A Study into Classification Models in R and Python

Sam Lorenz

August 2, 2022

Abstract

Classification models have become a strong tool in statistics and data science. There are many programs that use them, but two programs, R and Python, seem to have the most followers. The goal of this research was to run classification models in both programs to evaluate model performance as well as program performance. R performed better than Python in terms of classifying mushrooms as poisonous or edible; however, both programs are incredibly reliable, so choosing which to use should be up to the scientist and what parameters one wants available,

1 Introduction

Classification models are widely used among statisticians and data scientists. Classification models aim to read some input data and generate an output that will classify what was read into some category; a widely used example is predicting types of irises such as versicolor, setosa, or virginica. Some classification models can do this better (and more efficiently) than others, but the question worth studying is what makes each of these models unique, similar, and precise. The goal of this project was to discover the answers to these questions by classifying mushrooms as poisonous or non-poisonous using these classification models and tuning the parameters to make them as precise as possible. Another goal of this project is to use two programming languages (R and Python) and determine which is more efficient and clean (i.e, not confusing and jumbled) to interpret each of these models.

2 Methods

The data used was collected on Kaggle [Lea]. The set has 23 variables, including the target variable. Each variable has 8,125 entries. The target variable is whether the mushroom is edible (1) or poisonous (0), and the features are information about the mushroom (for example, the color, the shape of the stalk, the shape of the cap, etc).

Because some of the classification models used require encoded variables, one-hot encoded each variable. Additionally, after encoding the variables, I performed a correlation test to determine which variable combinations illustrated multicollinearity, and which variables make for good predictors to distinguish a poisonous mushroom from an edible mushroom.

Out of the 22 remaining variables, the missing being the target variable, the four that were not correlated amongst one another and the four best predictors were the odor of the mushroom, if it was bruised, the ring type, and the population size. This data frame was divided into training (about 80% of the data) and testing. Each model was done in R and in Python in order to compare the two. Below is a table of summary statistics of the variables kept, as well as another table of correlations.

Odor	Frequency
Almond	400
Anise	400
Creosote	192
Fishy	576
Foul	2160
Musty	36
None	3528
Pungent	256
Spicy	576

Bruises	Frequency
Bruised	3376
Not bruised	4748

Ring type	Frequency
Evanestent	2776
Flaring	48
Large	1296
None	36
Pendant	3968

Population	Frequency
Abundant	384
Clustered	340
Numerous	400
Scattered	1248
Several	4040
Solidarity	1712

Correlation	Odor	Bruises	Ring type	Population	Class
Odor	1.000	0.285	0.352	-0.276	0.786
Bruises	0.285	1.000	0.767	-0.179	0.502
Ring type	-0.276	0.767	1.000	-0.204	0.540
Population	-0.276	-0.179	-0.204	1.000	-0.444

3 Naïve Bayes

Naive Bayes is a classification model that is based on Bayes' Theorem. Recall that Bayes' Theorem is stated as:

$$P[A_i|B] = \frac{P[B|A_i]p[A_i]}{\sum_{j=1}^{n} P[B|A_j]P[A_j]}$$

For Bayes' Theorem, conditional independence is not assumed. Therefore, there is no assumed independence on predictor variables. For Naive Bayes (NB), independence is assumed, which is where 'naive' comes from. For example, a mushroom may be considered edible if it is red on the top, grows in a bunch, and has no odor. [Sun17] All of these features independently contribute to the notion that the mushroom is edible, which is where the Naive arises from. NB has many benefits due to its' simplicity; in fact, it even outperforms other complicated classification methods. There are many applications of Bayes, such as real-time predictions, multiclass predictions, spam classification (or other types of sentiment analysis), and a recommendation system (similar to the one you may notice on Netflix or HBO). [?] There are canned packages to perform NB in both Python and in R, plus ways to parameter tune as well. In R, there are many parameters that can be tuned for creating a naive Bayes model. Below is a table of parameters, and what they control [Camb].

laplace	Controls the Laplace smoothing constant; default is 0.
na.action	Specifies what to do in the case of NAs; the default is to not count them in probability factors. A specification of na.omit rejects cases of a variable with missing values.
type	If the type is "raw", conditional posterior probabilities are returned, as well as the class with maximal probability else.
threshold	Specified value that replaces cells with probabilities, specifically in the 'eps' range.
eps	Can double for specifying an epsilon-range to apply laplace smoothing or to replace zero and close-zero probabilities by the threshold command

Before tuning the parameters, I created a model with all the defaults to see how well it performed. The false negative rate was 8/1119, or 0.7%. The false positive rate was 0%. The overall accuracy of the model was 99%; the accuracy as well as the false positive and negative rates were great, but the model predicted 8 mushrooms as edible when they are really poisonous, which is something notable. The parameters I tuned were the Laplace smoothing constant, and the threshold since they made the most sense for the model. This did not affect the model at all; the only case in which the results of the model changed was when I added variables that were correlated with the original four, or when the odor variable was not present. The odor variable really impacted the classification of a mushroom, and without it, our model performed much worse.

In Python, the parameters are fewer but have the same impact. There is a parameter for the laplace smoothing constant, which has a default of 1 instead of 0. There is also an option to binarize (map to the booleans), and there are options to learn the prior probabilities and distribution. When the model was done in Python, the false negative rate was much higher, at 19%. The false positive rate was still 0%. [sldb] This is quite a change from R, but the reason could be the difference in how the programs are written or the different parameters from R to Python. Trying to make the models the exact same is very hard, especially because the lack of certain parameters in either program will make the model fail to compile.

3.1 Results and Discussion

In terms of overall model performance, R took the lead. This could be due to the structure of the algorithm itself; there were a lot fewer parameter options to tune and the parameters that were available did not contain a lot of mathematical contributions. In Python, there was a wider range of parameters to play with and they were on the more mathematical side. Furthermore, in Python, the naive Bayes was more of a Bernoulli-NB approach, meaning it treated the data like multivariate Bernoulli models. In R, it computes the conditional posterior probabilities of the class variable given the predictor variables using the Bayes rule described above. Both programs utilize the NB, but with a different twist. Depending on what the scientist or statistician wants, one program could be more beneficial than the other. As far as the speed goes, R was a few lines less than Python, but both were about equal as far as making the model goes. The results were read in R better, due to the package used to make the confusion matrix and other tables; but, something similar in Python can be done in just a few lines.

4 Logistic Regression

Logistic Regression is used when the dependent variable is categorical. Common examples are the iris example, predicting emails as spam or not spam, and other variables like so. For the mushroom classification, we want to determine whether a mushroom is edible or not. If we used linear regression, there would be an issue based upon which the classification can be done. [Swa18] If the actual classification is poisonous, predicted as 0.0 with a threshold value of 0.5, then the point will not be classified as a poisonous mushroom. [Swa18] This can lead to serious issues, therefore, linear regression is not suitable. So, logistic regression throws a bound on the data and ranges values from 0 to 1. [Swa18] This also helps fit the Sigmoid function, which is the basis of Logistic Regression. When using logistic regression, there are many great parameters to be tuned that go beyond regular linear regression. The threshold probability is just one of the many parameters to tune to make a great Logistic model. Below is a table of tune-able parameters for logistic regression in R [Cama].

Family	A parameter to specify details of the error distribution and link function used in the model.
Weights	Optional prior weights; either needs to be a numerical vector or NA.
na.action	Specifies what to do in the case of NAs; the default is to not count them in probability factors. A specification of na.omit rejects cases of a variable with missing values.
Start	Starting values for the parameters in the linear predictor.
Estart	Starting values for the linear predictor.
Mustart	Starting values for the vector of means.
Offset	During fitting, this can be sued to specify a priori known component. This should be null, or a numeric vector equal to the number of cases.
Control	A list of parameters to control the fitting process.
Intercept	Indicates whether an intercept should be included in the model.

Logistic regression had a bit of a limit on tune-able parameters; for example, playing with the intercept information would be useless since we do not have that information. For the family component, I kept it as binomial since that is the distribution of the data. This parameter should be left alone unless a data distribution can be considered as two different families; then, playing with both may be a good idea to see which results seem more realistic. The logistic regression model also had a false positive rate of 0.7% and a false negative rate of 0.0%. What is interesting about the logistic regression model is, much like naive Bayes, playing with the parameters did not change the results of the model. What did change the results was adding a variable correlated with another, or removing the odor variable.

In Python, there are far more parameters to be tuned for logistic regression. One can tune the penalty, the fit intercept, a solver for optimization, and how any iterations the model should perform [slda]. Other parameters specify multinomial models. Because of how the parameters in R differ from Python, it was very hard to get the models similar. In Python, the model had a false negative rate of 10% and a false positive rate of 2%. It is possible this model performed worse due to the vast difference in parameters; however, it still ended up being a fairly solid model. Depending on the preference of the statistician, if one valued certain parameters more than others, then there could be a more clear decision on what program to use.

4.1 Results and Discussion

Logistic regression in R and Python were, once again, different in terms of the mathematical parameters. Python had a lot of options for penalty tuning and creating a model based on an intercept, where R was more focused on the overall fitting process. Python had options for making a model based on error preferences, and R was more descriptive of errors. This is an interesting approach because one could use both models to make an overall conclusion about the data and modeling. In R, I felt I had a baseline idea of how my model was, and when I went to Python, I found myself playing with the different types of penalties and regularizations. The R model still performed better than the Python model, but perhaps with a more in-depth study of Python, one could create a better model. As far as the speeds of the performance, it was still much more simple to make the model in R than in Python; but, the speed at which results were given was the same.

5 Support Vector Machines

If you are aware of linear and logistic regression, you may also be familiar with Support Vector Machines (SVMs). A SVM can be used for both regression (SVR, or Support Vector Regression), and classification tasks. [Gan18] Typically, it is used in classification. SVMs are a bit math-y, in the sense that it attempts to find a hyperplane in an N-dimensional space that classifies the data points. [Gan18] Although many hyperplanes could be chosen, the goal is to find the maximum distance between the data points so that when new data is read in, it is classified with more confidence. [Gan18] There is also a multitude of parameters to tune with SVMs for better and sometimes, for worse. Using confusion matrices can help us determine what tune is best. Below is a table of tune-able parameters for SVM in R[Camd].

Scale	Indicates the variables needing to be scaled; if it is of length 1, then the value is reused as many times as needed. The scale is set to zero as a default.
Type	This parameter controls whether SVM or SVR is occurring. The two types of classification for SVM are C-classification and nu-classification.
Kernel	This is sued in training and predicting; depending on the form of the data, the kernel can be linear, polynomial, radial basis, or a sigmoid.
Degree	Needed for the kernel polynomial.
Gamma	Needed for all kernels but linear.
Coef0	Needed for kernels polynomial and sigmoid. Default is zero.
Cost	Defines cost constraints. Default is 1.
Nu	Parameter needed for nu-classification.
na.action	Specifies what to do in the case of NAs; the default is to not count them in probability factors. A specification of na.omit rejects cases of a variable with missing values.

There are many other parameters that can be toyed with in the SVM model, but the ones above seem to be the most crucial for the models. Furthermore, a lot of the extra parameters are for the different types of classifications and regressions, as well as the kernels. For the SVM model, I used C-classification with the linear kernel. If other kernels were used, the model was terrible (if it ran at all). Otherwise, tuning the parameters did not change the result much and the false negative rate was still 0.7%, with a false positive rate of 0.0%. It is interesting that among the different classification models, all had the same result; but, it is important to note the importance of the odor variable in our model.

The parameters for the SVM model in Python and R are very similar. There is one extra kernel in Python, called precomputed, that is not available in R. [sldd] Although the parameters are similar, the way in which the model performs is not. This is noted when comparing both the false positive and false negative rates; in Python, the false negative rate is 10% and the false positive rate is 2%. It is interesting that even with similar parameters, there is a difference between the false positive and negative rates. The canned programs compared to the made programs have the flaw of being written differently, so it is hard to produce exact results; but, due to the different parameters and extra options between both R and Python, it is an available option to use one over the other.

5.1 Results and Discussion

Both models had the option to do a support vector machine or support vector regression, which is convenient instead of having to install an entirely new package. Python had a few more kernels than R, so depending on how the data is, one may need to use Python over R. Furthermore, most of the parameters were the same across R and Python. So why did R perform better than Python? One obvious reason is the creators of the models wrote the two versions differ. Another reason could be that R is not as hands-on as Python; meaning, getting the exact SVM model you want in Python takes a lot more work and parameter tuning than it does in R. Really, you do not need to specify any parameters in R for the model to run well; but in Python, this could risk a lot of error. As far as the speed, both models were around the same in terms of coding it to work, getting the results, and interpreting the results. Perhaps because the idea of SVM is so new, and the packages in both programs are so new, speed and tune-able parameters turn out to be pretty much the same.

6 Decision Trees

A decision tree is a type of model used to categorize (or, make predictions) based on how a previous set of questions was answered. Usually, it is trained on a set of data that contains the categorization and tested on another set. [Inc] Sometimes, a decision tree doesn't have a clear answer or decision; but, it presents options so the data scientist can make an informative decision themselves. [Inc] Because decision trees imitate human thinking, the results are easily interpreted by the scientist. A decision tree can be categorical or continuous, or it can contain both types of variables, making it incredibly versatile. [Inc] A well-designed tree will have few nodes and few branches to avoid overfitting. A tree that has too many questions and paths to go based on the data will perform poorly on the testing and new data. [Inc] Decision trees have plenty of parameters to tune, such as the depth of the tree, minimum amount of splits, and many more to make the model the best it can be. A table of parameters is provided below [Camc].

Weights	A positive vector of observational weights.
Na.action	A function that filters missing data. Na.pass does nothing, and na.tree handles missing values by dropping them down the tree.
Split	Set a splitting criterion
Method	Practically the purpose of the tree. Class, Poisson, ANOVA, or exp are acceptable methods. If y is alone, then exp is assumed. If y has two columns, then Poisson is assumed. If y is a factor, class is assumed; otherwise, ANOVA is assumed.
Cost	Defines cost constraints when the tree splits. Default is 1.

For the tree model in R, the method wanted was class since that was the purpose of the tree. The tree model was unaffected by the other parameters due to the importance of the odor variable; when removed, the tree had a larger depth and more splits. The false negative rate was still 0.7%, and the false positive rate was 0.0%. Interestingly enough, when the model contained correlated variables, the tree performed the best due to the splitting as well as the cost. The odor variable was still a prominent predictor, even in the tree model. In the future, it would be interesting to see the tree model with other methods and evaluate the performance once more.

The parameters in a Python tree are a bit different than those in an R tree. For example, the max-depth can be fixed, as well as the minimum samples in a leaf and a split, the criterion for error, the strategy for a split, and weights for classifying and splitting. [sldc] Once again, it is challenging to get both models to be exact; the false negative rate was 10% and the false positive rate was 2%. It is interesting that all the R models performed the same and most of the Python models performed the same; this is an interesting conclusion that although the results were different between programs, they did not stray too far from each other. The only model that was different in Python was naive Bayes, and this could be because it takes a more Bernoulli approach.

6.1 Results and Discussion

In R, there are many packages and options to make a decision tree. I chose to stick with the `rpart` package because it seemed newer than the others and the most similar to the Python installation; but, there were still a lot more parameters available in Python than in R. The tree in R was completely dependent on whether or not the mushroom had an odor or not, but in Python, the tree happened to be much more in-depth. No matter what the parameters were tuned to in R, it did not change the model. The only time the model changed was with the absence of the odor variable. This could be a fault on my part; however, I believe the tree model in R was much more tuned to the variables. In Python, the parameters changed how the tree performed and split more so than in R. The speeds were overall the same, as far as the computation and evaluation area was concerned.

7 Overall Results

As far as model performance is concerned, R had better false positive and false negative scores in each classification model. The models ran much differently in Python and in R, and the parameters to tune tended to be much more mathematical in Python; meaning, changing the parameters in Python could really tamper with how the model is run. In R, the parameters were more or less polishing the model and did not tend to change much, unless doing an SVM model; depending on how the data is, if the wrong kernel is selected, the model will perform terribly across both programs. A common theme across R and Python was the importance of the odor variable. It seems this really contributed to the mushroom being classified as edible or poisonous; it was not perfect, but without it, the model did not do so well. So although the parameters can make or break the model (more so in Python), it is really up to how the variables represent the target feature. In Python, I liked how much I could change the canned packages to make the model run how I want it to run. In R, I liked the simplicity of relying on a solid model without really doing much to the parameters. At the end of the day, it depends on what the scientist wants, so choosing which program to use should be left up to the purpose of the research.

8 Conclusion

The overall performance of the models in both R and Python was good. It was interesting to evaluate why the results were different among the models in both programs and how the parameters can have something to do with that. In R, the models are more tuned toward the variables. In Python, it seems the models were more parameter-tuned. It was difficult to get the models similar in R and Python just because of how different some of the parameters and defaults were. In R, it is easier to control how a model operates, whereas in Python, it seemed more challenging to get things right without error. A common argument is whether R or Python is better; after this project, I say it depends. If a statistician wants certain parameters over others and values speed, how the overall model works in the program, and the cleanliness of the results, then one would have an easier time choosing R or Python. In the future, it would be useful to dive deeper into how the models are coded in the programs to see where some of these differences arise from.

References

[Cama] Data Camp. glm: Fitting Generalized Linear Models.

[Camb] Data Camp. naiveBayes: Naive Bayes Classifier.

[Camc] Data Camp. rpart: Recursive Partitioning and Regression Trees.

[Camd] Data Camp. svm: Support Vector Machines.

[Gan18] Rohith Gandhi. Support vector machine introduction to machine learning algorithms. 2018.

[Inc] 2U Inc. What is a decision tree?

[Lea] UCI Machine Learning. Mushroom classification.

[slda] scikit-learn developers. sklearn.linearmodel.LogisticRegression.

[sldb] scikit-learn developers. sklearn.naivebayes.BernoulliNB.

[sldc] scikit-learn developers. sklearn.tree.DecisionTreeClassifier.

[sldd] scikit-learn developers. Support Vector Machines.

[Sun17] Sunil. 6 easy steps to learn naive bayes algorithm with codes in python and r. 2017.

[Swa18] Saishruthi Swaminathan. Logistic regression - detailed overview. *Towards Data Science*, 2018.

9 Code

9.1 R Code

```r
##usual preamble and loading data
source("https://rfs.kvasaheim.com/stat200.R") ##just using 200.. seems to have a
library(readr)
mdt <- read_csv("C:/Users/18153/Downloads/mushroom.csv")
attach(mdt)
View(mdt)

##making sure I translated right
table(class, ringType)
table(capShape)
table(capSurface)
table(capColor)
table(bruises)
table(odor)
table(gillAttachment)
table(gillSpacing)
table(gillSize)
table(gillColor)
table(stalkShape)
table(stalkRoot)
table(stalkSurfaceAR)
table(stalkSurfaceBR)
table(stalkColorAR)
table(stalkColorBR)
table(veilType) #all partial, so this may not be helpful
table(veilColor)
table(ringNumber)
table(ringType)
table(sporePrintColor)
table(population)
table(habitat)

##correlation tests?

newClass <- ifelse(class=="edible", 1, 0)
newcapShape <- ifelse(capShape=="convex", 1, 0)
newcapSurface <- ifelse(capSurface=="scaly", 1, 0)
newcapColor <- ifelse(capColor=="brown", 1, 0)
newBruises <- ifelse(bruises=="TRUE", 1, 0)
newOdor <- ifelse(odor=="none", 1, 0)
newgillAttachment <- ifelse(gillAttachment=="free", 1, 0)
newgillSpacing <- ifelse(gillSpacing=="close", 1, 0)
newgillSize <- ifelse(gillSize=="broad", 1, 0)
newgillColor <- ifelse(gillColor=="buff", 1, 0)
newstalkShape <- ifelse(stalkShape=="tapering", 1, 0)
```

```
newstalkRoot <- ifelse(stalkRoot=="bulbous", 1, 0)
newstalksurfaceAR <- ifelse(stalkSurfaceAR=="smooth", 1, 0)
newstalksurfaceBR <- ifelse(stalkSurfaceBR=="smooth", 1, 0)
newstalkcolorAR <- ifelse(stalkColorAR=="white", 1, 0)
newstalkcolorBR <- ifelse(stalkColorBR=="white", 1, 0)
newveilColor <- ifelse(veilColor=="white", 1, 0)
newringNumber <- ifelse(ringNumber=="one", 1, 0)
newringType <- ifelse(ringType=="pendant",  1, 0)
newsporeprintColor <- ifelse(sporePrintColor=="white", 1, 0)
newPopulation <- ifelse(population=="several", 1, 0)
newHabitat <- ifelse(habitat=="grasses", 1, 0)

df <- data.frame(newClass, newBruises, newOdor,newringType, newPopulation)
cor(df)


######## Naive Bayes########

library(e1071)
mush_train <- df[1:6499,c("newBruises", "newOdor", "newringType", "newPopulation
train_labels <- df[1:6499,c("newClass")]
mush_test <- df[6499:8124,c("newBruises", "newOdor", "newringType", "newPopulati
test_labels <- df[6499:8124,c("newClass")]
mushMod1 <- naiveBayes(mush_train, train_labels, laplace = 0)
mushpreds <- predict(mushMod1,mush_test,type="raw")
predclass <- ifelse(mushpreds[,2]>=.6,1,0)

library(gmodels)
CrossTable(test_labels, predclass)

##  SVM   ###########

mushMod2 = svm(formula = train_labels ~ .,
                data = mush_train,
                type = 'C-classification',
                kernel = 'linear')

mushpreds2 <- predict(mushMod2,mush_test,type="raw")
CrossTable(test_labels, mushpreds2)

#### Logistic Regression #########

x <- 1:6499
new_samp <- sample(x)
mush_train_2 = mdt[new_samp, c("bruises", "odor", "ringType", "population")]
mush_test_2 = mdt[-new_samp, c("bruises", "odor", "ringType", "population")]
newClass[-new_samp]

mushMod4 <- glm(newClass ~ mdt$bruises + mdt$odor + mdt$ringType + mdt$populatio
family=binomial, data=mush_train_2)
summary(mushMod4)
```

```
mushpreds4 <- predict(mushMod4, newdata = mush_test_2, type="response")
predClass4 <- ifelse(mushpreds4 >= 0.8, 1, 0)
CrossTable(newClass[-new_samp],predClass4[-new_samp])

#### Trees ########################
library(rpart)
library(rpart.plot)
mushMod5 <- rpart(newClass ~  mdt$bruises + mdt$odor + mdt$ringType + mdt$popula
mushpreds5 <- predict(mushMod5, newdata = mush_test_2, type="class")
rpart.plot(mushMod5)
CrossTable(newClass[-new_samp], mushpreds5[-new_samp])
```

9.2 Python Code

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 14 14:28:57 2022

@author: Samantha Lorenz
"""

import pandas as pd
import sklearn as sk
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

mdt = pd.read_csv('C:/Users/18153/Downloads/mushroom.csv')

##encoding vars as 1 or 0 (same way as R)
#print( mdt["class"].value_counts() )
newClass = {"class": {"edible":1, "poisonous": 0} }
mdt = mdt.replace(newClass)

#print( mdt["bruises"].value_counts() )
mdt["bruises"] = mdt["bruises"]*1

#print( mdt["odor"].value_counts() )
newOdor = {"odor": {"none":1, "foul": 0, "spicy": 0, "fishy": 0, "anise": 0, "al
mdt = mdt.replace(newOdor)

#print( mdt["ringType"].value_counts() )
newringType = {"ringType": {"pendant":1, "evanescent": 0, "large": 0, "flaring":
mdt = mdt.replace(newringType)

#print( mdt["population"].value_counts() )
newPopulation = {"population": {"several":1, "solitary": 0, "scattered": 0, "num
mdt = mdt.replace(newPopulation)
```

```
##training and testing

d  = {'class': mdt["class"], 'bruises': mdt["bruises"], 'odor': mdt["odor"], 'ri
df = pd.DataFrame(data = d)
x = df[['bruises', 'odor', 'ringType', 'population']].copy()
x1 = np.array(x)
y = df[['class']]
y1 = np.array(y)

x_train, x_test, y_train, y_test = train_test_split(x1, y1, test_size=0.2)

##naive bayes

from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

modBayes = BernoulliNB(alpha=0, fit_prior=True)
modBayes.fit(x_train, y_train )
prediction_prob = modBayes.predict_proba(x_test)
prediction = modBayes.predict(x_test)

print(confusion_matrix(y_test, prediction, labels=[0, 1]))

report = classification_report(y_test, prediction)
print(report)
accuracy = modBayes.score(x_test, y_test)
print(f'The accuracy is: {accuracy*100:.1f}%')

## SVM

from sklearn import svm
modSVM = svm.SVC(kernel='linear')
modSVM.fit(x_train, y_train)
svmPred = modSVM.predict(x_test)
print(confusion_matrix(y_test, svmPred, labels=[0, 1]))
svmReport = classification_report(y_test, svmPred)
print(svmReport)
accuracySVM = modSVM.score(x_test, y_test)
print(accuracySVM)

## logistic

from sklearn.linear_model import LogisticRegression
modLog = LogisticRegression(penalty='l2')
modLog.fit(x_train, y_train)
logPred = modLog.predict(x_test)
print(confusion_matrix(y_test, logPred, labels=[0, 1]))
logReport = classification_report(y_test, logPred)
```

```
print(logReport)
accuracyLog = modLog.score(x_test, y_test)
print(accuracyLog)

## trees

from sklearn import tree
treeMod = tree.DecisionTreeClassifier()
treeMod.fit(x_train, y_train)
treePred = treeMod.predict(x_test)
print(confusion_matrix(y_test, treePred, labels=[0, 1]))
treeReport = classification_report(y_test, treePred)
print(treeReport)
accuracyTree = treeMod.score(x_test, y_test)
print(accuracyTree)
```