighbors) for 1-7, and 24, finding that the most accurate knn score was for k=1, at 1.0, followed by k=3, at .908 , declining as k increased (e.g. k=24, score = 0.75). I plotted the accuracy for k/n_neighbors between 1, 24, on the training and test sets. I found that for both the accuracy was highest when k was lower in general.

I also adjusted the parameters of 'metric' to test different ways of estimating the distance between two points. Euclidean distance is the default metric, but there is also Manhattan distance, which I tested. This is seen below. As can be seen visually when comparing the two charts, the test accuracy varies a lot more for the Manhattan distance than the default.





I also adjusted the weight parameter, for weights='distance.' The distance weighting, instead of taking votes of the k-nearest neighbors closest to the new data point, weighs each vote by the distance of each of the k closest points to the new data point. Again, I ran this on n_neighbors 1-24, and found it increased training accuracy, but decreased test accuracy.

To find the optimal hyper parameters, I used cross-validation and grid search. I used K-Fold cross validation, with cv being the number of folds for the K-fold cross-validation. Grid Search uses different combinations of all the specified hyperparameters and calculates the performance of each combination to select the optimal value for the hyperparameters. Cross-validation divides the training data further into the validation data and the training data. The K-fold cross-validation divides the training data into *k* number of subsets or partitions. Each of these iterations keeps a partition for testing and the remaining for training the model. CV tells the model how many times to do these iterations. At the end, the average performance of all the iterations is given as the score.

I conducted a CV Gird Search to find the best number of neighbors from 1-25 with a five-fold cross-validation. The outcome was n_neighbors = 1, with an accuracy score of 0.78. I also did this process for weights, and distance was considered to be more optimal with a score of 0.747. I compared the results of the default model to the optimal model through calculating the precision, recall, f1-scores -- calculated through a confusion matrix. These are compared below, with the default model on the left, the optimal in the middle, and the worst parameters on the right. The precision is the ratio of tp / (tp+fp). It is the ability of the classifier to not label a sample as positive that is actually negative, with the best value being 1 and the worst being 0. The optimal parameters got the best overall precision at 0.67 (vs. 0.58 and 0.56 for the default and worst parameters, respectively). The recall score measures the model performance through counting true positives out of all actual positive values, also on a scale of 1-10. The recall was only slightly higher for the optimal parameters (at 0.64 weighted average) vs. the default parameters and the worst parameters (0.61 and 0.56 respectively). The F1 score is calculated from the mean of precision and recall. In this case, as expected, the optimal parameters had the highest f1 score at 0.64 (vs. 0.6 for default, and 0.56 for worst).

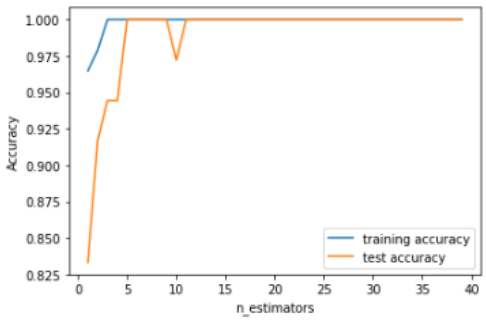
Weighted average	Default	Optimal	Least
Precision	0.58	0.67	0.56
Recall	0.60	0.64	0.56
F1 score	0.58	0.64	0.56

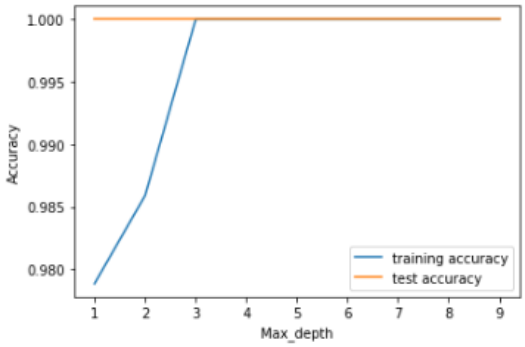
Random Forest Classifier Model

For the second supervised learning model, I chose the Random Forest Classifier (RFC) model. The RFC model works well for my data set, as the scaling is not important, and it works well with mixed data types. The model predicts classifications based on slightly varying decision trees, and then averages the predictions of the trees which create the 'forest.' This helps to reduce overfitting. This happens when a

model fits exactly to the training data, and over relies on it, so it is unable to accurately predict unseen data. When the model trains for too long of a period of time on a sample data set or when there are a lot of features, the model might pick up on non-important information or noise in the dataset, reducing its ability to generalize accurately to new data. The model merges the trees together to get a more accurate prediction. Random forest creates randomness in the model -- where it searches for the best feature within a random subset of features. The model is known to create pretty accurate results, even without tuning many hyper-parameters. The model is made up of nodes and leafs, with each leaf representing the class label or decision taken whereas the node is the test to decide each branch. The n_estimators are the number of trees in the forest -- with the general rule being that the more number of estimators the better. Maximum features is the amount of randomness, with a smaller amount reducing overfitting.

In Project 2, first I ran the RFC with the default parameters, where n_estimators = 100 and max_depth = 2, random_state =0. Then I fit the model to the X and y training sets. The accuracy score was 0.978. I then tuned and tested the model manually for different sets of estimators and depth parameters. For example, I tried out n_estimators 10, 4, and 50, and found that 10 had the best model score at 0.986 for my data, while a lower number of estimators (4) had a lower score of 0.908. Increasing the max depth (for example from 2 to 5) increased the model score to 1.0. I then tested a series of different n_estimators between 1-40 on the training and test sets, and compared them on a plot. At 11 or 12 n_estimators and above, the testing and training accuracy goes to 1.0. I also conducted this for max_depth, from 1-10. At a max_depth of 3, the accuracy on the training and test sets was also at 1.0. For max_features, the same plot rendered an optimal 1.00 score from 1-7, with only a decrease in the test accuracy at 8. In general, a smaller number helps to reduce overfitting for max_features.





I followed this with cross-validation, as explained for k-nearest neighbors in the section above. I ran the n_estimators on the range from 1-25 with a five-fold GridSearchCV, and found that the best n_estimators parameters is 15, with a best score of 0.979. For max_depth, the best parameters from the Grid Search was 24. The bias-variance tradeoff means that the more n_estimators there are, or decision trees, the more the RFC might learn the data too well. If the data is not representative, it could lead to false results and overfitting. If there is underfitting, there might be high bias and less variance in the predictions. As the model keeps learning, the bias reduces and the variance increases

I then calculated the precision, recall, and f1 scores, as described in the section above, on the default, optimal, and least performing models. Interestingly I received scores of 1.0 across all three parameter settings, showing the already high accuracy of the RFC model and potentially the simplicity of my data set to begin with.

III. PCA AND UNSUPERVISED LEARNING ALGORITHMS

Principal Component Analysis (PCA)

PCA for feature selection finds the pattern in the data set and uses this to re-format in a more-compressed format in order to make subsequent calculations more efficient. It is especially important with big data, to get rid of noise which can be an obstacle for machine learning. Dimension reduction helps bring the data features down to their most essential components. During PCA, the samples transform to have a mean of zero.

In Project 3, I first scaled the X train and test data to have the same variance. Using the PCA function, I kept all the principal components of the data set and fit it to the scaled X training set. I then calculated the explained variance ratio, which tells us for each of the principal components how much of the variance has been exploited. For dimensionality reduction, we want to explain 95% of the variance. In order to do this, I calculated how many components are needed to capture 95% of the variance for the X training and testing data sets, and transformed the data onto the PCA at 95%. The number of components was 10.

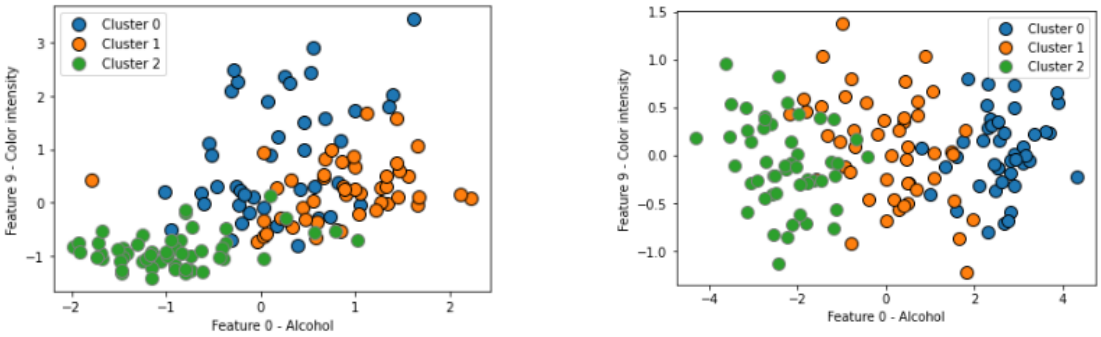
I then evaluated whether the PCA at 95% (PCA95) improved my best performing model from Project 2. I compared the Random Forest Classifier models. Indeed, the PCA improved the accuracy test sets from Project 2 (from 0.33 to 0.97), while the accuracy on the training sets stayed the same (1.0).

K-Means clustering

In project 3, I first conducted k-means clustering for pre-processing. K-means finds a specified number of clusters in the data samples. In K-means, you specify as an input into the algorithm the number of clusters you want to find. You then fit the model to the data. If new data is added, K-means can determine to which clusters the samples belong without starting anew. The clustering method works by remembering the mean of the samples in each cluster, which are known as 'centroids.' Any new sample is assigned to the cluster which has the closest mean/centroid.

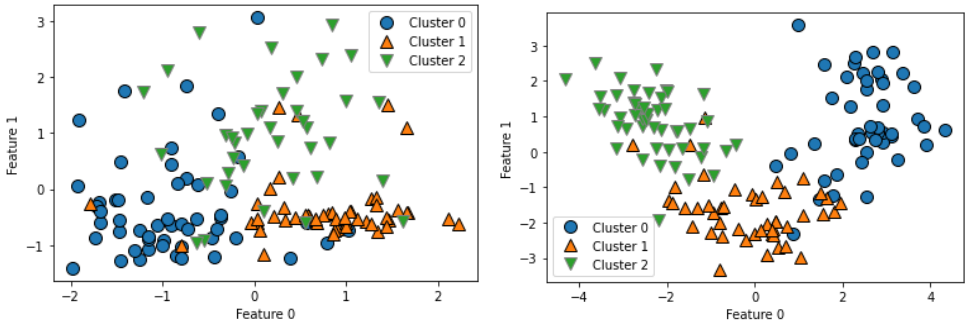
Using the elbow plot, I found that the optimal k is 3, the point after which the inertia starts to slow more slowly with each additional cluster. Inertia is how well the data set was clustered by the K-means algorithm through measuring the summed squared distance between each data point and its assigned centroid. A model that has low inertia and a low number of clusters is considered better.

I ran the model with and without PCA. Without PCA, there was more of an apparent correlation in the visualization between the features 'Alcohol' and 'Color intensity.' With PCA, when the variables were on the same scale, there appeared to be less of a relationship between color intensity and alcohol %, but cluster 2 appeared to have lower mean alcohol % overall, followed by cluster 1, and finally cluster 2. The graphs below are with and without PCA clustering.

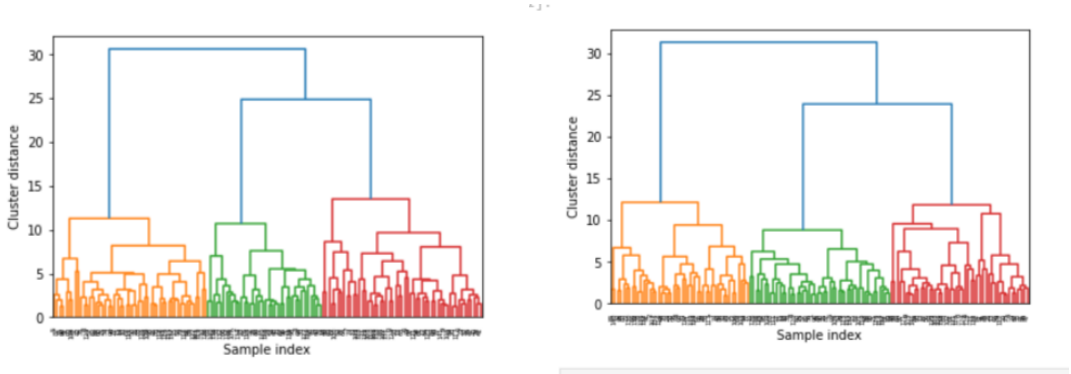


Agglomerate/hierarchical clustering

In agglomerative clustering, a cluster is made for each point which is then connected based on predefined linkage criteria. Linkages include single linkages, which uses the minimum of the distances between all observations, average linkage which uses the average of the distances of each observation, complete linkage which uses the maxim distances between all observations, and ward linkage which reduces the merged cluster variance. When looking at the clusters together, you receive hierarchical information about the relationship of the clusters. Dendrograms help to visualize this and the distance between each hierarchical clustering. First I conducted the clustering without PCA for 3 clusters. When compared to the clustering with PCA (right chart), there appears to be more variance when PCA is not used within clusters (left chart).



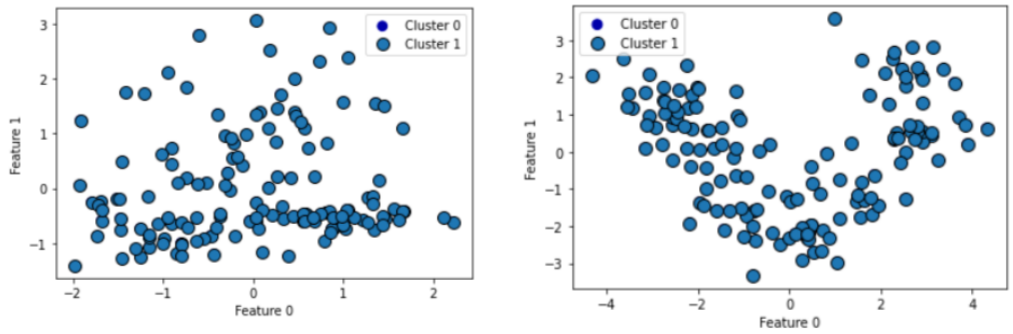
This can also be seen in the dendrograms, where the distance between each hierarchical clustering appears slightly less with PCA (on the right) than without (on the left).



DBSCAN

Unlike the aforementioned clustering techniques, during density-based spatial clustering of applications with noise (DBSCAN) you do not need to specify the number of clusters. The algorithm tries to differentiate areas that are densely and sparsely populated in the data with points between and within a cluster. It works through checking how many other points are within a certain defined distance of a selected data point. The data point is considered core if there are as many data points as the defined minimum number. Points are considered noise when they aren't clustered. The parameters include eps, which when increased gives more points for each cluster, and min_samples, which determines the minimum number of data points within the eps distance that is required to form a cluster. Increasing this results in more points being considered noise.

I applied DBSCAN with and without PCA. I played around with different min_samples and eps (min_samples=3, eps=1.0 // min_samples = 10, eps. 5). Changing these did not change the clustering significantly. Utilizing PCA definitely reduced the variance. Below, without PCA is left, and with PCA is right.



Silhouette Scores

For the three clustering algorithms, I then calculated the silhouette score, which measures on a scale of 1-10 the separation distance between clusters and shows how close each point in a certain cluster is to the points in nearby clusters. It is calculated with the mean nearest-cluster distance for a sample, and

the mean intra-cluster distance. The closer to 1 the score is the further away that cluster's samples are from those of neighboring clusters, whereas 0 indicates that a sample is very close to the decision marker between nearby clusters. Negative points indicate that the data points might have been misclassified in the wrong cluster. The overall silhouette score is the average of all these. The KMeans clustering gave the highest Silhouette Score at 0.28, with Agglomerative only slightly lower at 0.26.

IV. CONCLUSION

Overall, I learned a solid foundation for an introduction to machine learning with Python and classification models. I learned what types of data sets are useful to answer certain types of classification prediction questions, and which are better for regression. I learned how to use Python to clean and visualize the data set, and make it ready to be split into training and test sets and run on the various models. I learned about the importance of scaling and transforming data. I learned the fundamentals of supervised and unsupervised machine learning algorithms for classification. I learned about several different machine learning models, their requirements, applications, strengths, and weaknesses. I learned how to assess and evaluate the performance of various models against my data set, learning about how to adjust different hyper-parameters to see the effect they have on overall accuracy and performance. I also learned key metrics and indicators that can be used to determine a model's performance (precision, recall, etc.)

Overall, my data set worked decently and was appropriate for an introduction to machine learning. It was complete with no missing data, and clearly labeled from the beginning, so it did not present any major challenges. If I were to re-do the projects and have more time, I would work with an alternative data set of interest and try to ask more interesting/insightful questions that could be beneficial and useful professionally. I would have worked with a data set that had more distinct and labeled classes that are easier to conceptually understand the differences between.