Mini-project1
The Mushrooms dataset.

Made by:

Mateus Jose

Anass Abounouh

Crystal McCay

Helmi Ben Khalifa

Content

Abstract

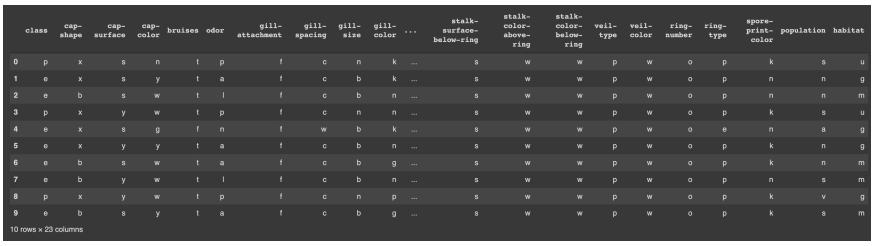
Machine Learning models are used to predict data of interest using known data as input. For mushrooms, it is useful to be able to predict if a mushroom will be edible or poisonous. Classifiers are the Machine Learning resources for said predictions. The goal of this study is to test several classifiers for predicting mushrooms poisonousness (either poisonous or edible), show their performance and optimize them, in order to choose the best of them.

1 Introduction

Separating edible from poisonous mushrooms requires meticulous attention to detail; there is no single trait by which all toxic mushrooms can be identified, nor one by which all edible mushrooms can be identified. People who collect mushrooms for consumption are known as mycophagists, and the act of collecting them for such is known as mushroom hunting, or simply "mushrooming". That is the purpose of our study using machine learning algorithms and techniques to define whether the mushrooms are safe to eat (aka edible) or not (aka poisonous). We worked on a dataset found on Kaggle that contains approximately about 23 features for each mushroom, and about more than 8000 mushrooms.

2 Exploratory Data Analysis

The dataset used in this project is a mushrooms dataset available on Kaggle that contains 8124 instances of mushrooms with 23 features like cap-shape, cap-surface, cap-color, bruises, odor and so on. First thing we have to do is to use Pandas to read our CSV file. Once this is done we can finally visualize our dataset but as the number of rows is big, we will limit ourselves to 10 rows.



	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	population	habitat
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p	k	s	u
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p	w	o	p	n	n	g
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	n	m
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p	k	s	u
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e	n	a	g
5	e	x	y	y	t	a	f	c	b	n	...	s	w	w	p	w	o	p	k	n	g
6	e	b	s	w	t	a	f	c	b	g	...	s	w	w	p	w	o	p	k	n	m
7	e	b	y	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	s	m
8	p	x	y	w	t	p	f	c	n	p	...	s	w	w	p	w	o	p	k	v	g
9	e	b	s	y	t	a	f	c	b	g	...	s	w	w	p	w	o	p	k	s	m

10 rows × 23 columns

Figure 1: mushroom dataset

Attribute Information:

1. **cap-shape**: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
2. **cap-surface**: fibrous=f,grooves=g,scaly=y,smooth=s
3. **cap-color**: brown=n,buff=b,cinnamon=c,gray=g,green=r, pink=p,purple=u, red=e, white=w,yellow=y
4. **bruises?**: bruises=t,no=f
5. **odor**: almond=a,anise=l,creosote=c,fishy=y,foul=f, musty=m,none=n,pungent=p,spicy=s
6. **gill-attachment**: attached=a,descending=d,free=f,notched=n
7. **gill-spacing**: close=c,crowded=w,distant=d
8. **gill-size**: broad=b,narrow=n

9. **gill-color**: black=k,brown=n,buff=b,chocolate=h,gray=g, green=r,orange=o,pink=p, purple=u,red=e, white=w,yellow=y
10. **stalk-shape**: enlarging=e,tapering=t
11. **stalk-root**: bulbous=b,club=c,cup=u,equal=e, rhizomorphs=z,rooted=r,missing=?
12. **stalk-surface-above-ring**: fibrous=f,scaly=y,silky=k,smooth=s
13. **stalk-surface-below-ring**: fibrous=f,scaly=y,silky=k,smooth=s
14. **stalk-color-above-ring**: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p, red=e,white=w,yellow=y
15. **stalk-color-below-ring**: brown=n,buff=b,cinnamon=c,gray=g,orange=o, pink=p, red=e,white=w,yellow=y
16. **veil-type**: partial=p,universal=u
17. **veil-color**: brown=n,orange=o,white=w,yellow=y
18. **ring-number**: none=n,one=o,two=t
19. **ring-type**: cobwebby=c,evanescent=e,flaring=f,large=l, none=n,pendant=p,sheathing=s,zone=z
20. **spore-print-color**: black=k,brown=n,buff=b,chocolate=h,green=r, orange=o, purple=u,white=w,yellow=y
21. **population**: abundant=a,clustered=c,numerous=n, scattered=s,several=v,solitary=y
22. **habitat**: grasses=g,leaves=l,meadows=m,paths=p, urban=u,waste=w,woods=d

2.1 Feature Overview

Now that we have a summary of the dataset's attributes and symbol definitions, we can begin to visualize our data.

Because our problem enlists a classification task, we will consider the categories into which our mushroom instances will be classified. Based on various analyses of pertinent attributes, we will determine whether the mushroom instance is poisonous or edible. We have affirmed the absence of duplicates and omissions from our data, revealing that there are 8124 unique mushroom instances for us to evaluate. Below is a visualization of the categories into which they fall.

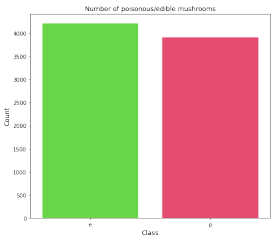


Figure 2: Data Analysis: Edible vs. Poisonous

We can see that the split is roughly into halves. We can be sure that the training process will not be adulterated by overexposure to either classification category because there is a reasonable balance. It will be the job of our prediction model to thoroughly evaluate the relationship of each mushroom's attributes to whether it is poisonous or edible.

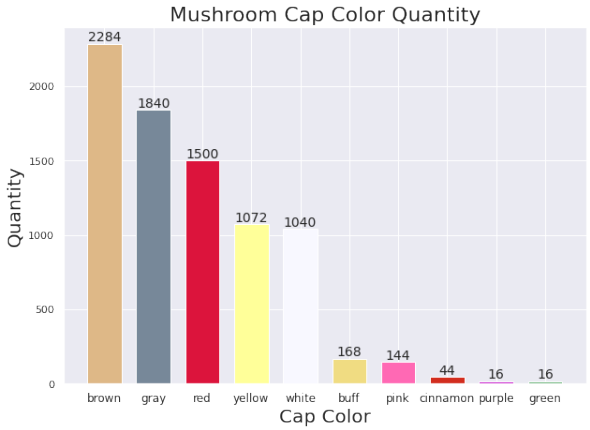


Figure 3: Feature Distribution: Cap Color

We can see that there is some appreciable variety of color in the sample population. Very common cap colors appear to be brown, gray, and red, while moderately common colors are yellow and white. The least common colors in the sample are pink, purple, and green. Below follows a juxtaposition of cap color against edible or poisonous status.

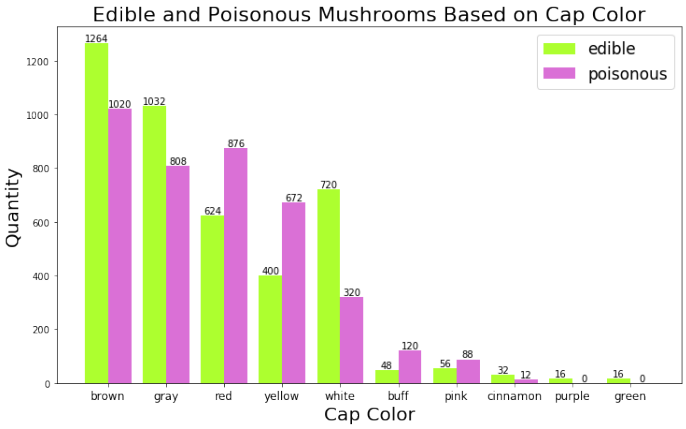


Figure 4: Cap Color and Edibility

All of the colors of mushrooms represent both poisonous and edible instances except purple and green, for which all instances were edible. The distinguishing factor in this graph is whether more instances of a given color were poisonous than were edible. For brown, gray, and white mushrooms, noticeably more instances were edible than were poisonous, whereas for red, yellow, and pink mushrooms, more of them were poisonous than were edible.
The next attribute of the mushrooms we visualize is the odor. There is an appreciable variety of odors in the sample population, ranging from neutral and aromatic smells to stronger, more offensive ones.

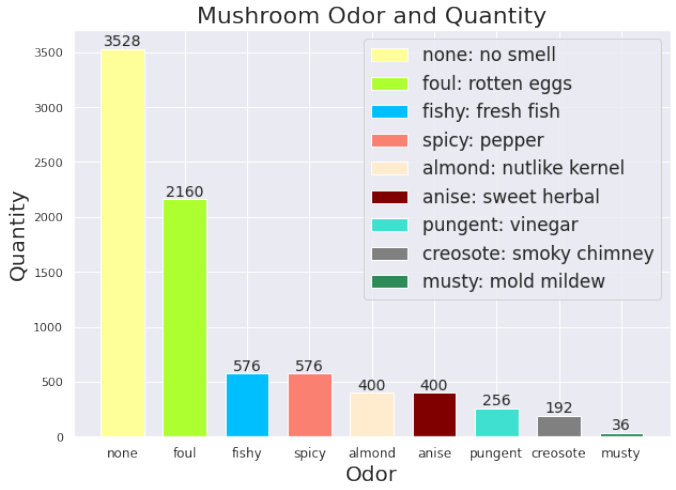


Figure 5: Mushroom odor distribution

We can see that the two most populated categories were neutral and foul-smelling mushrooms, with the other categories of more nuanced scents each containing a decently split quantity of mushrooms.

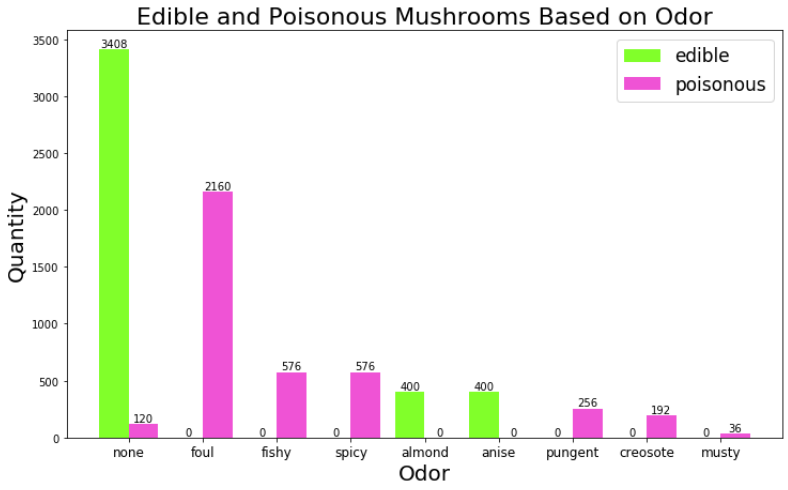


Figure 6: Mushroom odor vs. edibility

This bar graph illustrates the count of mushrooms from each odor category that was poisonous, and displays this quantity against the count from that odor category that was edible. From

the neutral, almond, and anise categories, all or almost all of the mushrooms were edible. However, from strong scent categories like fishy, spicy, and foul, almost all of the samples from that category were poisonous.

Because our dataset contains many attributes, we will apply a technique to aid in identifying the most consequential attributes in predicting the outcome. From here we will isolate a small group of highly impactful features for the prediction of edibility. This will maximize our efficiency both in the visualization process, as well as in model performance. This identification and selection occurs in our data preparation step. We will find the few features that have the biggest influence on predicting edibility or toxicity, and we will base our model training process on these features. The process of our feature selection and data preparation is outlined in the following section.

3 Data preparation

Machine learning models require all input and output variables to be numeric.
This means that if the data contains categorical features, we must encode it to numbers before we can fit and evaluate a model or even before selecting the useful features we will work with. The data is categorical so we'll use **LabelEncoder** to convert it to ordinal. **LabelEncoder** converts each value in a column to a number. This approach requires the category column to be of **'category'** datatype. By default, a non-numerical column is of **'object'** datatype. From the **df.describe()** method, we saw that our columns are of **'object'** datatype. So we will have to change the type to **'category'** before using this approach.
The category data type in pandas is a hybrid data type. It looks and behaves like a string in many instances but internally is represented by an array of integers. This allows the data to be sorted in a custom order and to more efficiently store the data.
That is why, it is a good idea to do this change first before encoding the data.

```
[ ] #categories instead of objects
    df = df.astype('category')
    df.dtypes
```

```
class                        category
cap-shape                    category
cap-surface                  category
cap-color                    category
bruises                      category
odor                         category
gill-attachment              category
gill-spacing                 category
gill-size                    category
gill-color                   category
stalk-shape                  category
stalk-root                   category
stalk-surface-above-ring     category
stalk-surface-below-ring     category
stalk-color-above-ring       category
stalk-color-below-ring       category
veil-type                    category
veil-color                   category
ring-number                  category
ring-type                    category
spore-print-color            category
population                   category
habitat                      category
dtype: object
```

Figure 7: object to category

As we can see, our columns are now of type **'category'**. We can now use **LabelEncoder** to convert categorical values to ordinal and finally map each feature attribute to numerical values. The next figure is what we finally have after the encoding.

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	population	habitat
0	1	5	2	4	1	6	1	0	1	4	...	2	7	7	0	2	1	4	2	3	5
1	0	5	2	9	1	0	1	0	0	4	...	2	7	7	0	2	1	4	3	2	1
2	0	0	2	8	1	3	1	0	0	5	...	2	7	7	0	2	1	4	3	2	3
3	1	5	3	8	1	6	1	0	1	5	...	2	7	7	0	2	1	4	2	3	5
4	0	5	2	3	0	5	1	1	0	4	...	2	7	7	0	2	1	0	3	0	1

5 rows × 23 columns

Figure 8: encoded mushroom dataset

Now we see that all the column values are converted to ordinal and there are no categorical values left.
The column **veil-type** is always null so it will never contribute to the results we want to predict. This column has to be dropped. We will then try to keep eliminating more features to be left with a reasonable number of features to start our tests on different machine learning models.
The best way to start is to look at the correlation between those features to start eliminating different features.

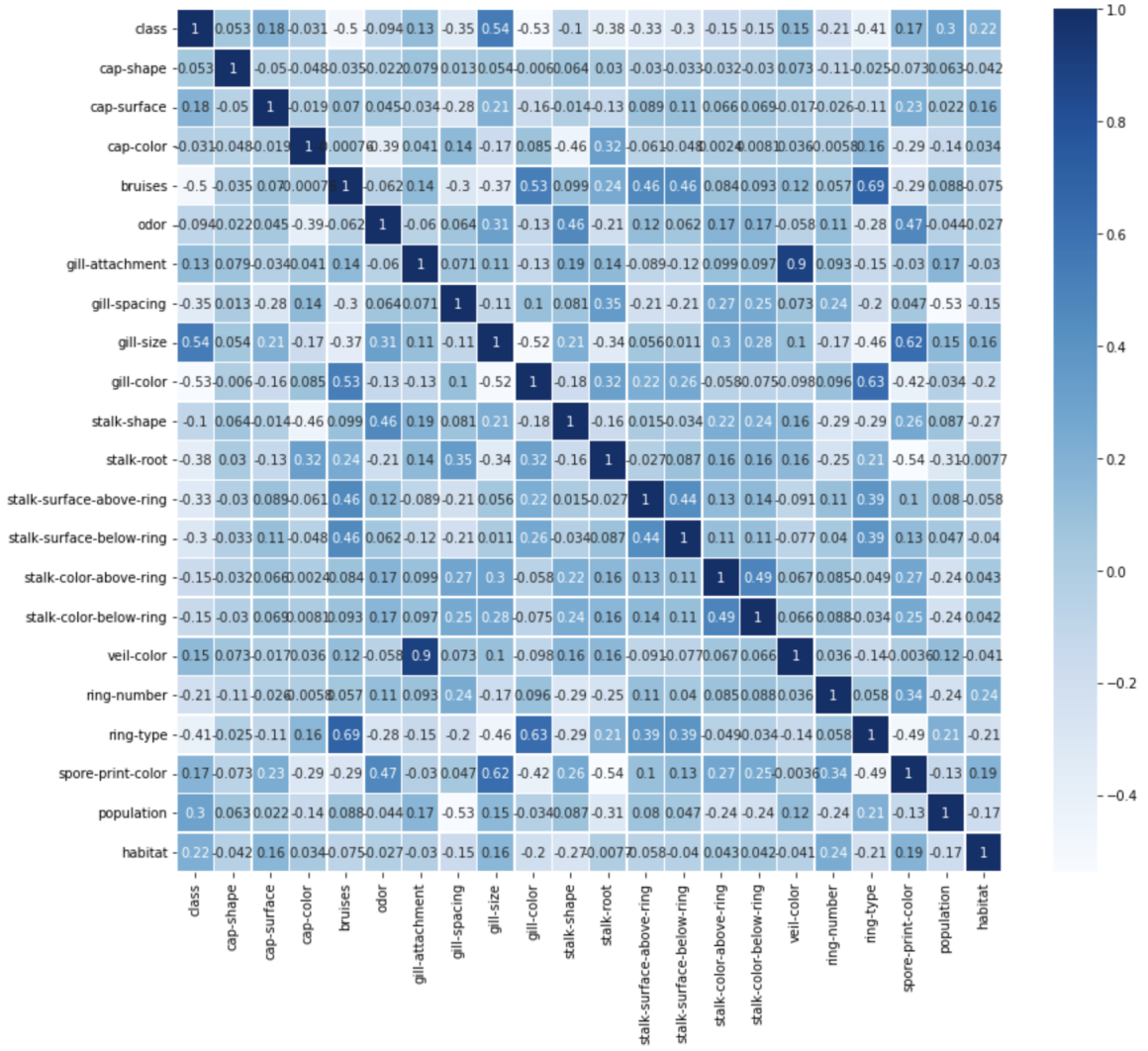


Figure 9: Correlation between all the features

We will have first to look at the different correlations between our features and the class which is the target. Maybe later, after choosing the best features for the study, we will have to look again at the inter-correlation between them.
Let's take a closer look at the first lign of this figure.

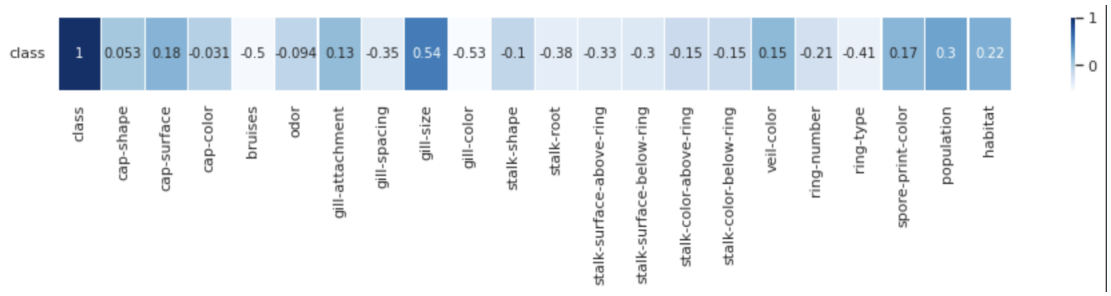


Figure 10: Correlation between features and the class

We will try now to define based on a method named **target-correlation(dataset, threshold)**, the features that are not that correlated with the target class, and we will eliminate them first. Our **target-correlation** as defined takes two parameters: dataset which is our data frame **df** we read with Pandas library, and a threshold that every feature will have to challenge. After applying this method to our dataset with a treshold of 0.3, we will drop 14 features (we could've chosen to work directly with the visuals provided by figure 8 but it is better this way). And those features are:

- 'population'
- 'gill-attachment'
- 'ring-number'
- 'stalk-color-above-ring'
- 'odor'
- 'cap-color'
- 'stalk-color-below-ring'
- 'stalk-shape'
- 'veil-color'
- 'spore-print-color'
- 'cap-surface'
- 'habitat'
- 'cap-shape'

- 'stalk-surface-below-ring'

Let's check the new shape of the dataset:

```
[ ] df_corr = df.drop(non_corr_features_target,axis=1)

[ ] df_corr.shape

    (8124, 8)
```

Figure 11: new shape of the dataset

Consequently, the new correlation matrix (which will be smaller this time) looks like this:

Figure 12: correlation between selected features

A number of 8 features is very good compared to the 23 we had at the beginning. We checked about null values and unknown values. We removed already null data but we didn't find any unknown values. Our dataset is ready for training.

4 Model tuning

Model tuning is also known as hyperparameter optimization. Hyperparameters are variables that control the training process. These are configuration variables that do not change during a Model training job. Model tuning provides optimized values for hyperparameters, which maximize your model's predictive accuracy.

4.1 Dataset split

Data splitting is when data is divided into two or more subsets. Typically, with a two-part split, one part is used to evaluate or test the data and the other to train the model. Data splitting is an important aspect of data science, particularly for creating models based on data. Here, we will be splitting our data to 0.80 for training and 0.20 for testing. After doing so, we can visualize the new shapes of the mini new datasets in the next figure:

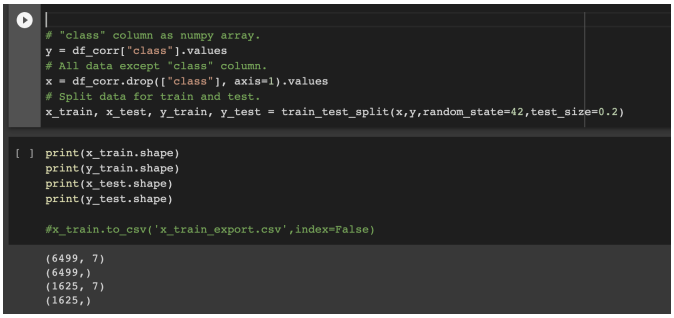


Figure 13: data split train/test

4.2 KNN

4.2.1 Presentation

KNearestNeighbors algorithm works by holding instances of training data and classifying by issuing a majority vote across "k" nearest neighbor of each point as to which class it belongs.

4.2.2 Defining Parameters

The representative parameter for KNN is n neighbors: The number of neighboring instances taking part in the classification vote.
In that case, we need the complexity curve to choose a good value of k which is the neighbors.

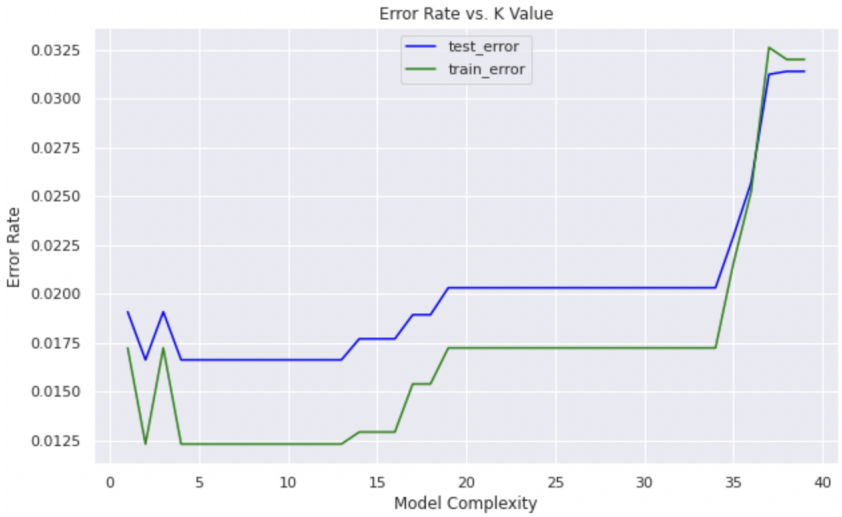


Figure 14: complexity curve

It seems based on the curve that 2 may be a good fit for our model. We found later that another parameter was slightly better than that. We couldn't tell the difference based on that curve (figure 9).
We were able to perform a score of 98.34% with a neighbor value of 9.

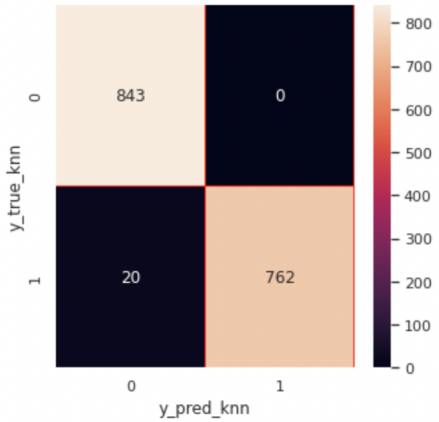


Figure 15: KNN confusion matrix

We can see that there were 762 true positive and 843 true negative. negative predictions. In total 1605 correct predictions as opposed to 20 false predictions amounting to $1585/1605 = 0.98$ accuracy.

4.2.3 Model Evaluation

Using the parameter number of neighbors=9, we plot the learning curve showing test and validation scores:

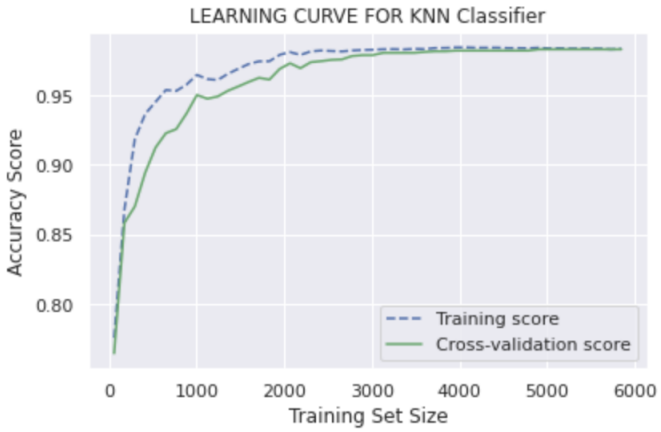


Figure 16: learning curve

4.3 SVM

4.3.1 Presentation

Support vector machines **(SVMs)** are a set of supervised learning methods used for classification, regression and outliers detection.

4.3.2 Defining Parameters

On this model, we have 3 parameters: C, kernel, degree and gamma. We picked the C parameter and did a complexity curve based on it to achieve the best precision possible.

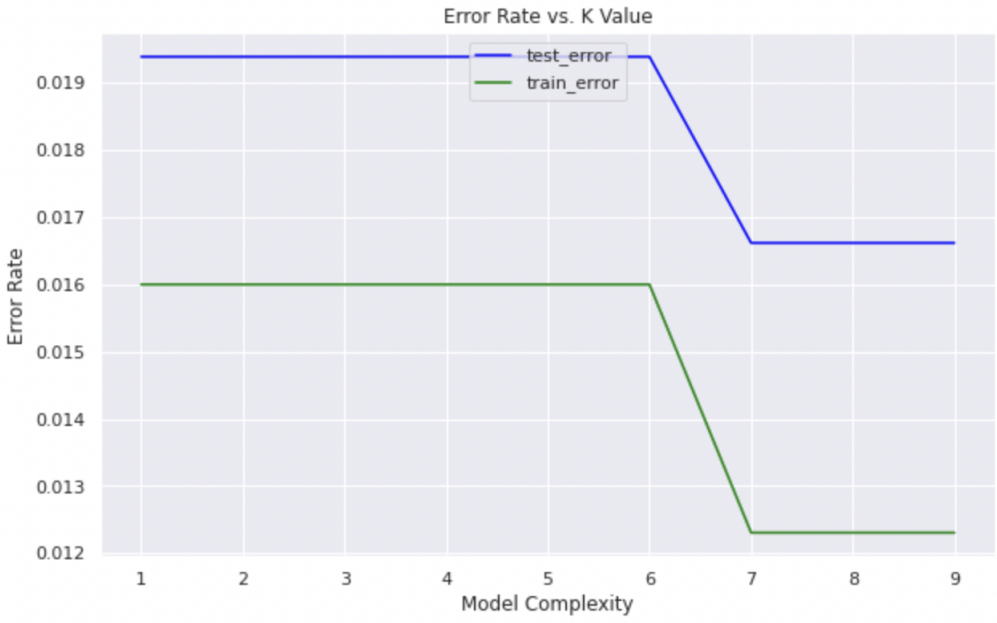


Figure 17: Complexity curve SVM

It is clear that the best parameter for C in that case is 7.
We achieved a precision of 98.77% with that value of C. Let's look at the confusion matrix:

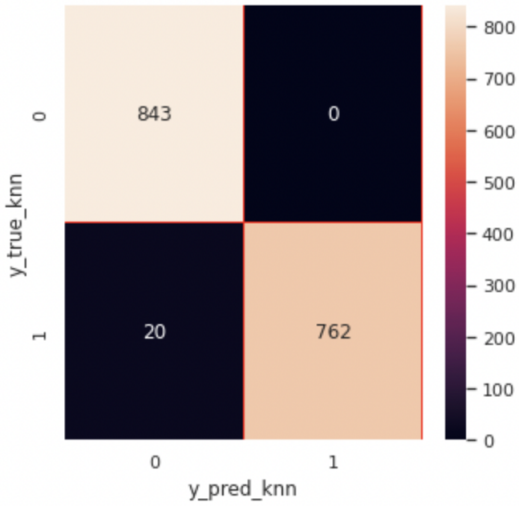


Figure 18: SVM confusion matrix

4.3.3 Model Evaluation

Chosen the parameter: (C=7), a learning curve shows us the training and validation scores for different data sizes.
This way, we are able to say that the score is bounded above 98%, and the model doesn't seem to continue learning after 2500/3000 training rows.

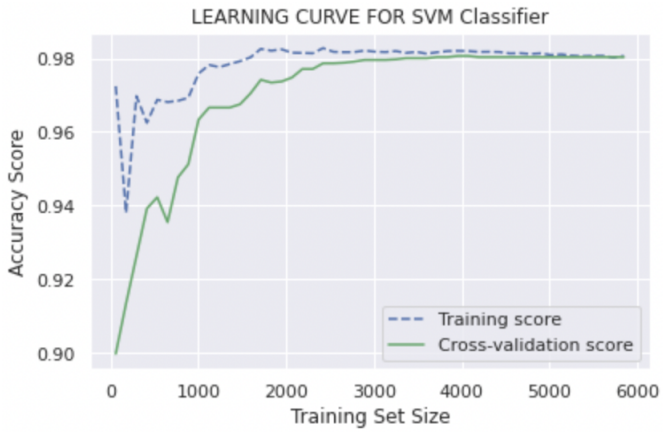


Figure 19: SVM learning curve

4.4 Decision Tree

4.4.1 Presentation

The decision tree is an algorithm that can be used for machine learning tasks like classification and regression. This is also the building block of the random forest. The algorithm determines decision boundaries, which are used to make predictions.

4.4.2 Defining Parameters

The problem faced by decision trees is an inclination to overfit data when the depth is too high. Contrarily, the tendency is to underfit when the depth is too shallow. Our task requires the determination of an ideal depth, at which the minimization of error is greatest, but the complexity of the model is minimal.

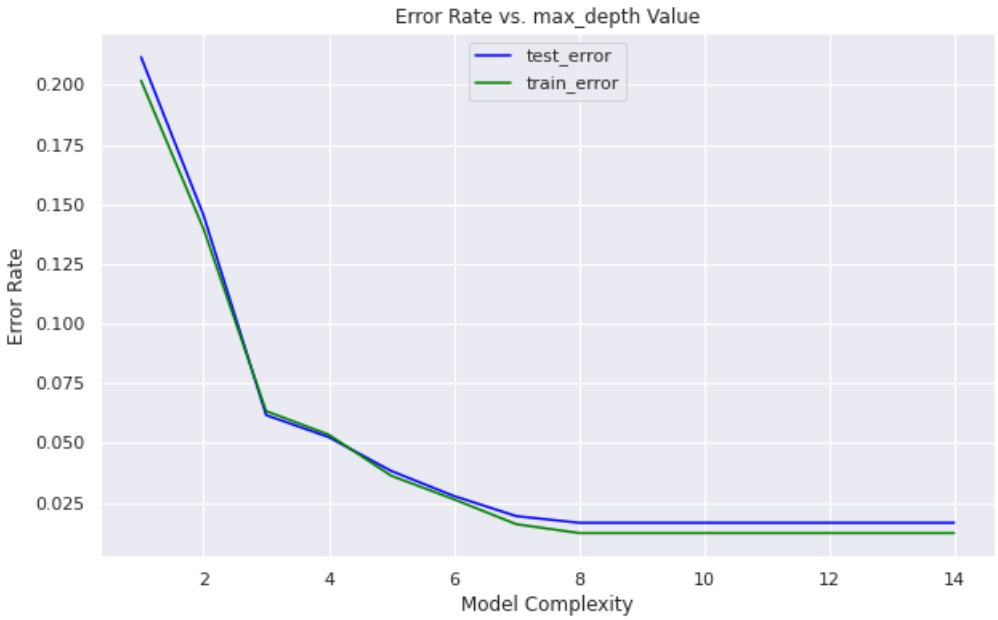


Figure 20: DT error rate vs. model complexity

The confusion matrix for the Decision Tree model is highlighted below. We see a count of true positives, true negatives, false positives, and false negatives. We see that with the decision tree, only 20 instances were incorrectly predicted.

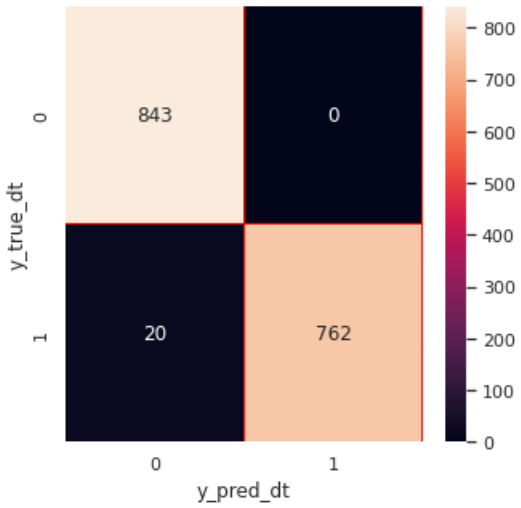


Figure 21: DT confusion matrix

4.4.3 Model Evaluation

The accuracy of the decision tree is 98.77 percent. We can see the principle of diminishing marginal returns present as we continue to increase the model complexity. At first, the accuracy of the model considerably improves with each tier of complexity that is added. However, after a certain mark, particularly after tier 7 or 8, this improvement in accuracy almost completely flattens, and additional complexity no longer enhances the problem. This helps us to visualize the number of levels that will strike an ideal balance for our decision tree.

The learning curve for this model shows a large AOC area, illustrating that the performance of the predictor significantly exceeds the y = x baseline.

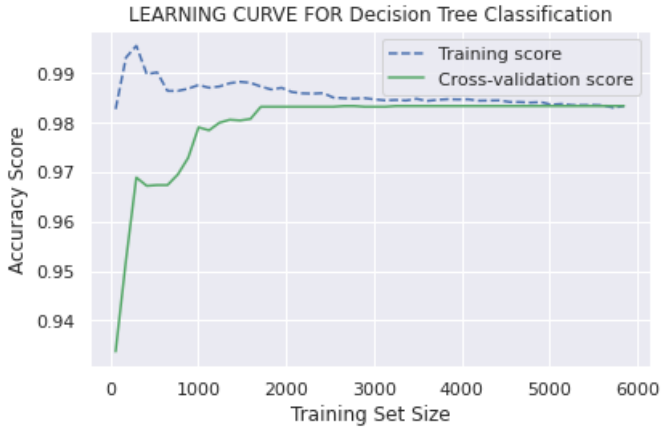


Figure 22: DT learning curve

4.5 Neural Networks

4.5.1 Presentation

Neural Networks (or Multilayer Perceptrons) are a model of supervised learning, represented by composed non-linear functions.

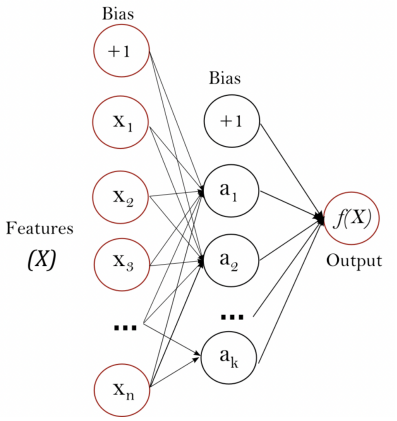


Figure 23: One hidden layer MLP

Class MLPClassifier implements a multi-layer perceptron (MLP) algorithm that trains using Backpropagation.
MLP trains on two arrays: array X of size (n_samples, n_features), which holds the training samples represented as floating point feature vectors; and array y of size (n_samples)

4.5.2 Defining Parameters

The parameter we chose to work with is the alpha parameter. Let's work on the complexity curve to try to find the best alpha parameter for our study.

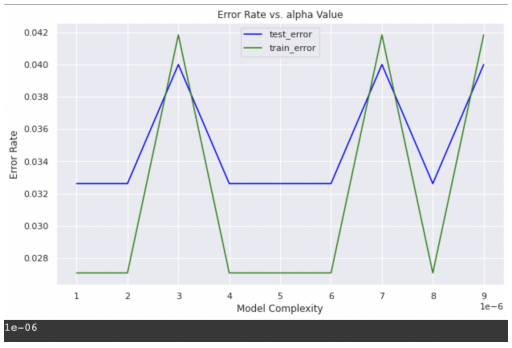


Figure 24: NN Complexity curve

As shown here, the value of alpha should be 1e-06 (we actually printed the optimal value for that).
The confusion matrix for the Neural Network model is highlighted below. We see a count of

true positives, true negatives, false positives, and false negatives. We see that with the Neural Network, 68 instances were incorrectly predicted.

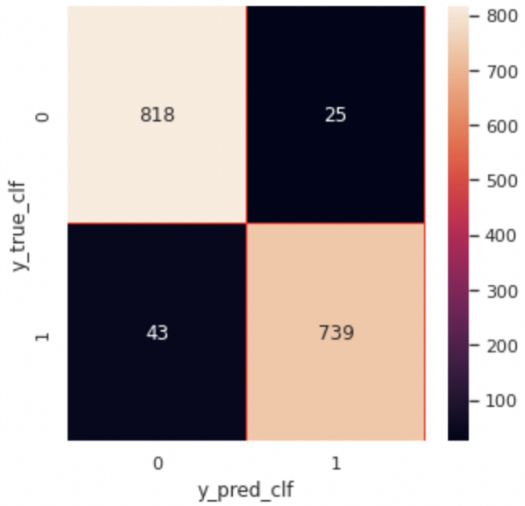


Figure 25: NN confusion matrix

4.5.3 Model Evaluation

The accuracy of the NN model reached 97.29% using the right alpha (we just had to look on the complexity curve to find it). We run some experiments on this model and the fact that we chose a value of alpha=1e-06 makes the prediction more accurate.
The learning curve for this model illustrate that the model doesn't seem to continue learning after 1100-1200 rows.

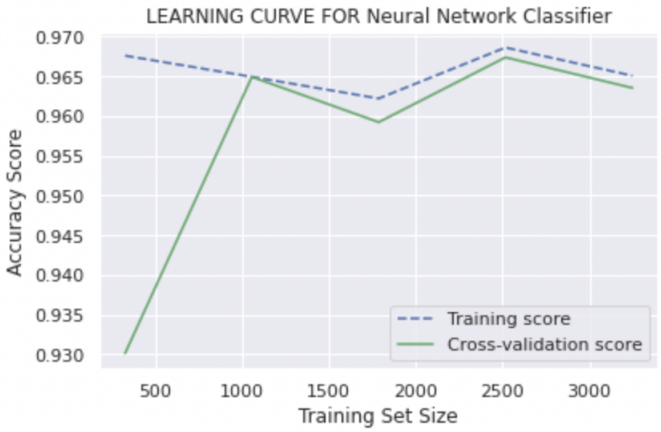


Figure 26: NN learning curve

4.6 Naive Bayes

4.6.1 Presentation

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n:

$$p\left(y|x_1,...,x_n\right) = \frac{p\left(x_1,...,x_n|y\right)p\left(y\right)}{p\left(x_1,...,x_n\right)}$$

Depending on the independence between every pair, we are left with:

$$p\left(x_i|y,x_1,...,x_i-1,x_i+1,...,x_n\right) = p\left(x_i|y\right)$$

In our case, we worked with the Gaussian Naive Bayes, or **GaussianNB** which implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right)$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

4.6.2 Defining Parameters

In our case, the parameter that we chose was **"var_smoothing"**. This complexity curve shows us what is the best value of this parameter that will give us the best performance for our model:

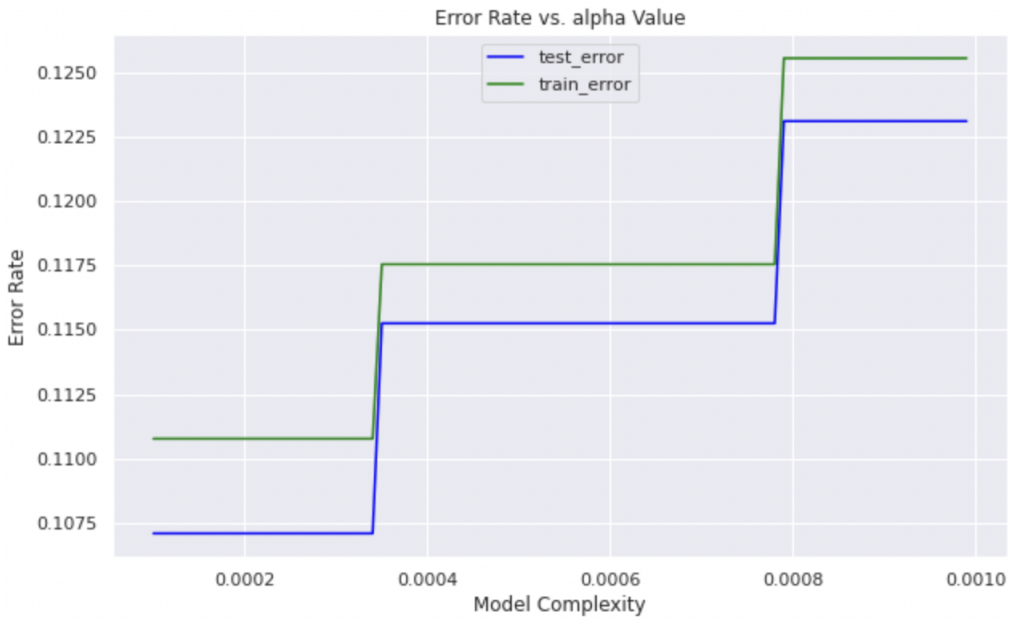


Figure 27: Naive Bayes complexity curve

The best value should be between 0.0003 and 0.0004.
The confusion matrix for the Naive Bayes model is highlighted below. We see a count of true positives, true negatives, false positives, and false negatives. We see that with the Naive Bayes, 180 instances were incorrectly predicted.

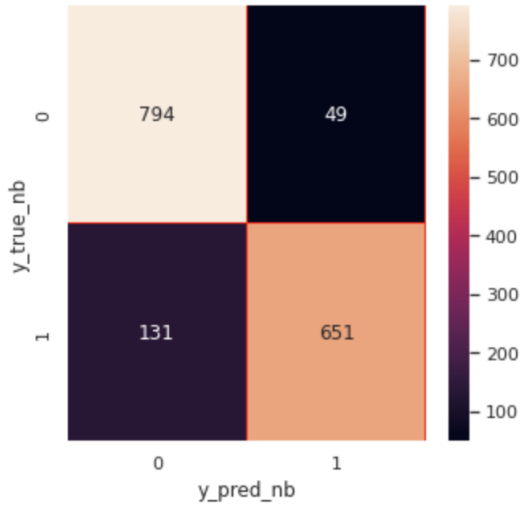


Figure 28: Naive Bayes confusion matrix

We were able to perform a score of 88.92% with a smoothing of any value between 0.0001 and 0.0002.

4.6.3 Model Evaluation

Chosen the parameter: (var_smoothing=0.0001), a learning curve shows us the training and validation scores for different data sizes.
This way, we are able to say that the score is bounded at about above 88%-89%, and the model doesn't seem to continue learning after 1500/2000 training rows.

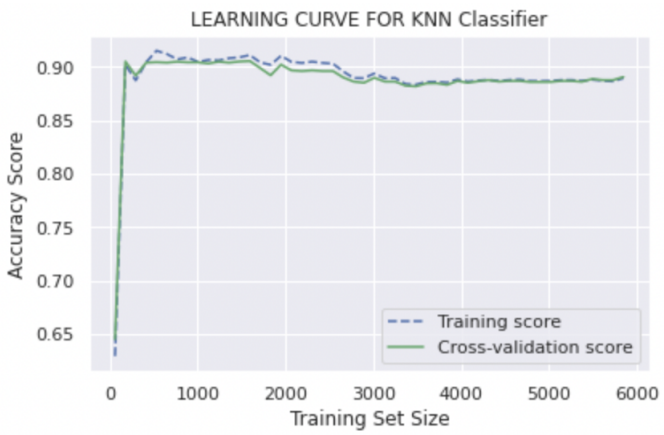


Figure 29: Naive Bayes learning curve

4.7 Random Forest

4.7.1 Presentation

The random forest utilizes the principle of the collective. By randomizing tree generation and applying a voting mechanism, an aggregate of predictors achieves a better prediction accuracy than individual predictors alone.

4.7.2 Defining Parameters

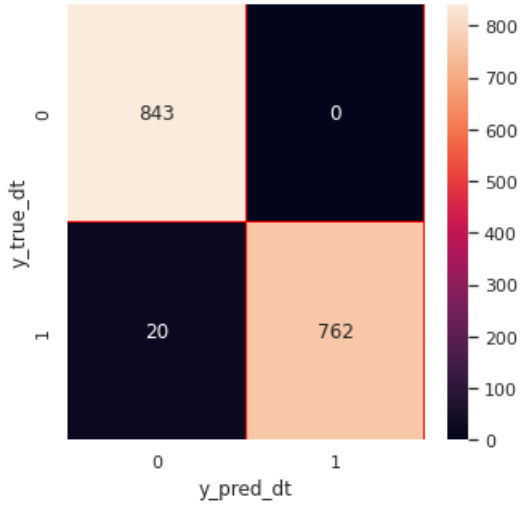


Figure 30: Random forest: confusion matrix

The random forest has an accuracy of 98.77 percent. Only 20 instances were incorrectly predicted. Interestingly, we seem to have a similar level of performance from the decision tree as the random forest.

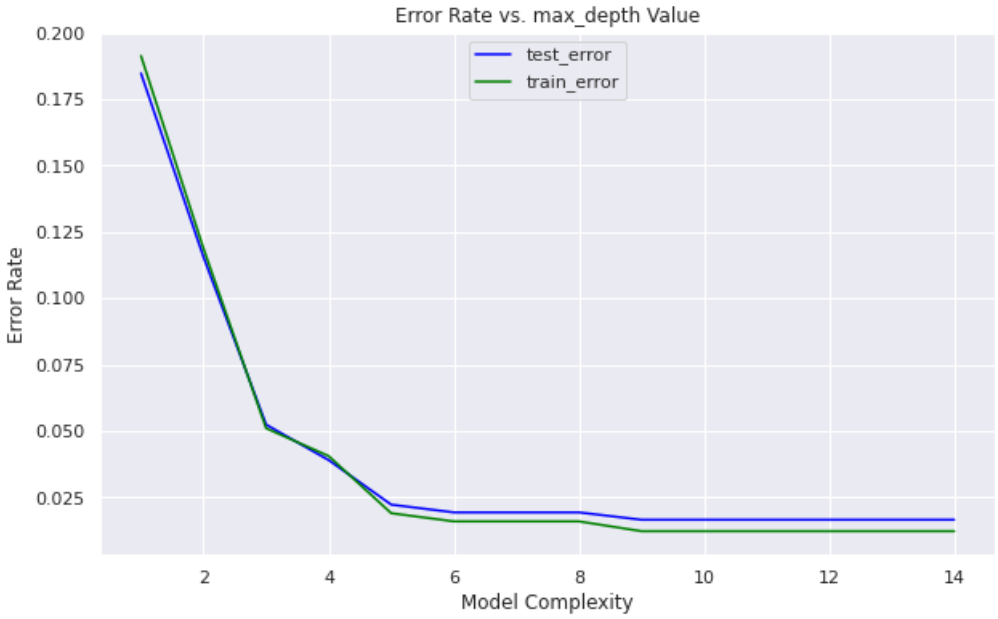


Figure 31: Random forest: error rate vs. complexity

This graph illustrates the complexity vs. the error reduction for the random forest. The progression is visually very similar to that of the decision tree. There is a desirable drop in

the error rate for the first few reductions in simplicity, but after about the eighth tier, the reductions do not continue to occur, while undesirable increases in complexity continue.

4.7.3 Model Evaluation

Chosen the parameter: (depth=9), a learning curve shows us the training and validation scores for different data sizes. This way, we are able to say that the score is bounded at about above 98.77%, and the model doesn't seem to continue learning after 2500-3000 training rows.

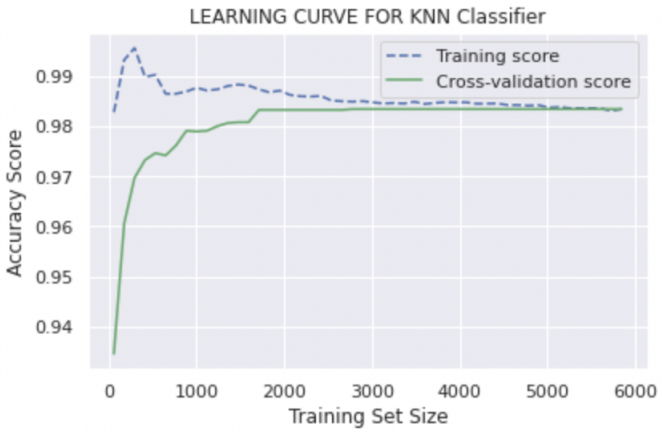


Figure 32: Random forest learning curve

4.8 Voting classifier

A voting classifier is a machine learning estimator that trains various base models or estimators and predicts on the basis of aggregating the findings of each base estimator. The aggregating criteria can be combined decision of voting for each estimator output.

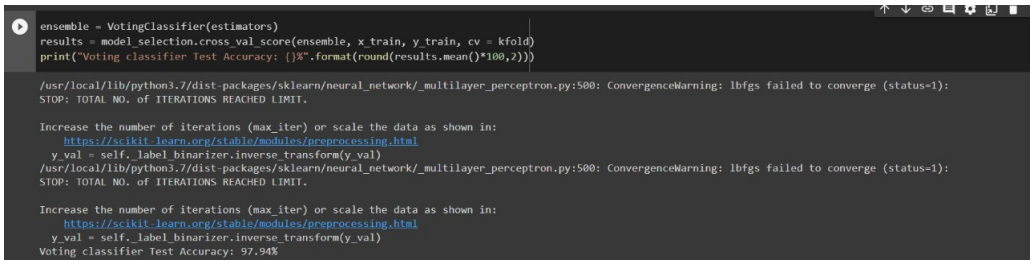


Figure 33: Voting classifier accuracy

4.9 AutoML

4.9.1 Presentation

AutoML provides a condensation of steps in the machine learning pipeline into a more concise, comprehensive step. The processes of acquiring, exploring, preparing, selecting models,

training, and tuning all become encompassed in the use of the AutoML framework. The use of this tool enables the analyst to step from data ingestion directly to prediction.

4.9.2 Experience

Simply fitting the training data to the AutoSklearnClassifier gives us an "optimal" model. The accuracy we had using autoML was: 98.65% We use the following code for obtaining the accuracy:

print("Accuracy score", sklearn.metrics.accuracy_score(y_test, predictions))

4.10 AUC curves

4.10.1 Presentation

It is a graph that evaluate a performance of a machine learning classification model at various thresholds settings. The higher the AUC, the better the model is at predicting the result of a classification.

4.10.2 Explanation

The graph below shows a comparation a all six classifiers and they are represented according to their assigned colors. The graph varies from 0 to 1 in both Y and X coordinates. The graph illustrates the true positive rate and false positive rate for the AUC Curve. From this illustration, we can observe that the KNN and Random Forest have almost the same trend and they overlap as they grow. It is evident from the plot that the AUC for these two classifiers are the highest. therefore we can say that KNN and Random did a better job of classifying the positive class in the dataset.

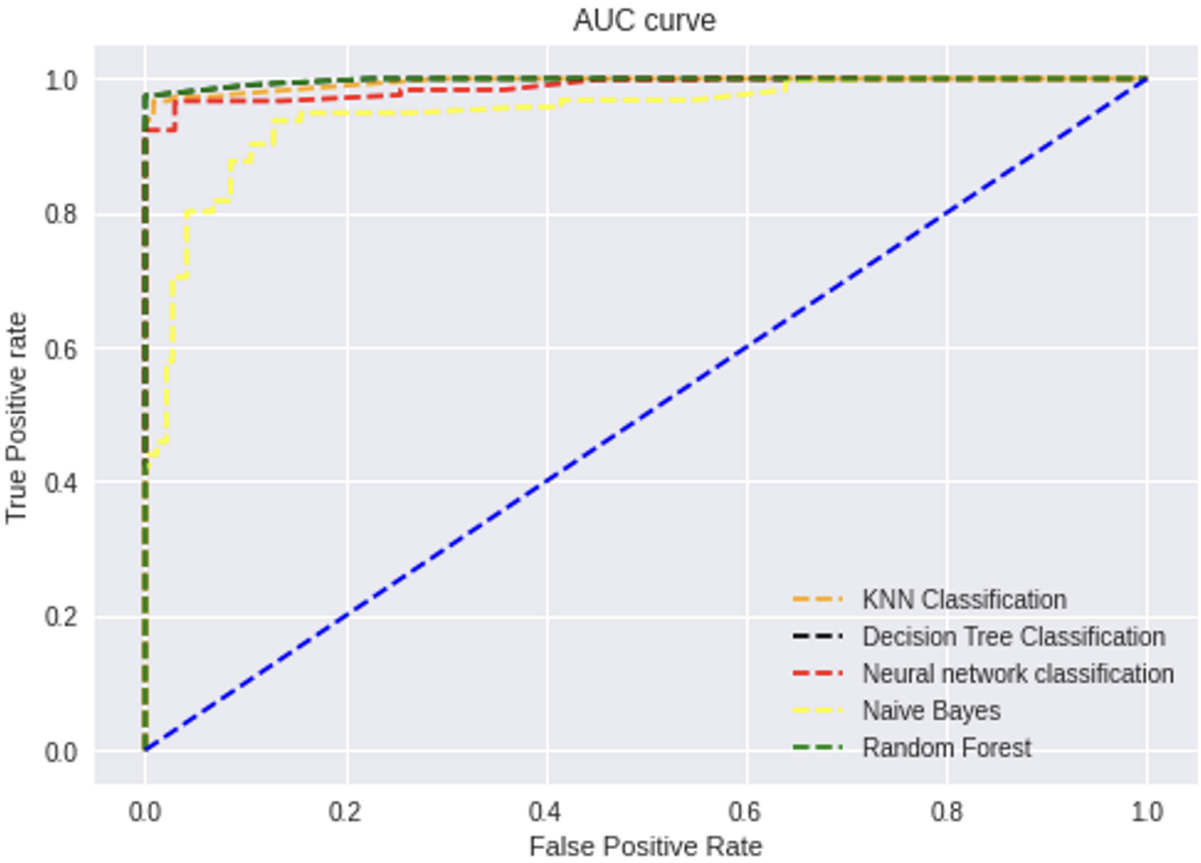


Figure 34: AUC curve

4.11 Best model for the study

The last part clearly states that KNN and Random Forest were the best because they did the best job.

5 Dimensionality reduction

5.1 Feature engineering/selection techniques

Feature engineering is the process of selecting and transforming knowledge from the raw data into features that can be used to improve the quality and the results of a machine learning model for this project we will use the Principal Component Analysis (PCA).

5.2 PCA

Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data for this project we will use it to generate new feature that have more importance for the learning process we reduce the dimensions from 23 to 6

```
x_pca.shape

(8124, 6)
```

Figure 35: pca shape

We use the new features to train our models:

- KNN After PCA:

```
Best KNN Value: 1
Test Accuracy: 100.0%
```

Figure 36: KNN

```
Test Accuracy: 99.82%
```

Figure 37: SVM

```
Test Accuracy: 92.43%
```

Figure 38: Naive Bayes

we can notice that the models have been improved after using PCA:
KNN from 98.34% to 100%.
SVM from 98.4% to 99.82%.
Naive Bayes from 88.92% to 92.43%.

5.3 Bonus - LIME

5.3.1 Presentation

LIME stands for Local Interpretable Model-Agnostic Explanations, and it enables us to test the causality of a feature, its reliability and can also help us debug the model appropriately. It tries to model the local neighborhood of any prediction.

5.3.2 Explanation

The graph below shows us the actual predicted value, a bar chart showing weights of how features contributed to this prediction, and a table showing actual feature values. In the graph below, it was used the method named show_in_nodetebook from the explainer, which will explain how we come to a particular prediction based on available features from the data.
It possible to see that for the current dataset, stalk-root and ring-type attribute contribute more for a random mushroom to be poisonous. However, stalk-surface-above-ring, gill-size, gill-spacing, gill-color and bruises contribute more in a positive way.

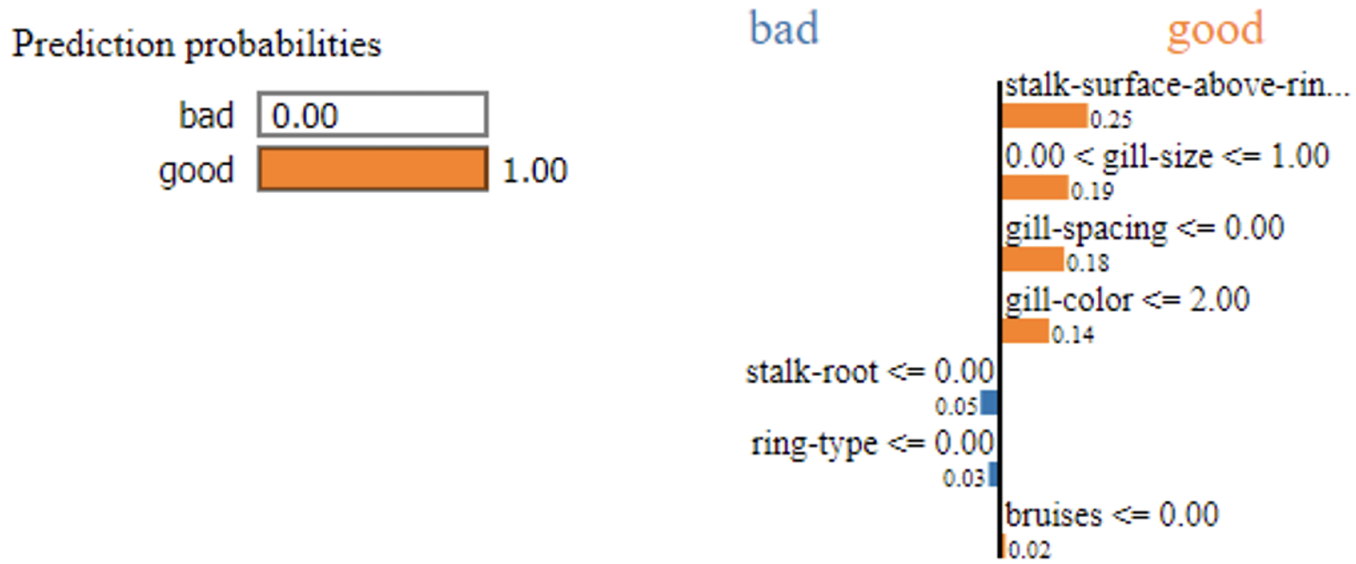


Figure 39: LIME output1

Feature	Value
stalk-surface-above-ring	1.00
gill-size	1.00
gill-spacing	0.00
gill-color	0.00
stalk-root	0.00
ring-type	0.00
bruises	0.00

Figure 40: LIME output2