EMORY UNIVERSITY

Introduction to Business Analytics [MSBA]

Homework #2

Carl Xi

(put your full name above (incl. any nicknames))

Note: This is an individual homework. Discussing this homework with your classmates is a violation of the Honor Code. If you borrow code from somewhere else, please add a comment in your code to make it clear what the source of the code is (e.g., a URL would sufficient). If you borrow code and you don't provide the source, it is a violation of the Honor Code.

Total grade: ________ out of ___150___ points

1) (15 points) Would you frame the problem of e-mail spam detection as a supervised learning problem or an unsupervised learning problem? Please justify your answer.

I would frame the problem of e-mail spam detection as a supervised learning problem. We have a specific target purpose in mind (identifying whether an email is spam or not, and it is easy to classify past emails as spam or not to create training/test datasets for our target variable for our model. This is a classic classification problem with a binary categorial target.

2) (15 points) What is a test set and why would you want to use it?

A test set is used in holdout validation, where given only one data set, we hold out some data (the test set) for the evaluation of our mode to see its generalized performance (performance on data not used for the training of the model). The test set helps us find "out-of-sample" accuracy, which is a much better predictor of how the model actually performs compared to "in-sample" accuracy which is accuracy on the training data. It is important to note that the test set is different from the validation set, where the former is left in isolation during model training while the latter is often used to help refine the model (e.g. selecting the best *'k'* for k-nn). By seeing the difference in performance between the test and training set, we will also be able to spot overfitting and make adjustments to our model.

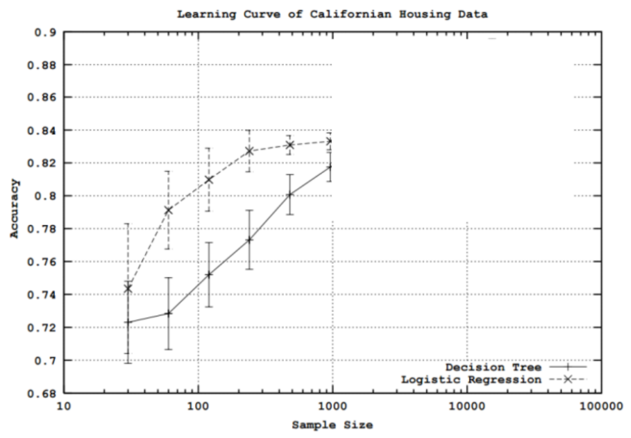
3) (20 points) What are the similarities and differences of decision trees and logistic regression? When might you prefer to use one over another?

Both Decision Trees and Logistic Regression are popular data mining tools because they are easy to understand, implement, use and are computationally cheap. Both are used for predicting target variables that are categorical. While logistic regression has regression in its name and produces a numeric result, it is still used for categorical estimation and the number is simply the probably of a class within the target variable. Their advantages in model comprehensibility are also important for model evaluation and when communicating with non-datamining-savvy stakeholders.

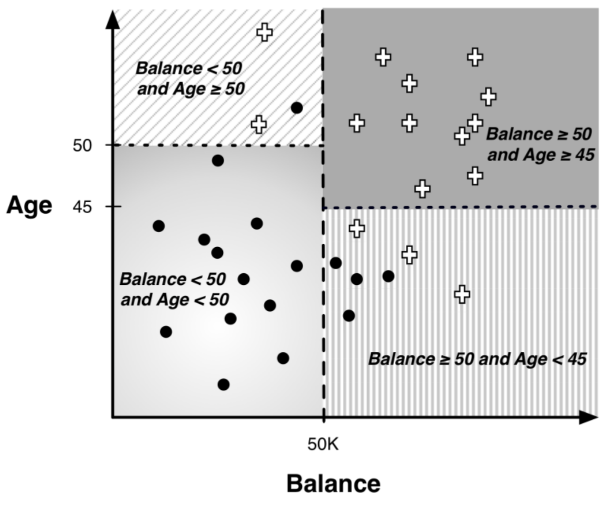
However, the two have some inherent differences that make them better than each other at certain tasks. Classification trees are often thought of as a set of rules, while logistic regression is often thought of as a mathematical function. Decision trees use decision boundaries that are perpendicular to the instance-space axes, and only select a single attribute at a time. On the other hand, linear classifiers like logistic regression can use decision boundaries of any direction or orientation, as they utilize a weighted combination of all the attributes. While logistic regression places a **single** decision surface through the entire space, classification trees cut up the instance space **arbitrarily** finely into **very small regions**.

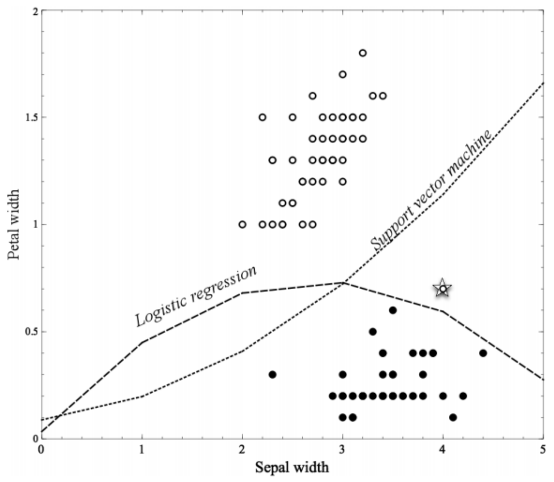
Lastly, logistic regression and decision trees are better for smaller and larger training-datasets respectively, as logistic regression typically yields better generalization accuracy for smaller datasets whereas decision trees tend to over-fit more. But classification trees are also a more flexible model representation than logistic regression, and this flexibility is especially helpful in large training sets, where the decision trees can represent substantially nonlinear relationships between the features and the target.

At the end of the day, picking whether to use decision tree or logistic regression is very case-by case. Would a set of rules or a numeric function make more sense to stakeholders? What is the smoothness of the underlying phenomenon being modeled? (Trees need large datasets to approximate curved boundaries). How "non-linear" is the underlying phenonium being modeled? (A lot of data engineering will be needed to utilize logistic regression if the data is very non-linear). How much data do we have? How does the data look like, what are its characteristics? Are there irregularities like missing values? Are some of the variables irrelevant? Is there multicollinearity? Are the variables numeric or something else? (Trees are typically more robust in dealing with more complications). Theses are the questions one has to ask themselves when deciding between decision trees and logistic regression. With all this said, a simple look at the Learning curve will help to visualize which model is more effective for your specific data mining problem.



Example learning curve that shows accuracy of logistic regression vs decision tree at different sample sizes


Balance < 50 and Age ≥ 50
Balance ≥ 50 and Age ≥ 45
Balance < 50 and Age < 50
Balance ≥ 50 and Age < 45
Age
50
45
50K
Balance
Example of decision tree rules



Example of a logistic regression boundary

4) (30 points) You have a fraud detection task (predicting whether a given credit card transaction is "fraud" vs. "non-fraud") and you built a classification model for this purpose. For any credit card transaction, your model estimates the probability that this transaction is "fraud". The following table represents the probabilities that your model estimated for the validation dataset containing 10 records.

Actual Class (from validation data)	Estimated Probability of Record Belonging to Class "fraud"	0.3 Cutoff	0.8 Cutoff	FP=0	FN=0
fraud	0.95	TP	TP	TP	TP
fraud	0.91	TP	TP	TP	TP
fraud	0.75	TP	FN	TP	TP

non-fraud	0.67	FP	TN	TN	FP
fraud	0.61	TP	FN	FN	TP
non-fraud	0.46	FP	TN	TN	FP
fraud	0.42	TP	FN	FN	TP
non-fraud	0.25	TN	TN	TN	TN
non-fraud	0.09	TN	TN	TN	TN
non-fraud	0.04	TN	TN	TN	TN

Based on the above information, answer the following questions:

a) **What is the overall accuracy of your model, if the chosen probability cutoff value is 0.3? What is the overall accuracy of your model, if the chosen probability cutoff value is 0.8?**

I basically set the cutoff in the chart above and labeled every datapoint as TP/TN/FP/FN. Using the formula for accuracy below, I have calculated the accuracy for cutoff values of 0.3 and 0.8 respectively.

$$Accuracy(0.3) = \frac{TP+TN}{Total} = \frac{5+3}{10} = 0.8 \qquad Accuracy(0.8) = \frac{TP+TN}{Total} = \frac{2+5}{10} = 0.7$$

b) **What probability cutoff value should you choose, in order to have Precision fraud = 100% for your model? (Explain.) What is the overall accuracy of your model in this case?**

We know that precision is calculated by the formula $Precision = \frac{TP}{TP+FP}$. For precision fraud to be 100%, we must have 0 false positives at our cutoff. This means our cutoff must be **greater than 0.67**, as any lower and our model will flag the [non-fraud] datapoint at 0.67 as fraud, making it a False Positive prediction.
Assuming our cutoff is right above 0.67, our accuracy would be $\frac{TP+TN}{Total} = \frac{3+5}{10} = 0.8$

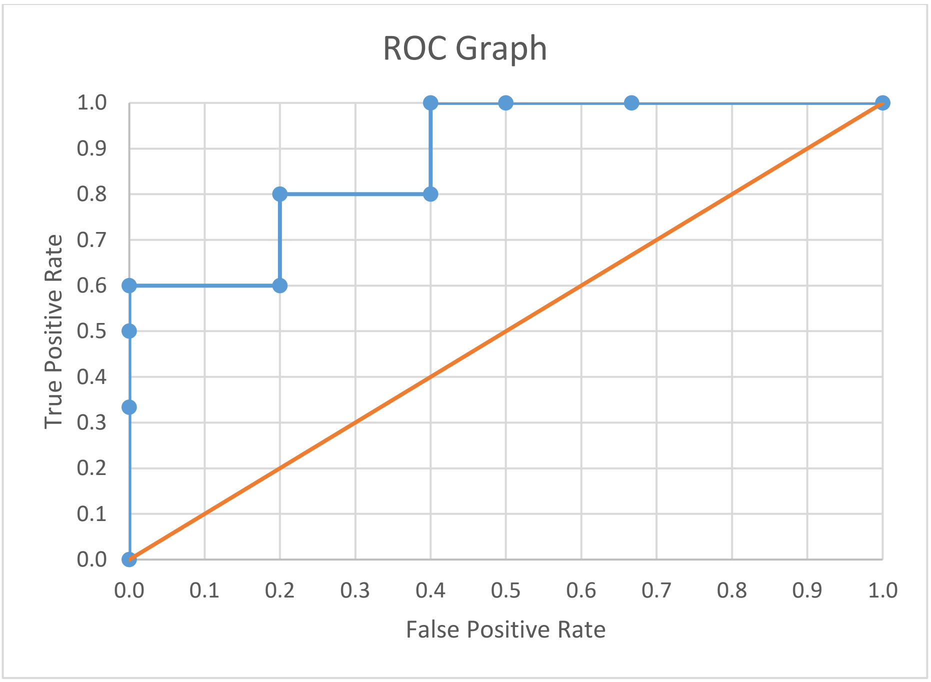
c) **What probability cutoff value should you choose, in order to have Recall fraud = 100% for your model? (Explain.) What is the overall accuracy of your model in this case?**

We know that recall is calculated by the formula $Recall = \frac{TP}{TP+FN}$. For recall fraud to be 100%, we must have 0 false negatives at our cutoff. This means our cutoff must be **less than 0.42**, as any higher and our model will flag the [fraud] datapoint at 0.42 as non-fraud, making it a False Negative prediction.
Assuming our cutoff is right above 0.67, our accuracy would be $\frac{TP+TN}{Total} = \frac{5+3}{10} = 0.8$

d) **Draw a ROC curve for your model. (Not graded, but looks cool and was a good learning experience)**

Actual Class	**Estimated Probability of "Fraud"**	**Cutoff = 0.99**	**Cutoff = 0.95**	**Cutoff = 0.91**	**Cutoff = 0.75**	**Cutoff = 0.67**	**Cutoff = 0.61**	**Cutoff = 0.46**	**Cutoff = 0.42**	**Cutoff = 0.25**	**Cutoff = 0.09**	**Cutoff = 0.04**
fraud	0.95	FN	TP	TP	TP	TP	TP	TP	TP	TP	TP	TP
fraud	0.91	FN	FN	TP	TP	TP	TP	TP	TP	TP	TP	TP
fraud	0.75	FN	FN	FN	TP	TP	TP	TP	TP	TP	TP	TP
non-fraud	0.67	TN	TN	TN	TN	FP	FP	FP	FP	FP	FP	FP
fraud	0.61	FN	FN	FN	FN	FN	TP	TP	TP	TP	TP	TP
non-fraud	0.46	TN	TN	TN	TN	TN	TN	FP	FP	FP	FP	FP
fraud	0.42	FN	FN	FN	FN	FN	FN	FN	TP	TP	TP	TP
non-fraud	0.25	TN	TN	TN	TN	TN	TN	TN	TN	FP	FP	FP
non-fraud	0.09	TN	TN	TN	TN	TN	TN	TN	TN	TN	FP	FP
non-fraud	0.04	TN	TN	TN	TN	TN	TN	TN	TN	TN	TN	FP

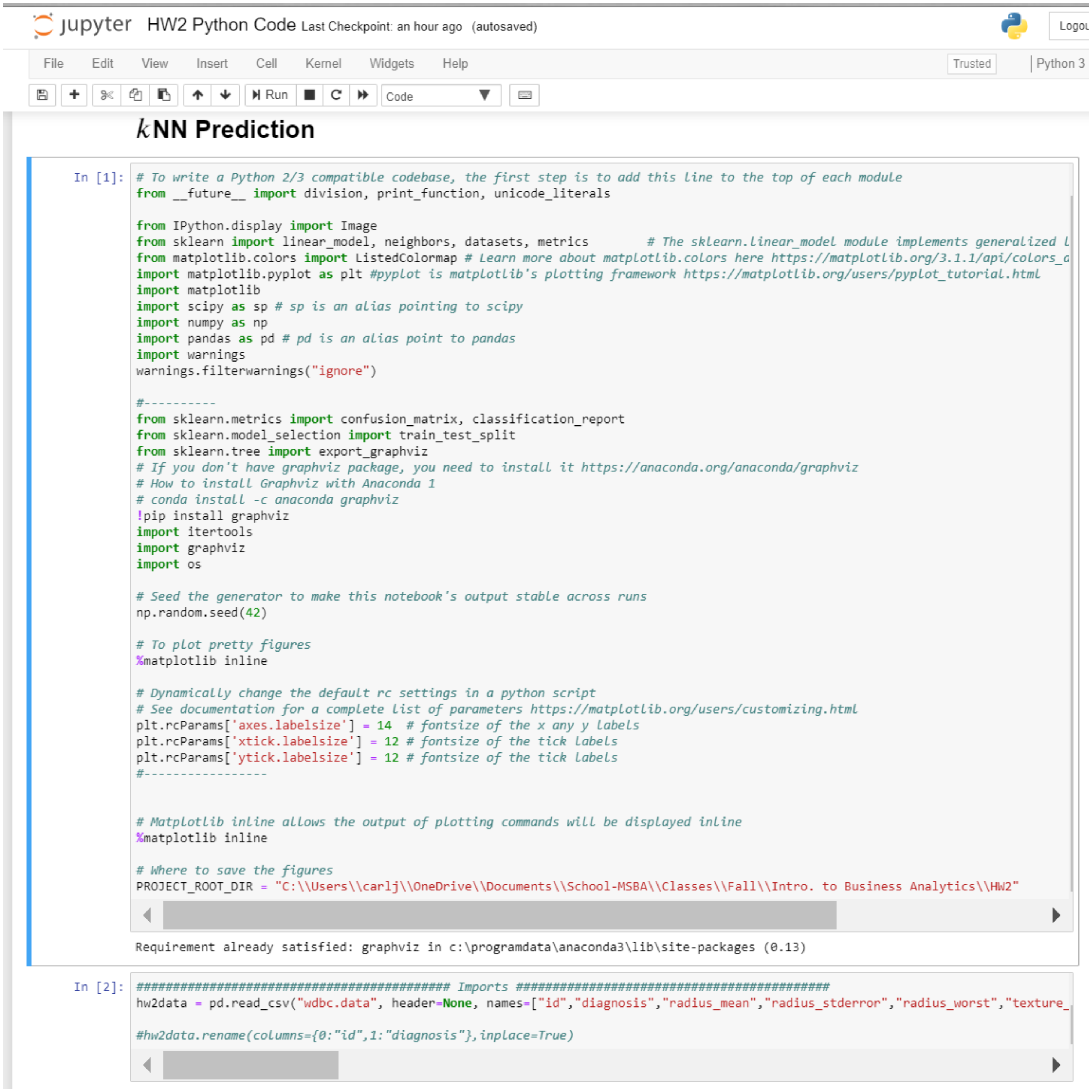
Sum (TP)	Sum (TN)	Sum (FP)	Sum (FN)	Sum (FN)	FP Rate	TP Rate
0	5	0	2	0	0.0	0.0
1	5	0	2	0.1	0.0	0.3
2	5	0	2	0.2	0.0	0.5
3	5	0	2	0.3	0.0	0.6
3	4	1	2	0.4	0.2	0.6
4	4	1	1	0.5	0.2	0.8
4	3	2	1	0.6	0.4	0.8
5	3	2	0	0.7	0.4	1.0
5	2	2	0	0.8	0.5	1.0
5	1	2	0	0.9	0.7	1.0
5	0	2	0	1	1.0	1.0



5) (70 points) [Mining publicly available data. Please implement the following models both with Rapidminer and Python but explore the data (e.g., descriptive statistics etc.) just with Python]
Please use the dataset on breast cancer research from this link: http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data [Note: RapidMiner can import .data files in the same way it can import .csv files. For Python please read the data directly from the URL without downloading the file on your local disk.**] The description of the data and attributes can be found at this link: http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.names. Each record of the data set represents a different case of breast cancer. Each case is described with 30 real-valued attributes: attribute 1 represents case id, attributes 3-32 represent various physiological characteristics, and attribute 2 represents the type (benign or malignant). If the dataset has records with missing values, you can filter out these records using Python and/or Rapidminer before proceeding with the models. Alternatively, if the data set has missing values, you could infer the missing values.**
Perform a predictive modeling analysis on this same dataset using the a) k-NN technique (for k=3) and b) Logistic Regression. Please be specific about what other parameters you specified for your models.
Present a brief overview of your predictive modeling process, explorations, and discuss your results.
Compare the k-NN model with the Logistic Regression model: which performs better? Make sure you present information about the model "goodness" (i.e., confusion matrix, predictive accuracy, precision, recall, f-measure). Please be clear about any assumptions you might make when you choose the best performing model.
Please show screenshots of the models you have built with Rapidminer and Python, show screenshots of the performance results, and the parameters you have specified.

I have listed the definition and importance of the performance metrics here, and will be referencing this throughout our both models on both Python and Rapidminer:
Accuracy: % of total predictions that are correct.
Precision $_{\text{Malignant}}$: % of total Malignant predictions that are Malignant in reality.
Recall $_{\text{Malignant}}$: The % of total Malignant cases that are accurately predicted.
F-Measure $_{\text{Malignant}}$: A harmonic evaluation metric for both Precision and Recall, where 0 is really bad and 1 is the best.

Python k-NN Prediction
The process for k-NN prediction in python is simple. I heavily referenced our in-class example on k-nn prediction. First, I made sure to import all the necessary packages, set the project root directory and import the dataset itself. Because I know the dataset has no headers, I made sure to set header=None and labeled all the columns during import.


jupyter HW2 Python Code Last Checkpoint: an hour ago (autosaved)
Logou
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

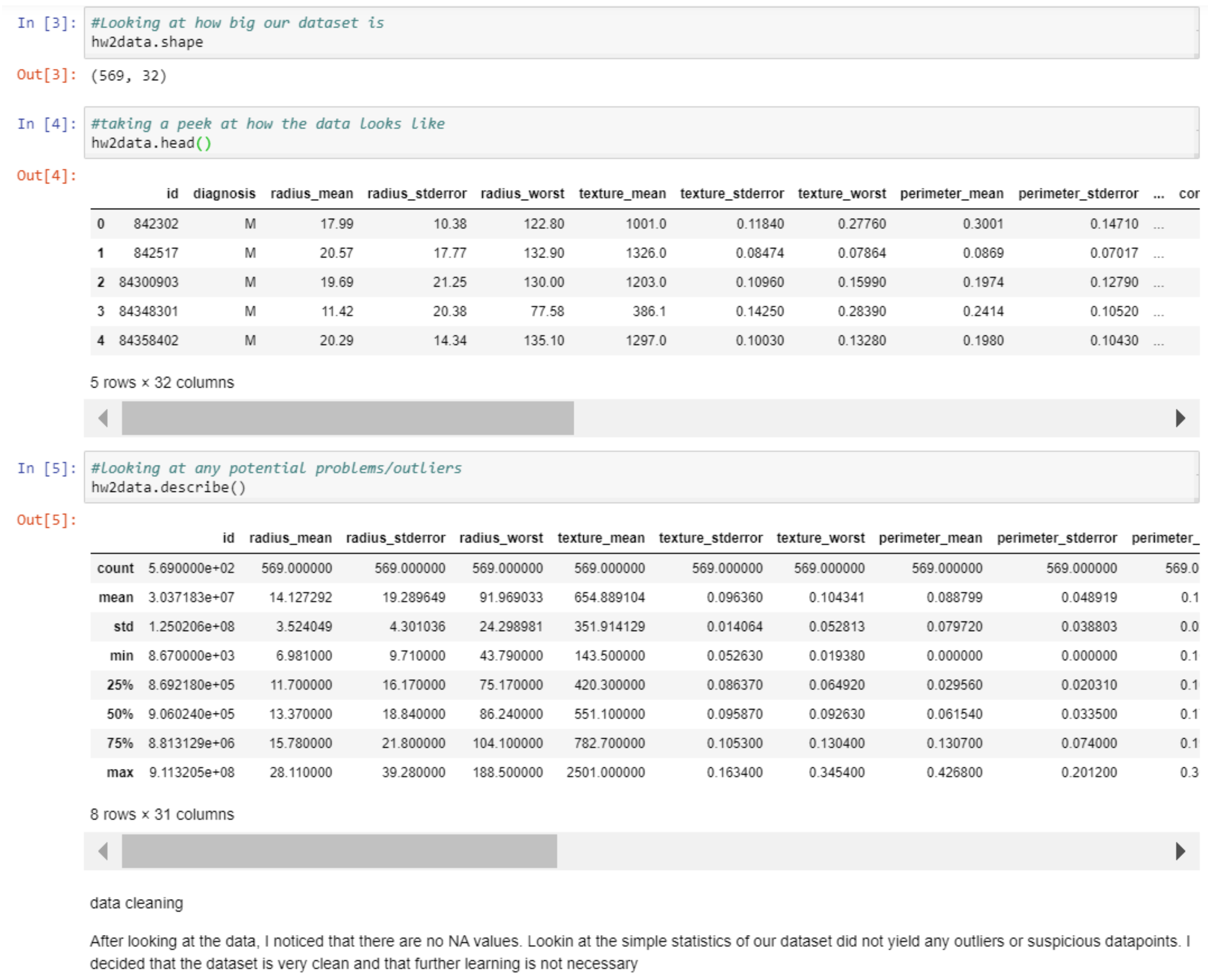
kNN Prediction

In [1]:
```
# To write a Python 2/3 compatible codebase, the first step is to add this line to the top of each module
from __future__ import division, print_function, unicode_literals

from IPython.display import Image
from sklearn import linear_model, neighbors, datasets, metrics        # The sklearn.linear_model module implements generalized l
from matplotlib.colors import ListedColormap # Learn more about matplotlib.colors here https://matplotlib.org/3.1.1/api/colors_a
import matplotlib.pyplot as plt #pyplot is matplotlib's plotting framework https://matplotlib.org/users/pyplot_tutorial.html
import matplotlib
import scipy as sp # sp is an alias pointing to scipy
import numpy as np
import pandas as pd # pd is an alias point to pandas
import warnings
warnings.filterwarnings("ignore")

#----------
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split
from sklearn.tree import export_graphviz
# If you don't have graphviz package, you need to install it https://anaconda.org/anaconda/graphviz
# How to install Graphviz with Anaconda 1
# conda install -c anaconda graphviz
!pip install graphviz
import itertools
import graphviz
import os

# Seed the generator to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline

# Dynamically change the default rc settings in a python script
# See documentation for a complete list of parameters https://matplotlib.org/users/customizing.html
plt.rcParams['axes.labelsize'] = 14  # fontsize of the x any y labels
plt.rcParams['xtick.labelsize'] = 12 # fontsize of the tick labels
plt.rcParams['ytick.labelsize'] = 12 # fontsize of the tick labels
#-----------------


# Matplotlib inline allows the output of plotting commands will be displayed inline
%matplotlib inline

# Where to save the figures
PROJECT_ROOT_DIR = "C:\\Users\\carlj\\OneDrive\\Documents\\School-MSBA\\Classes\\Fall\\Intro. to Business Analytics\\HW2"
```

```
Requirement already satisfied: graphviz in c:\programdata\anaconda3\lib\site-packages (0.13)
```

In [2]:
```
############################################ Imports ############################################
hw2data = pd.read_csv("wdbc.data", header=None, names=["id","diagnosis","radius_mean","radius_stderror","radius_worst","texture_

#hw2data.rename(columns={0:"id",1:"diagnosis"},inplace=True)
```

We then examined our dataset. Everything looks good, this is a very clean dataset.



In [3]:
```
#Looking at how big our dataset is
hw2data.shape
```

Out[3]:
```
(569, 32)
```

In [4]:
```
#taking a peek at how the data looks like
hw2data.head()
```

Out[4]:

	id	diagnosis	radius_mean	radius_stderror	radius_worst	texture_mean	texture_stderror	texture_worst	perimeter_mean	perimeter_stderror	...	cor
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	

5 rows × 32 columns

In [5]:
```
#looking at any potential problems/outliers
hw2data.describe()
```

Out[5]:

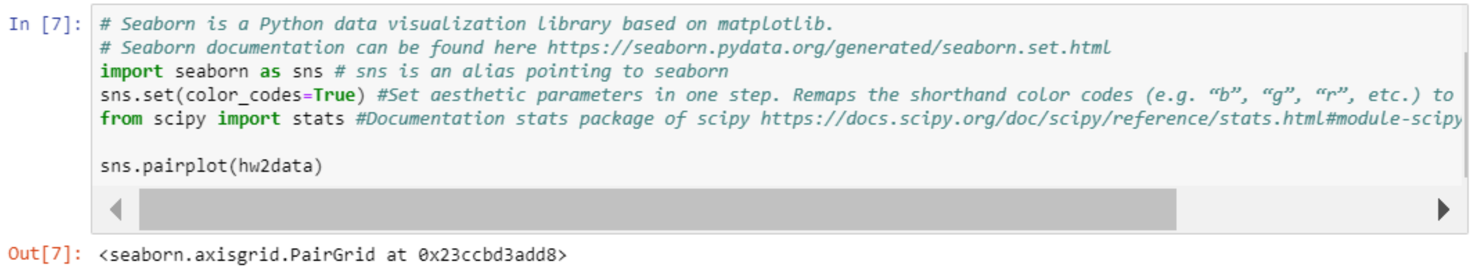
	id	radius_mean	radius_stderror	radius_worst	texture_mean	texture_stderror	texture_worst	perimeter_mean	perimeter_stderror	perimeter_
count	5.690000e+02	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.0
mean	3.037183e+07	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	0.1
std	1.250206e+08	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.0
min	8.670000e+03	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	0.1
25%	8.692180e+05	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	0.020310	0.1
50%	9.060240e+05	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	0.1
75%	8.813129e+06	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	0.1
max	9.113205e+08	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.3

8 rows × 31 columns

data cleaning

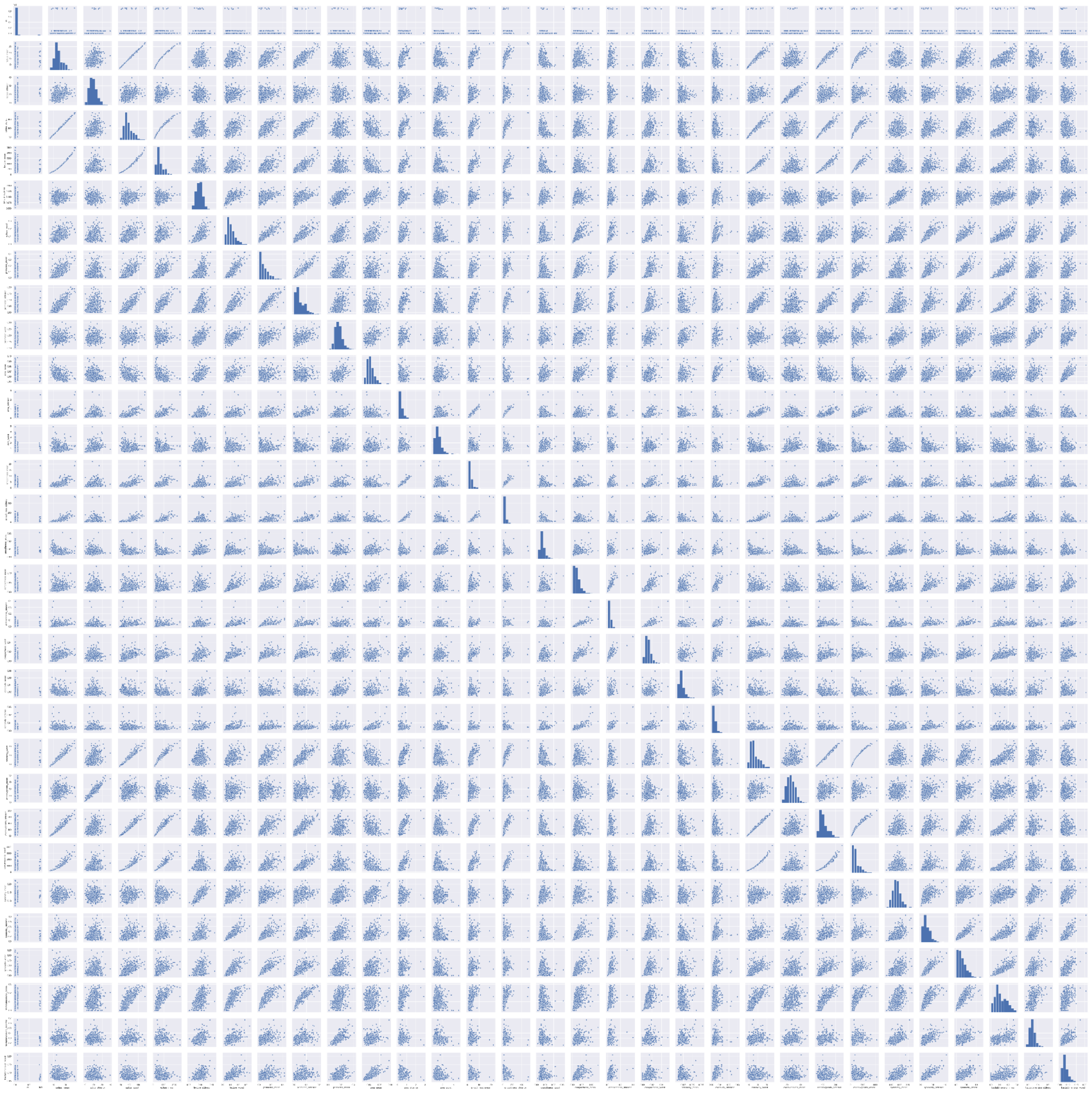
After looking at the data, I noticed that there are no NA values. Lookin at the simple statistics of our dataset did not yield any outliers or suspicious datapoints. I decided that the dataset is very clean and that further learning is not necessary

We then ran pairplot, which is only readable if we zoom in. I went through all the plots and noticed that many of them are highly correlated. This is very understandable as every 3 variables are based the same base variable. (3*10)



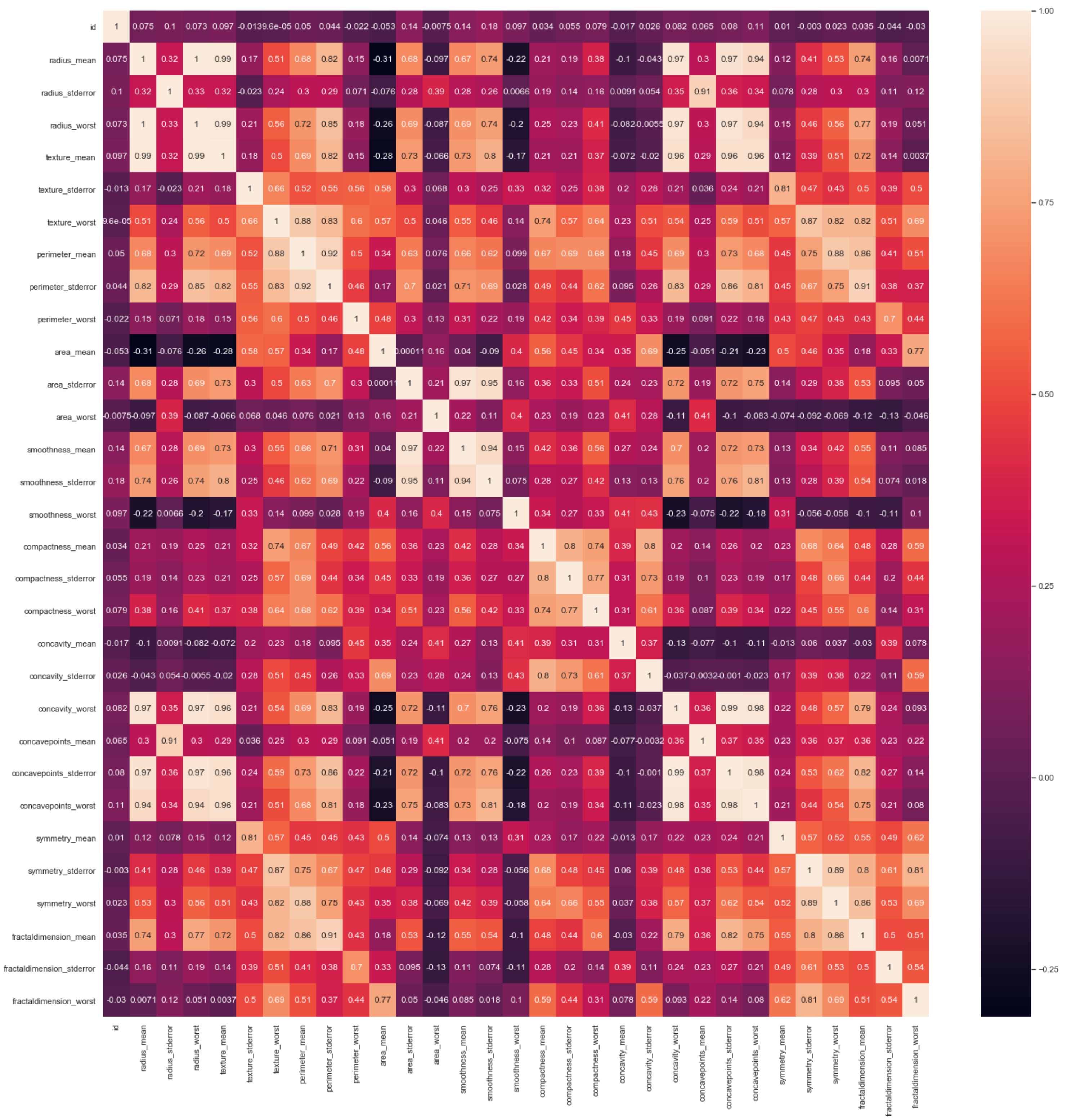
In [7]:
```
# Seaborn is a Python data visualization library based on matplotlib.
# Seaborn documentation can be found here https://seaborn.pydata.org/generated/seaborn.set.html
import seaborn as sns # sns is an alias pointing to seaborn
sns.set(color_codes=True) #Set aesthetic parameters in one step. Remaps the shorthand color codes (e.g. “b”, “g”, “r”, etc.) to
from scipy import stats #Documentation stats package of scipy https://docs.scipy.org/doc/scipy/reference/stats.html#module-scipy

sns.pairplot(hw2data)
```

Out[7]:
```
<seaborn.axisgrid.PairGrid at 0x23ccbd3add8>
```

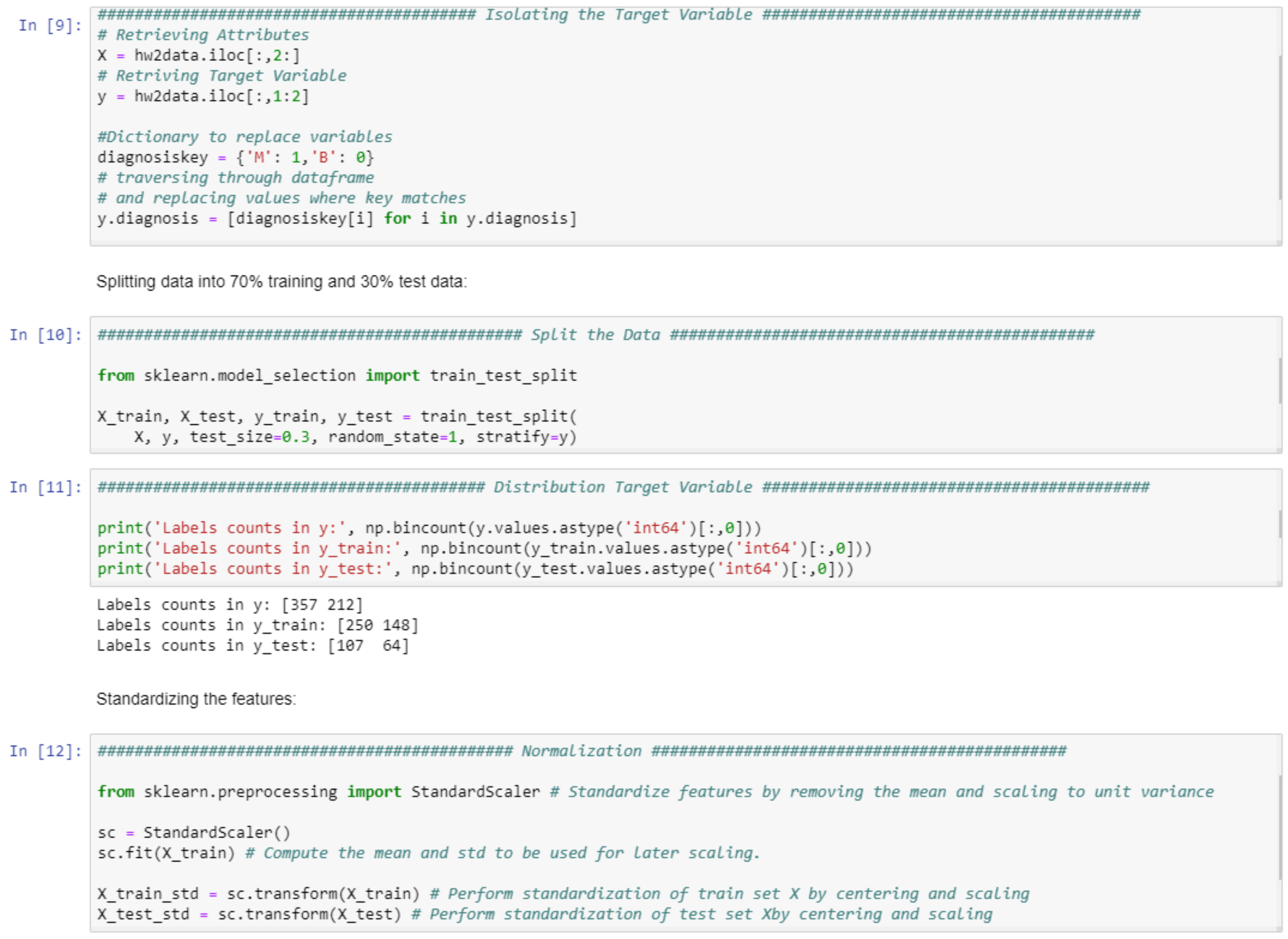


I also looked at a heatmap of our dataset, which tales a similar story for correlation

In [8]:
```
corrmatrix = hw2data.corr()
top_correlated_features = corrmatrix.index
plt.figure(figsize=(25,25))
#plot heat map
plot=sns.heatmap(hw2data[top_correlated_features].corr(),annot=True)
```



Next, we isolated the target variable, converted all the Malignant and Benign cases to 1 and 0 respectively, split the data in to 70% training and 30% test, checked the distribution of target variable and standardized the features by normalizing.



```
In [9]: ######################################### Isolating the Target Variable #########################################
        # Retrieving Attributes
        X = hw2data.iloc[:,2:]
        # Retriving Target Variable
        y = hw2data.iloc[:,1:2]

        #Dictionary to replace variables
        diagnosiskey = {'M': 1,'B': 0}
        # traversing through dataframe
        # and replacing values where key matches
        y.diagnosis = [diagnosiskey[i] for i in y.diagnosis]
```

Splitting data into 70% training and 30% test data:

```
In [10]: ############################################## Split the Data ##############################################

         from sklearn.model_selection import train_test_split

         X_train, X_test, y_train, y_test = train_test_split(
             X, y, test_size=0.3, random_state=1, stratify=y)
```

```
In [11]: ########################################## Distribution Target Variable ##########################################

         print('Labels counts in y:', np.bincount(y.values.astype('int64')[:,0]))
         print('Labels counts in y_train:', np.bincount(y_train.values.astype('int64')[:,0]))
         print('Labels counts in y_test:', np.bincount(y_test.values.astype('int64')[:,0]))

         Labels counts in y: [357 212]
         Labels counts in y_train: [250 148]
         Labels counts in y_test: [107  64]
```

Standardizing the features:

```
In [12]: ############################################## Normalization ##############################################

         from sklearn.preprocessing import StandardScaler # Standardize features by removing the mean and scaling to unit variance

         sc = StandardScaler()
         sc.fit(X_train) # Compute the mean and std to be used for later scaling.

         X_train_std = sc.transform(X_train) # Perform standardization of train set X by centering and scaling
         X_test_std = sc.transform(X_test) # Perform standardization of test set Xby centering and scaling
```

I then trained the model with n neighbors set to 3 and p to 4. I read the documentation on p by goggling online but I did not really understand what it meant, so I decided to keep it at 4 since it can be any positive number and changing it didn't seem to affect the model much. I then ran model evaluation, the results of which can be seen below:

In [33]:
```python
############################################### Train the Model ##############################################

# KNeighborsClassifier is a classifier implementing the k-nearest neighbors vote.
# Learn more about it here https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

# Set parameters of KNeighborsClassifier
knn = neighbors.KNeighborsClassifier(n_neighbors=3, #n_neighbors is the k in the kNN
                           p=4,
                           metric='minkowski') #The default metric is minkowski, which is a generalization of the Euclidean dist
                                               # with p=2 is equivalent to the standard Euclidean distance.
                                               # with p=1 is equivalent to the Mahattan distance.

# Train the model
knn = knn.fit(X_train_std, y_train)
```

In [34]:
```python
############################################### Evaluate the Model ##############################################

# The sklearn.metrics module includes score functions, performance metrics and pairwise metrics
# and distance computations.
# https://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics
from sklearn.metrics import accuracy_score, f1_score, cohen_kappa_score, classification_report

# Estimate the predicted values by applying the kNN algorithm
y_pred = knn.predict(X_test_std)
y_pred_insample = knn.predict(X_train_std)

# Accuracy
print('Accuracy (out-of-sample): %.6f' % accuracy_score(y_test, y_pred))
print('Accuracy (in-sample)    : %.6f' % accuracy_score(y_train, y_pred_insample))
print()

# F1 score
print('F1 score (out-of-sample): %.6f' % f1_score(y_test, y_pred, average='macro'))
print('F1 score (in-sample)    : %.6f' % f1_score(y_train, y_pred_insample, average='macro'))
print()
# Kappa score
print('Kappa score (out-of-sample): %.6f' % cohen_kappa_score(y_test, y_pred))
print('Kappa score (in-sample)    : %.6f' % cohen_kappa_score(y_train, y_pred_insample))
print()

cnf_matrix = confusion_matrix(y_test, y_pred)
np.set_printoptions(precision=2)

print("Confusion Matrix:")
print(confusion_matrix(y_test,y_pred))
print()

# Build a text report showing the main classification metrics (out-of-sample performance)
print("Summary Statistics")
print(classification_report(y_test, y_pred))
print()
```

```
Accuracy (out-of-sample): 0.959064
Accuracy (in-sample)    : 0.979899

F1 score (out-of-sample): 0.955563
F1 score (in-sample)    : 0.978302

Kappa score (out-of-sample): 0.911208
Kappa score (in-sample)    : 0.956614

Confusion Matrix:
[[106   1]
 [  6  58]]

Summary Statistics
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       107
           1       0.98      0.91      0.94        64

   micro avg       0.96      0.96      0.96       171
   macro avg       0.96      0.95      0.96       171
weighted avg       0.96      0.96      0.96       171
```

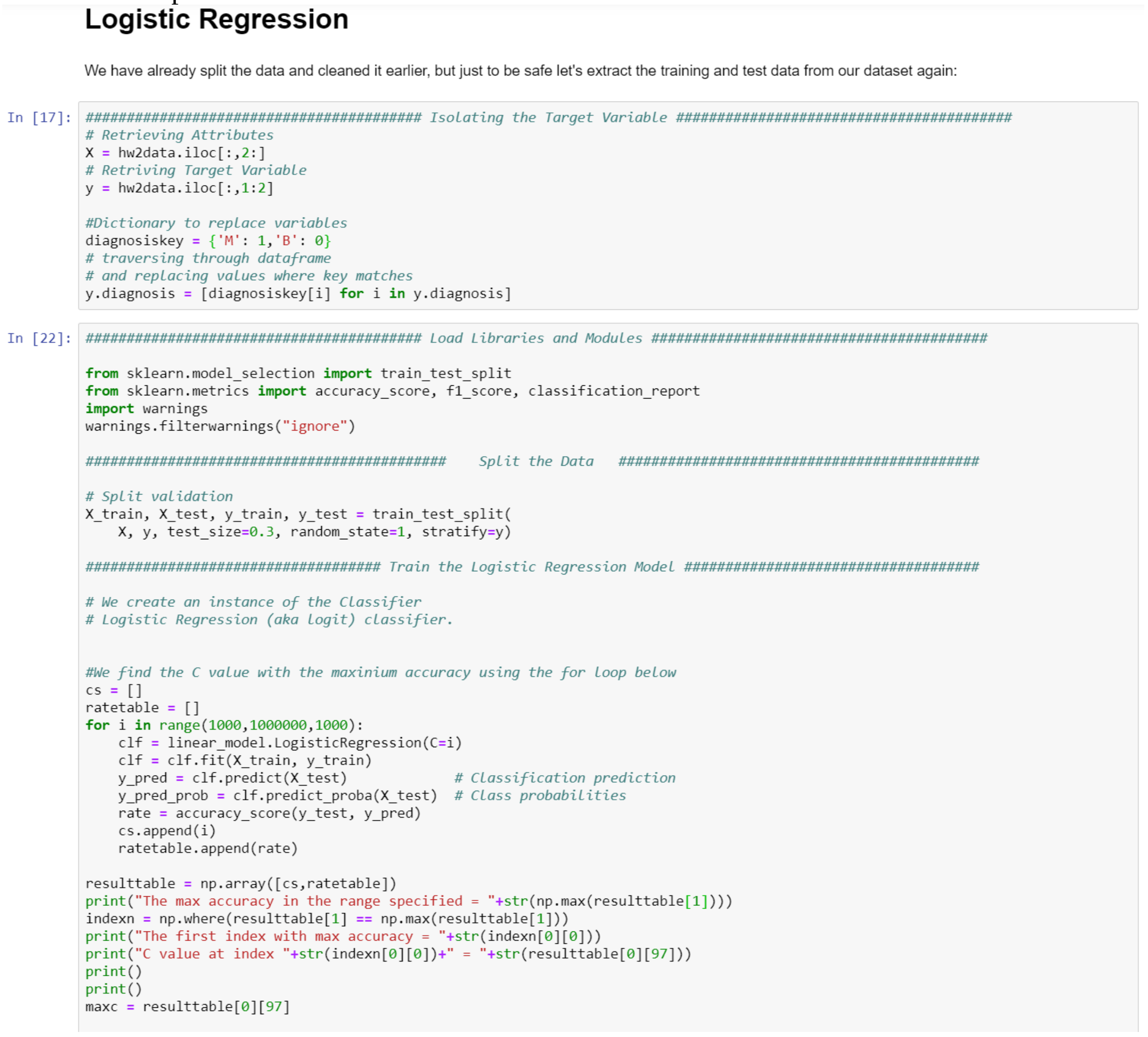
An out-of-sample accuracy of 95.91% is very high, with precision, recall and f-score at 98%, 91% and 94% respectively. (We are more interested in Malignant cases, so it is set to in the results above). The meaning of these metrics are described at the beginning of question 5 answer, and I will discuss how this model compares with the logistic regression model below.

I understand that my model might suffer from overfitting due to the curse of dimensionality, but I think a k of 3 is reasonable given how regularization is not applicable in k-NN. The difference between out-of-sample and in-sample accuracy is not too big, so I decided to keep everything and not use feature selection to reduce my dimensionality.

Logistic Regression

Logistic regression follows a similar process as above. Because both k-NN and Logistic Regression are in the same notebook, we don't have to redo all the imports, examination and cleaning. Re-defining variables and target and splitting the dataset into train and test are redundant steps, but I did them just to keep things consistent. I then created a for loop that ran through all the variables for C. I noticed that the maximum accuracy is **repeated many times at higher and higher C values,** so I chose the **first C value with the max accuracy**. I did this because we know that C is lambda/1 and that a bigger C means a smaller lambda which equals to viewing penalty as less important, which means a total less penalty on overfitting. By finding the minimum C that yields the max accuracy, the model should in theory have the highest penalty on overfitting while yielding great results. I then plugged the highest C I found into my actual model, applied it, and evaluated its performance.

I want to state that while I understand our model has high multicollinearity, our test and train performances are both pretty high. In addition, while L1 is designed to help us find and remove variables to reduce multicollinearity, the L1 model actually performs worse than our L2 model, which cannot help us. This is why I decided to keep all the variables within the model.

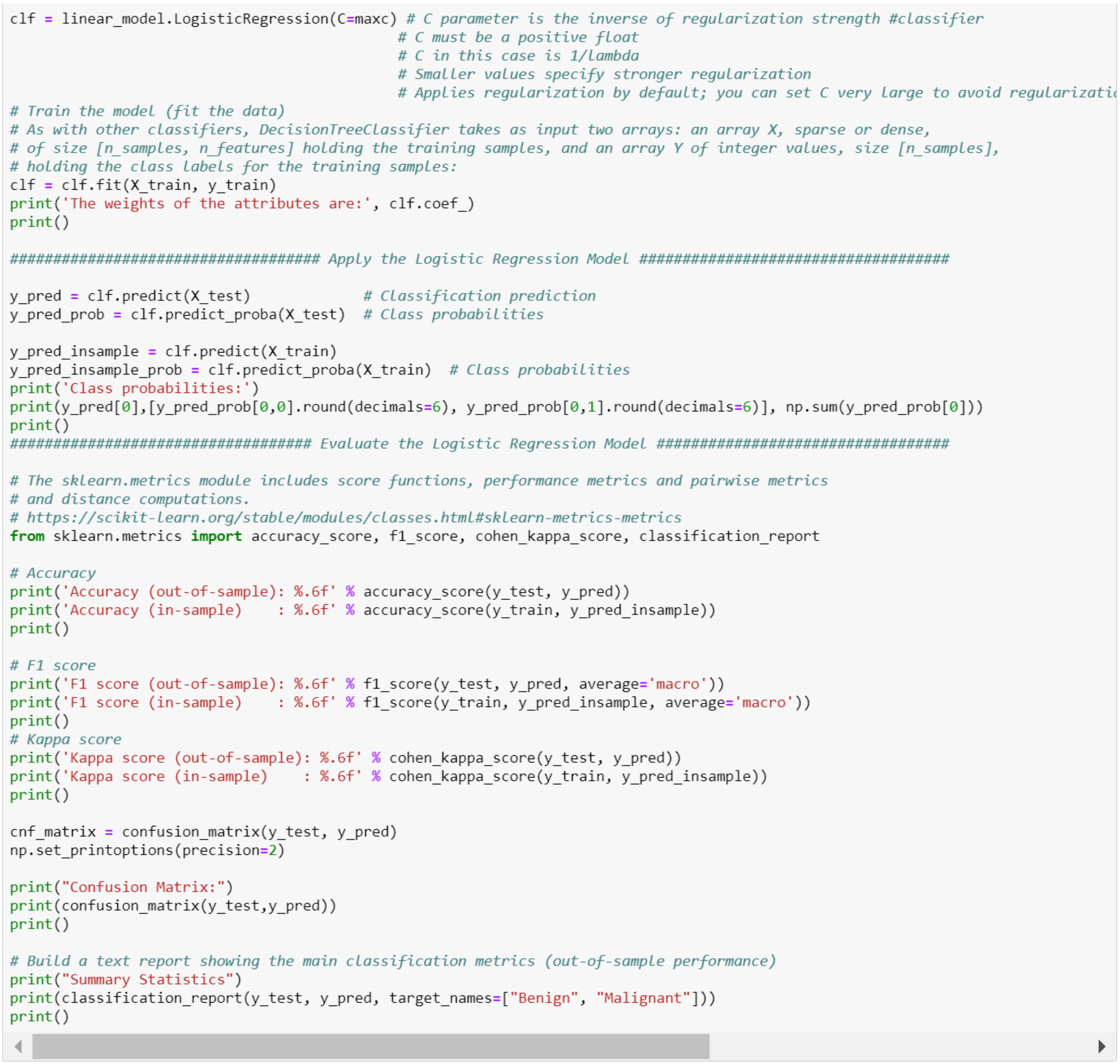


Logistic Regression

We have already split the data and cleaned it earlier, but just to be safe let's extract the training and test data from our dataset again:

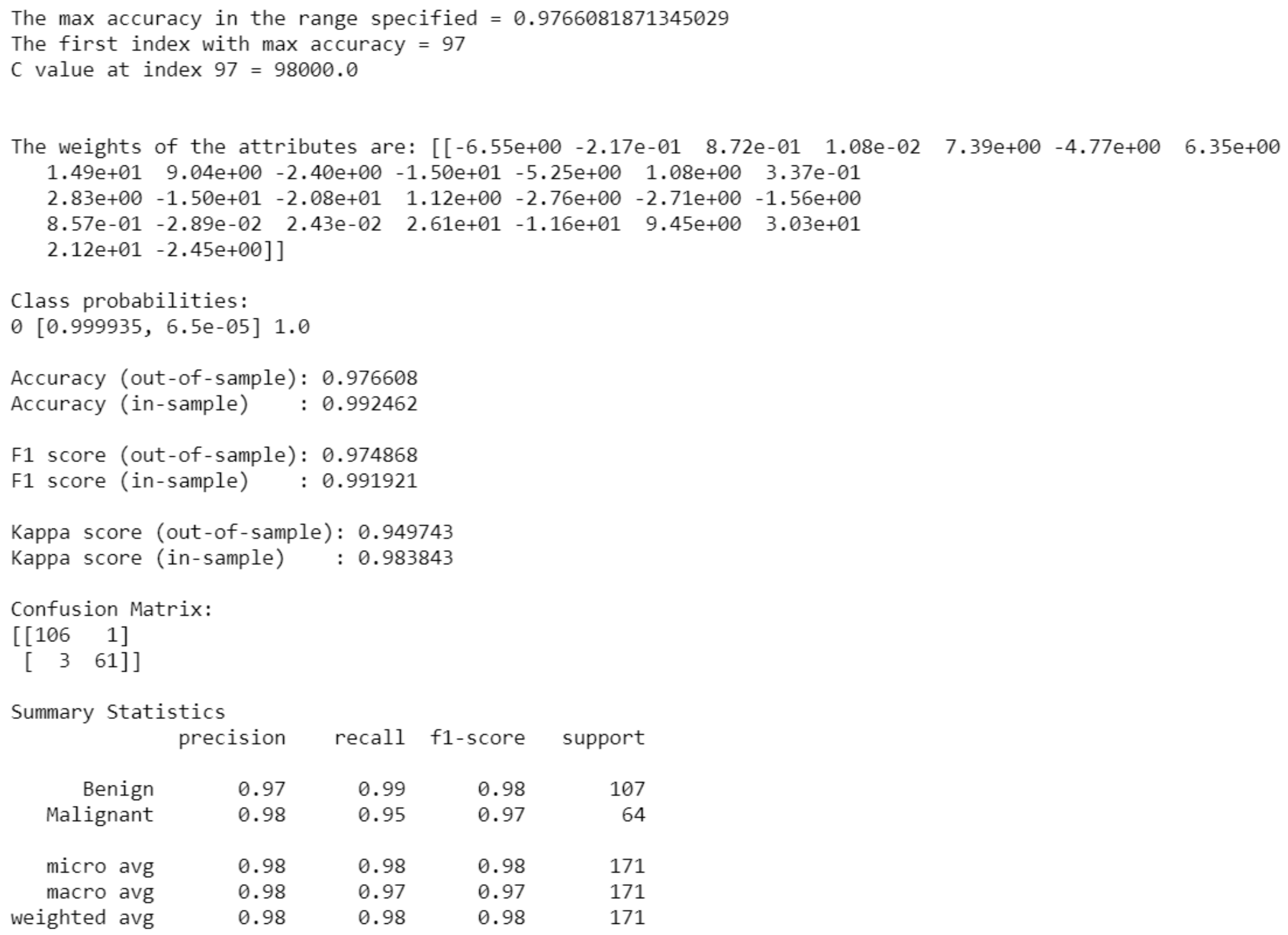
```
In [17]: ######################################### Isolating the Target Variable #########################################
         # Retrieving Attributes
         X = hw2data.iloc[:,2:]
         # Retriving Target Variable
         y = hw2data.iloc[:,1:2]

         #Dictionary to replace variables
         diagnosiskey = {'M': 1,'B': 0}
         # traversing through dataframe
         # and replacing values where key matches
         y.diagnosis = [diagnosiskey[i] for i in y.diagnosis]
```

```
In [22]: ######################################### Load Libraries and Modules #########################################

         from sklearn.model_selection import train_test_split
         from sklearn.metrics import accuracy_score, f1_score, classification_report
         import warnings
         warnings.filterwarnings("ignore")

         ############################################    Split the Data   ############################################

         # Split validation
         X_train, X_test, y_train, y_test = train_test_split(
             X, y, test_size=0.3, random_state=1, stratify=y)

         #################################### Train the Logistic Regression Model ####################################

         # We create an instance of the Classifier
         # Logistic Regression (aka logit) classifier.


         #We find the C value with the maxinium accuracy using the for loop below
         cs = []
         ratetable = []
         for i in range(1000,1000000,1000):
             clf = linear_model.LogisticRegression(C=i)
             clf = clf.fit(X_train, y_train)
             y_pred = clf.predict(X_test)            # Classification prediction
             y_pred_prob = clf.predict_proba(X_test)  # Class probabilities
             rate = accuracy_score(y_test, y_pred)
             cs.append(i)
             ratetable.append(rate)

         resulttable = np.array([cs,ratetable])
         print("The max accuracy in the range specified = "+str(np.max(resulttable[1])))
         indexn = np.where(resulttable[1] == np.max(resulttable[1]))
         print("The first index with max accuracy = "+str(indexn[0][0]))
         print("C value at index "+str(indexn[0][0])+" = "+str(resulttable[0][97]))
         print()
         print()
         maxc = resulttable[0][97]
```



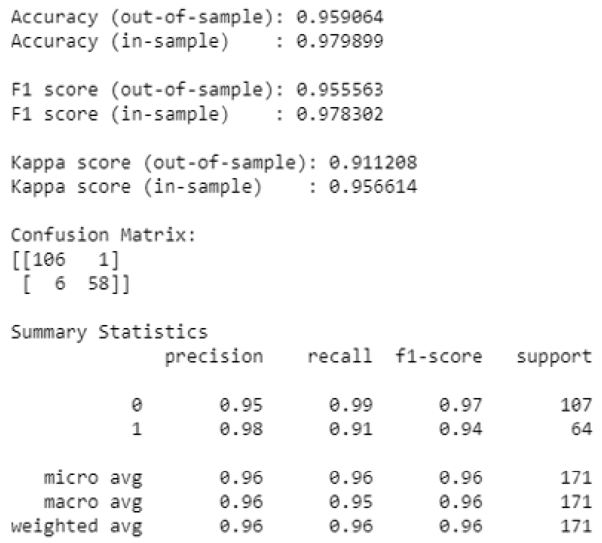
```python
clf = linear_model.LogisticRegression(C=maxc) # C parameter is the inverse of regularization strength #classifier
                                             # C must be a positive float
                                             # C in this case is 1/lambda
                                             # Smaller values specify stronger regularization
                                             # Applies regularization by default; you can set C very large to avoid regularizati
# Train the model (fit the data)
# As with other classifiers, DecisionTreeClassifier takes as input two arrays: an array X, sparse or dense,
# of size [n_samples, n_features] holding the training samples, and an array Y of integer values, size [n_samples],
# holding the class labels for the training samples:
clf = clf.fit(X_train, y_train)
print('The weights of the attributes are:', clf.coef_)
print()

#################################### Apply the Logistic Regression Model ####################################

y_pred = clf.predict(X_test)             # Classification prediction
y_pred_prob = clf.predict_proba(X_test)  # Class probabilities

y_pred_insample = clf.predict(X_train)
y_pred_insample_prob = clf.predict_proba(X_train)  # Class probabilities
print('Class probabilities:')
print(y_pred[0],[y_pred_prob[0,0].round(decimals=6), y_pred_prob[0,1].round(decimals=6)], np.sum(y_pred_prob[0]))
print()
################################### Evaluate the Logistic Regression Model ##################################

# The sklearn.metrics module includes score functions, performance metrics and pairwise metrics
# and distance computations.
# https://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics
from sklearn.metrics import accuracy_score, f1_score, cohen_kappa_score, classification_report

# Accuracy
print('Accuracy (out-of-sample): %.6f' % accuracy_score(y_test, y_pred))
print('Accuracy (in-sample)    : %.6f' % accuracy_score(y_train, y_pred_insample))
print()

# F1 score
print('F1 score (out-of-sample): %.6f' % f1_score(y_test, y_pred, average='macro'))
print('F1 score (in-sample)    : %.6f' % f1_score(y_train, y_pred_insample, average='macro'))
print()
# Kappa score
print('Kappa score (out-of-sample): %.6f' % cohen_kappa_score(y_test, y_pred))
print('Kappa score (in-sample)    : %.6f' % cohen_kappa_score(y_train, y_pred_insample))
print()

cnf_matrix = confusion_matrix(y_test, y_pred)
np.set_printoptions(precision=2)

print("Confusion Matrix:")
print(confusion_matrix(y_test,y_pred))
print()

# Build a text report showing the main classification metrics (out-of-sample performance)
print("Summary Statistics")
print(classification_report(y_test, y_pred, target_names=["Benign", "Malignant"]))
print()
```

<u>Logistic Regression Results:</u>



```
The max accuracy in the range specified = 0.9766081871345029
The first index with max accuracy = 97
C value at index 97 = 98000.0


The weights of the attributes are: [[-6.55e+00 -2.17e-01  8.72e-01  1.08e-02  7.39e+00 -4.77e+00  6.35e+00
   1.49e+01  9.04e+00 -2.40e+00 -1.50e+01 -5.25e+00  1.08e+00  3.37e-01
   2.83e+00 -1.50e+01 -2.08e+01  1.12e+00 -2.76e+00 -2.71e+00 -1.56e+00
   8.57e-01 -2.89e-02  2.43e-02  2.61e+01 -1.16e+01  9.45e+00  3.03e+01
   2.12e+01 -2.45e+00]]

Class probabilities:
0 [0.999935, 6.5e-05] 1.0

Accuracy (out-of-sample): 0.976608
Accuracy (in-sample)    : 0.992462

F1 score (out-of-sample): 0.974868
F1 score (in-sample)    : 0.991921

Kappa score (out-of-sample): 0.949743
Kappa score (in-sample)    : 0.983843

Confusion Matrix:
[[106   1]
 [  3  61]]

Summary Statistics
              precision    recall  f1-score   support

      Benign       0.97      0.99      0.98       107
   Malignant       0.98      0.95      0.97        64

   micro avg       0.98      0.98      0.98       171
   macro avg       0.98      0.97      0.97       171
weighted avg       0.98      0.98      0.98       171
```

k-NN Results



```
Accuracy (out-of-sample): 0.959064
Accuracy (in-sample)    : 0.979899

F1 score (out-of-sample): 0.955563
F1 score (in-sample)    : 0.978302

Kappa score (out-of-sample): 0.911208
Kappa score (in-sample)    : 0.956614

Confusion Matrix:
[[106   1]
 [  6  58]]

Summary Statistics
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       107
           1       0.98      0.91      0.94        64

   micro avg       0.96      0.96      0.96       171
   macro avg       0.96      0.95      0.96       171
weighted avg       0.96      0.96      0.96       171
```

Across, the board, our logistic regression model outperformed our k-NN model, with out-of-sample accuracy, precision, recall and f-score at 97.66%, 98%, 95% and 97% respectively. This means that logistic regression model is not only makes a higher % of total correct predictions, but it also makes less False positive predictions and less false negative predictions.
I also tried to see if doing cross-validation with 5 folds would improve my logistic regression model, but the resulting accuracy is actually lower, which was disappointing to see.

```
In [24]: ################################### Logistic Regression with Cross Validation ###################################

         from sklearn import metrics
         from sklearn.linear_model import LogisticRegression
         from sklearn.model_selection import cross_val_score

         # Fit model to all the data
         clf_lr = linear_model.LogisticRegression(C=maxc)

         # Evaluate performance with cross-validation
         # Read more about cross_val_score in the following link
         # http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html#sklearn.model_selection.cross_va

         # Accuracy
         scores=cross_val_score(clf_lr, X, y, cv=5)
         print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
         print(scores)

         # F-1 scores
         scores_f1=cross_val_score(clf_lr, X, y, cv=5, scoring='f1_macro')
         print("F1-score: %0.2f (+/- %0.2f)" % (scores_f1.mean(), scores_f1.std() * 2))# returns an array of scores of the estimator for e
         print(scores_f1)
```

```
Accuracy: 0.95 (+/- 0.02)
[0.95 0.94 0.96 0.95 0.96]
F1-score: 0.95 (+/- 0.02)
[0.94 0.93 0.95 0.94 0.96]
```

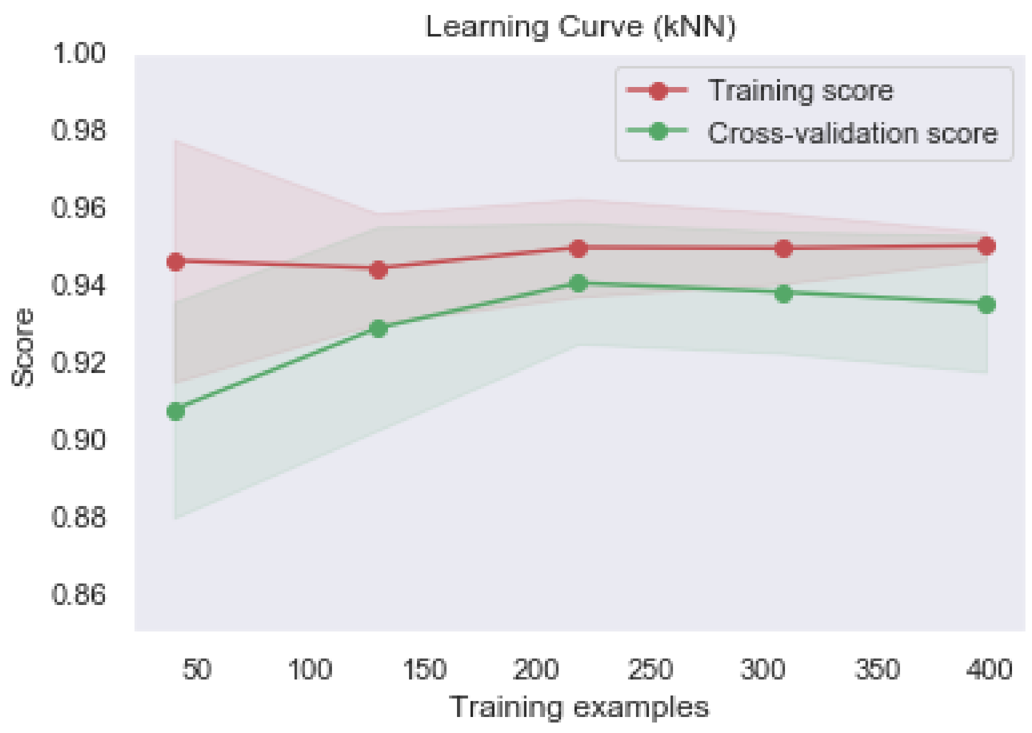
```
In [38]: # Use all features of the data
         scores = cross_val_score(clf_lr, X, y, cv=5, scoring='f1_macro')
         print(scores)
         print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

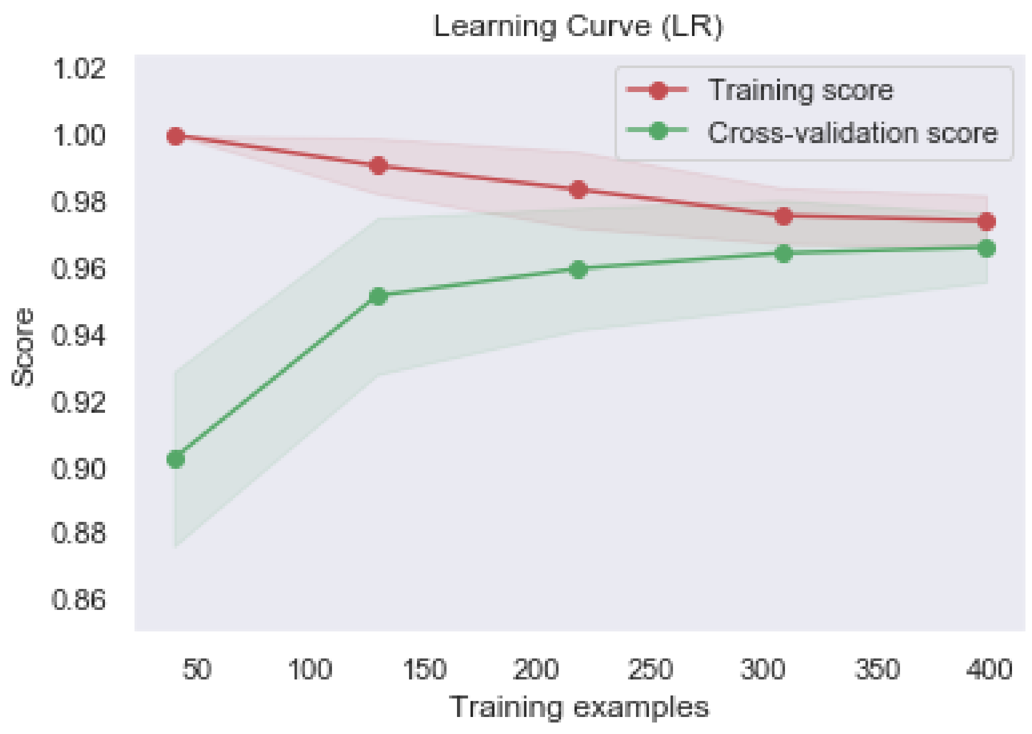
```
[0.94 0.93 0.95 0.94 0.96]
Accuracy: 0.95 (+/- 0.02)
```

This was very interesting, so I decided to plot the learning curves of both k-NN and Logistic Regression. Doing so yielded results that made sense. It seems like cross-validation consistently underperforms across dataset size.

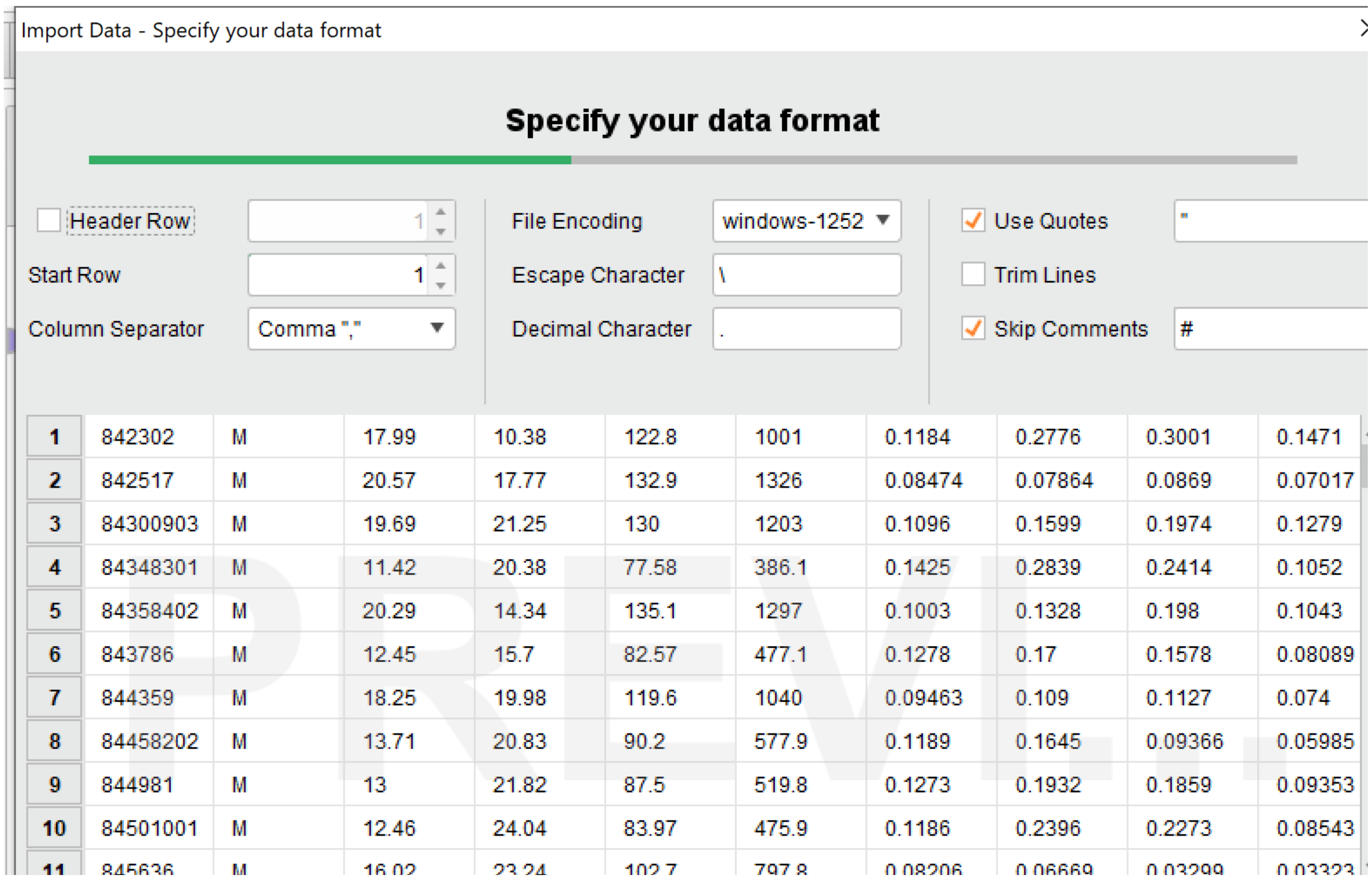
In [26]:
```
################################### Define function that plots Learning Curves ###################################

def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,
                        n_jobs=1, train_sizes=np.linspace(.1, 1.0, 5)): # np.linspace(.1, 1.0, 5) will return evenly
                                                                         # spaced 5 numbers from 0.1 to 1.0
                        # n_jobs is the number of CPUs to use to do the computation.
    # Visualization patamters
    plt.figure()
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training examples")
    plt.ylabel("Score")

    # Estimate train and test score for different training set sizes
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes) # learning_curve Determines cross-validated
                                                                         # training and test scores for different
                                                                         # training set sizes.

    # Estimate statistics of train and test scores (mean, std)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    plt.grid()

    # Fill the area around the mean scores with standard deviation info
    plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r") # Fill for train set scores

    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="g")  # Fill for test set scores

    # Visualization parameters that will allow us to distinguish train set scores from test set scores
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-validation score")

    plt.legend(loc="best")
    return plt
```

In [40]:
```
####################################### Plot Learning Curves (LR and kNN) #######################################

from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit
from sklearn.linear_model import LogisticRegression
from sklearn import neighbors

title = "Learning Curve (LR)"
# SVC is more expensive so we do a lower number of CV iterations:
cv = ShuffleSplit(n_splits=10, test_size=0.3, random_state=42)
estimator = LogisticRegression(C=1e5)
plot_learning_curve(estimator, title, X, y, (0.85, 1.025), cv=cv, n_jobs=4)

plt.show()

title = "Learning Curve (kNN)"
# SVC is more expensive so we do a lower number of CV iterations:
cv = ShuffleSplit(n_splits=10, test_size=0.3, random_state=42)
estimator = neighbors.KNeighborsClassifier(n_neighbors=3) #n_neighbors=
plot_learning_curve(estimator, title, X, y, (0.85, 1.0), cv=cv, n_jobs=4)

plt.show()
```





Before we dive into the actual models, I just wanted to show the data import process. We made sure to uncheck header row to keep all the data



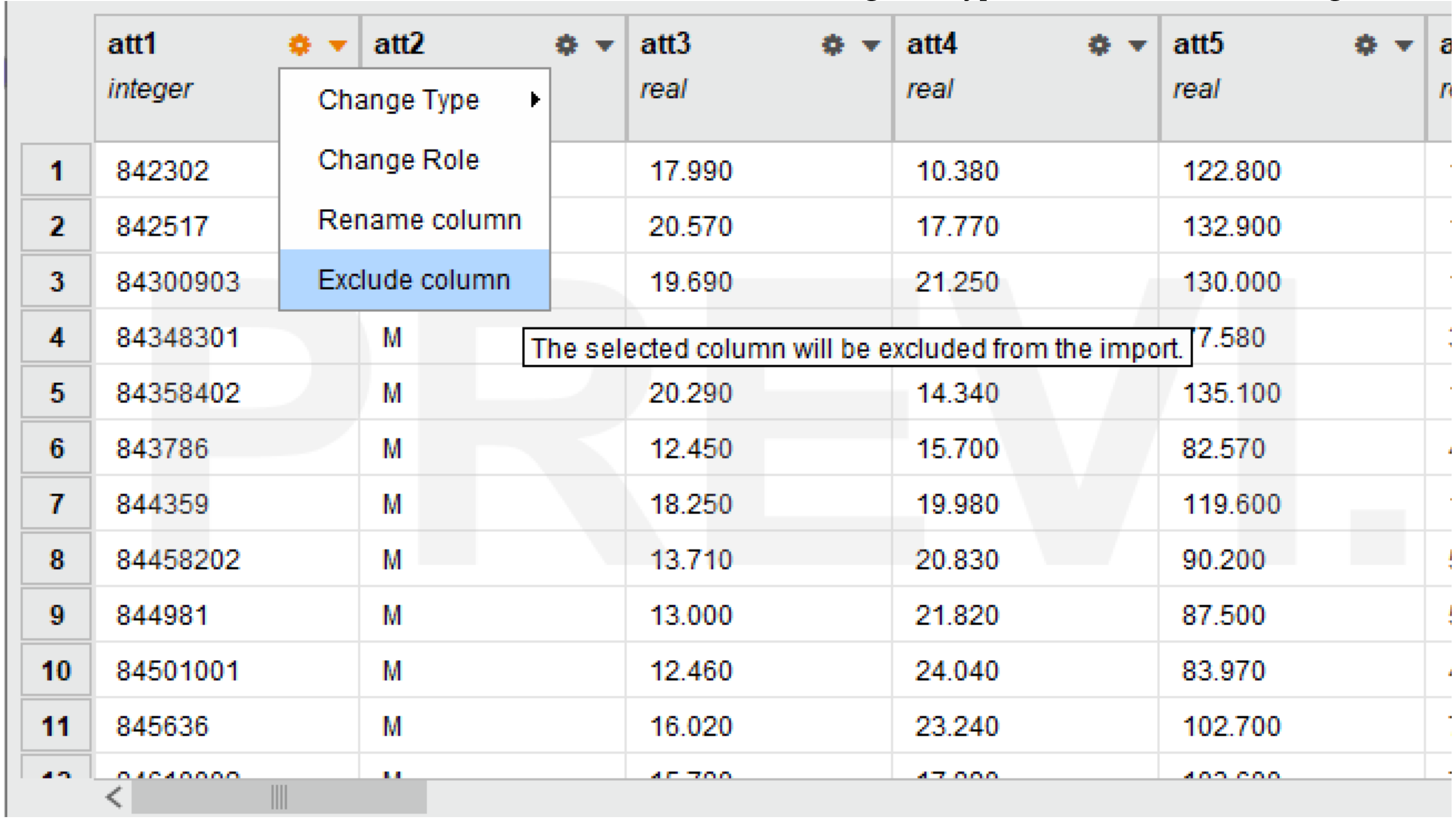
Import Data - Specify your data format

Specify your data format

Header Row	1	File Encoding	windows-1252	Use Quotes ✓	"
Start Row	1	Escape Character	\	Trim Lines	
Column Separator	Comma ","	Decimal Character	.	Skip Comments ✓	#

1	842302	M	17.99	10.38	122.8	1001	0.1184	0.2776	0.3001	0.1471
2	842517	M	20.57	17.77	132.9	1326	0.08474	0.07864	0.0869	0.07017
3	84300903	M	19.69	21.25	130	1203	0.1096	0.1599	0.1974	0.1279
4	84348301	M	11.42	20.38	77.58	386.1	0.1425	0.2839	0.2414	0.1052
5	84358402	M	20.29	14.34	135.1	1297	0.1003	0.1328	0.198	0.1043
6	843786	M	12.45	15.7	82.57	477.1	0.1278	0.17	0.1578	0.08089
7	844359	M	18.25	19.98	119.6	1040	0.09463	0.109	0.1127	0.074
8	84458202	M	13.71	20.83	90.2	577.9	0.1189	0.1645	0.09366	0.05985
9	844981	M	13	21.82	87.5	519.8	0.1273	0.1932	0.1859	0.09353
10	84501001	M	12.46	24.04	83.97	475.9	0.1186	0.2396	0.2273	0.08543
11	845636	M	16.02	23.24	102.7	797.8	0.08206	0.06669	0.03299	0.03323

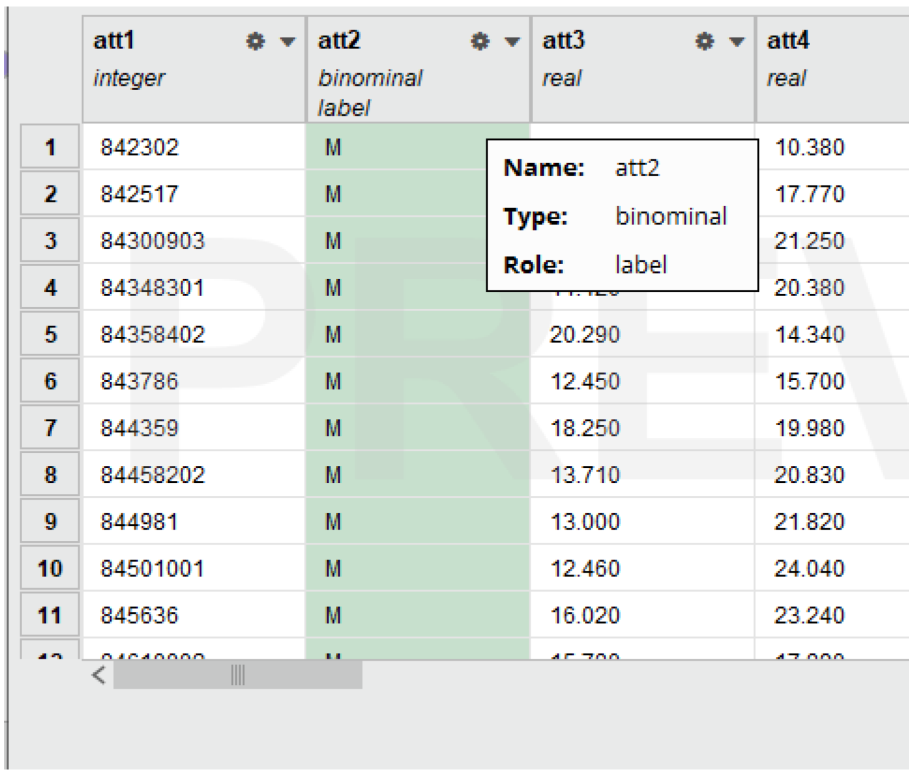
We then made sure to exclude the id column. We could've also changed its type to ID but this makes things easier.



The selected column will be excluded from the import.

	att1 integer	att2	att3 real	att4 real	att5 real
1	842302		17.990	10.380	122.800
2	842517		20.570	17.770	132.900
3	84300903		19.690	21.250	130.000
4	84348301	M			7.580
5	84358402	M	20.290	14.340	135.100
6	843786	M	12.450	15.700	82.570
7	844359	M	18.250	19.980	119.600
8	84458202	M	13.710	20.830	90.200
9	844981	M	13.000	21.820	87.500
10	84501001	M	12.460	24.040	83.970
11	845636	M	16.020	23.240	102.700

Lastly, we made sure to change role of the classification column to label and its type to binomial.



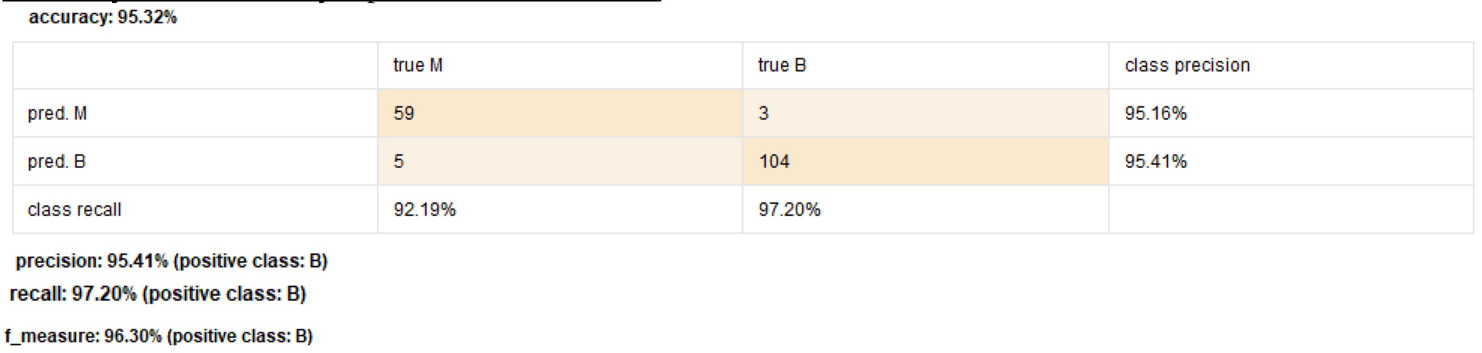
	att1 integer	att2 binominal label	att3 real	att4 real
1	842302	M		10.380
2	842517	M		17.770
3	84300903	M		21.250
4	84348301	M		20.380
5	84358402	M	20.290	14.340
6	843786	M	12.450	15.700
7	844359	M	18.250	19.980
8	84458202	M	13.710	20.830
9	844981	M	13.000	21.820
10	84501001	M	12.460	24.040
11	845636	M	16.020	23.240

Name: att2
Type: binominal
Role: label

Rapidminer k-NN Model

The process for RapidMiner is similar to the one for Python. We simply imitated the model from the in-class example (3b-SplitValidation-kNN-Bank-StorePreProc). We double checked the normalization module and ensured that we conducted a 0.7 to 0.3 split validation. The end result was an accuracy of 95.32%. Rapidminer only gave us the precision, recall and f-measure of benign cases, but we are more interested in the statistics for Malignant cases more, so we refer back to the summary table and find precision and recall at 95.16% and 92.19% respectively, With the two, we can then calculate the f-measure, which is $2 * \frac{Precision*Recall}{Precision+Recall} = 2 * \frac{0.9516*0.9219}{0.9516+0.9219} =$ 0.936515 or 93.65%. All metrics have a very high value, which shows that our model is very robust at predicting Malignant cases. We have discussed the meaning of the evaluation metrics extensively in our previous homework, as well as in the table at the beginning of this answer. We will discuss whether this model will out/underperform the Logistic Regression model in the discussions for that model below. Lastly, we also printed out the AUC graph, which shows our ROC curve for the model. We know the left top most point of the graph is the best result, and it seems like it is at around 0.974.

Summary Statistics of my rapidminer k-NN model:



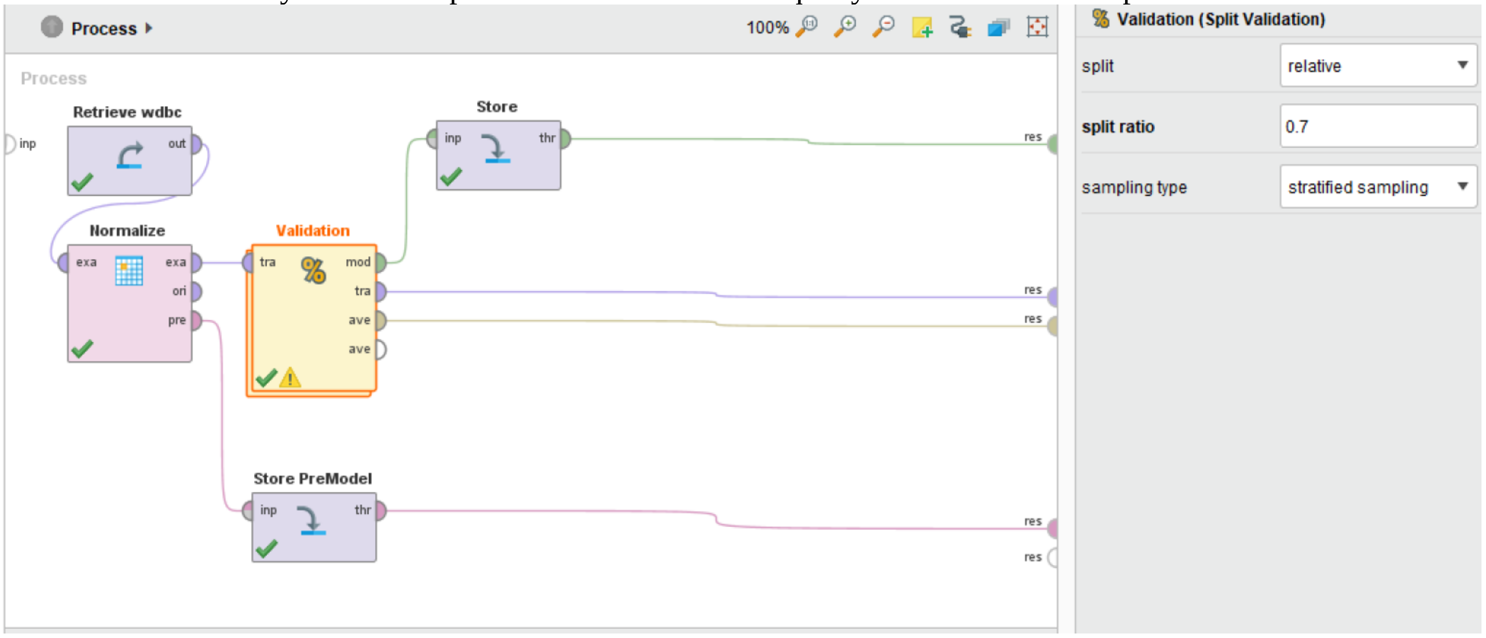
accuracy: 95.32%

	true M	true B	class precision
pred. M	59	3	95.16%
pred. B	5	104	95.41%
class recall	92.19%	97.20%	

precision: 95.41% (positive class: B)
recall: 97.20% (positive class: B)
f_measure: 96.30% (positive class: B)

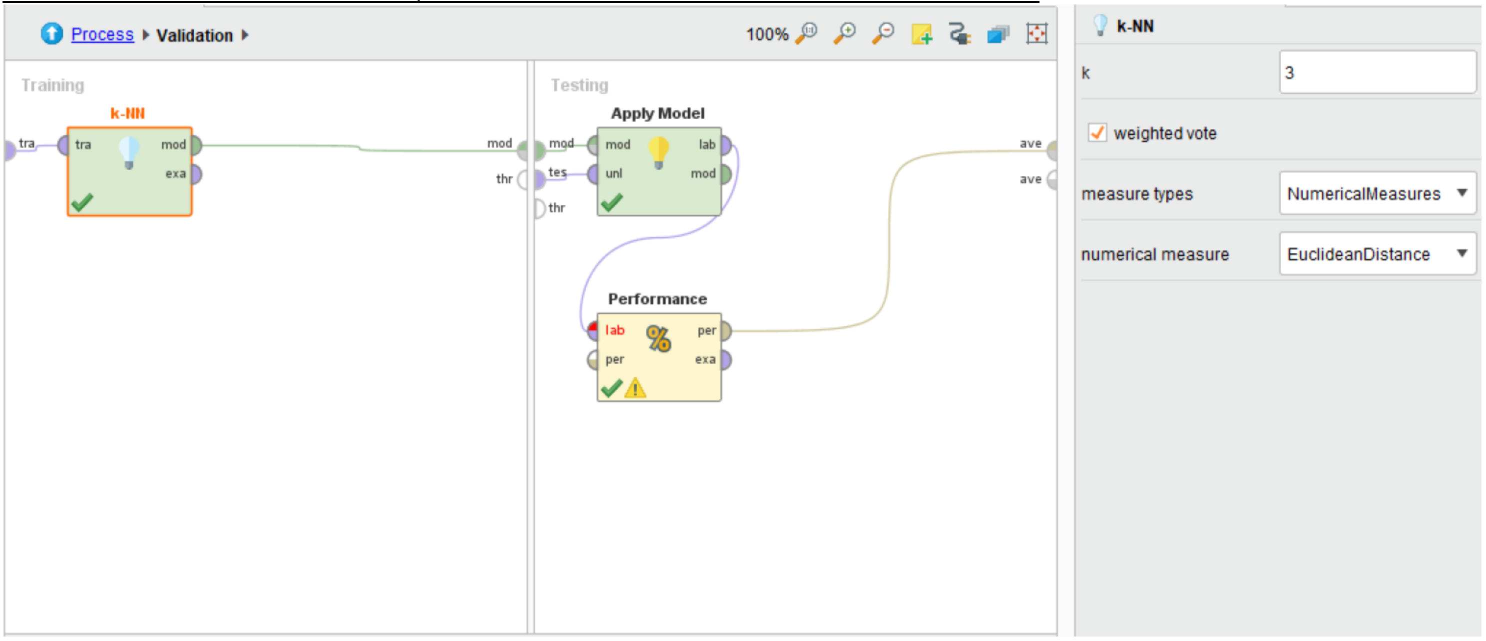
Process of my k-NN Rapidminer Model, notice the 0.7 split ratio and stratified sampling type.

- We also included the normalization module to normalize our data to make sure all our variables are similar in scale so they can be compared and can contribute equally to the distance computations.



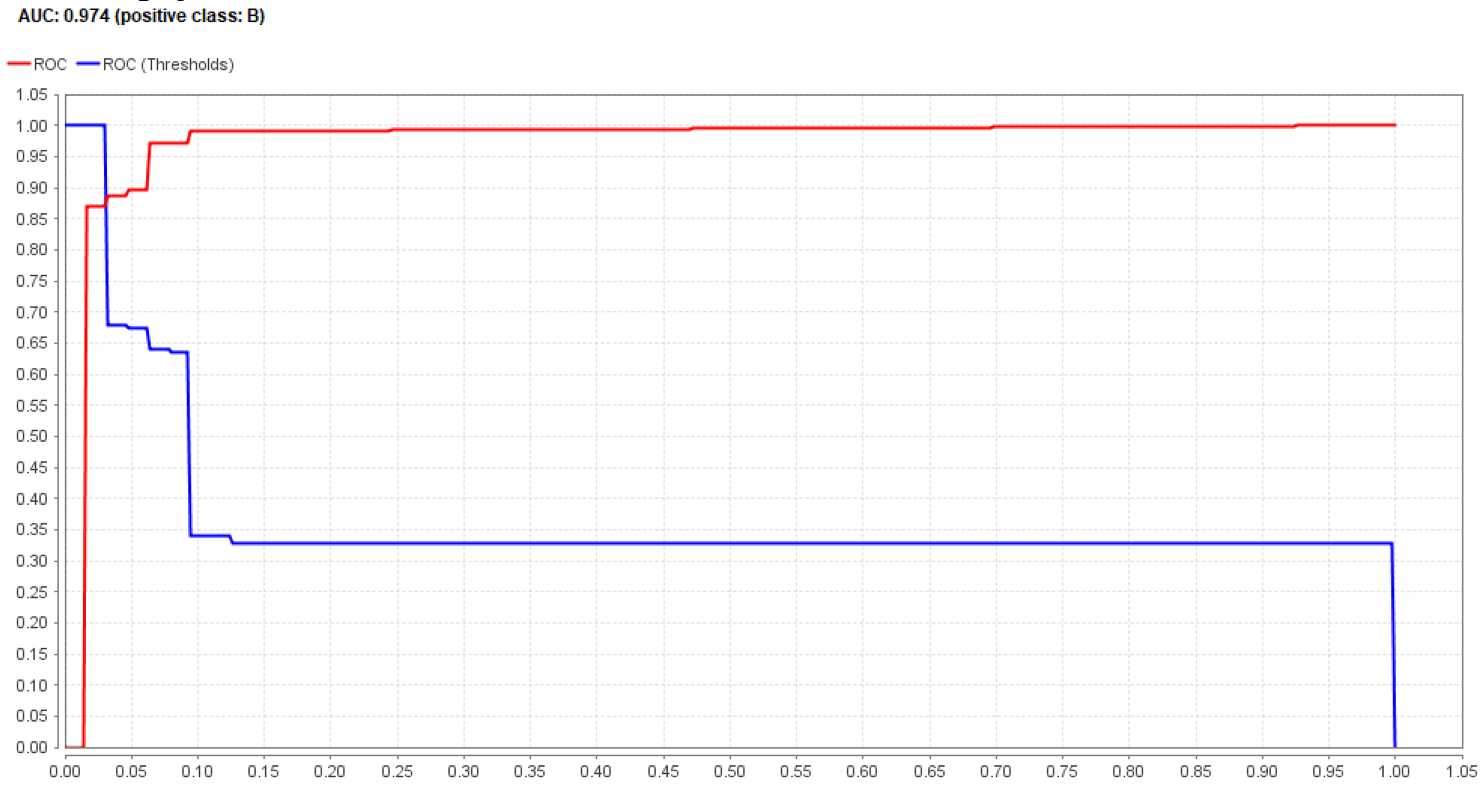
split	relative
split ratio	0.7
sampling type	stratified sampling

Details of the validation module, notice the k=3 and Euclidean Distance measure



k	3
✓ weighted vote	
measure types	NumericalMeasures
numerical measure	EuclideanDistance

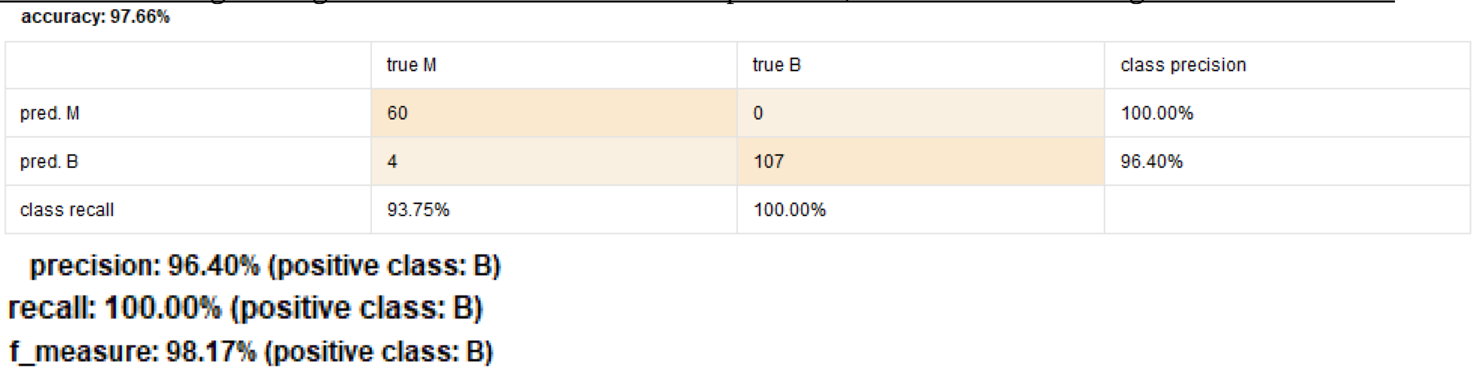
Our AUC graph with the ROC curve



Rapidminer Logistic Regression Model

Our Logistic Regression model for RapidMiner follows again a similar process. We imitated our in-class example then made sure key values like split ratio, sampling type and regularization are correct. Comparing our results, we see that our Logistic Regression model marginally outperforms our k-NN model, with accuracy at 97.66%. Looking at the table, we can find our precision and recall for malignant cases at 100% and 93.75% respectively. Once again, both metrics outperform our k-NN model, with higher being better as explained in the table at the top. We once again calculate the f-measure, which is $2 * \frac{Precision*Recall}{Precision+Recall} = 2 * \frac{1.0*0.9375}{1.0+0.9375} = 0.967742$ or 96.77%, which once again outperforms our k-NN model. Across the board, our logistic regression model is better. Lastly, we can look at our ROC curve and its max value of 0.996, which again outperforms the k-NN model.

Results of our logistic regression model. Notice how the precision, recall and f-measure given are for class b.

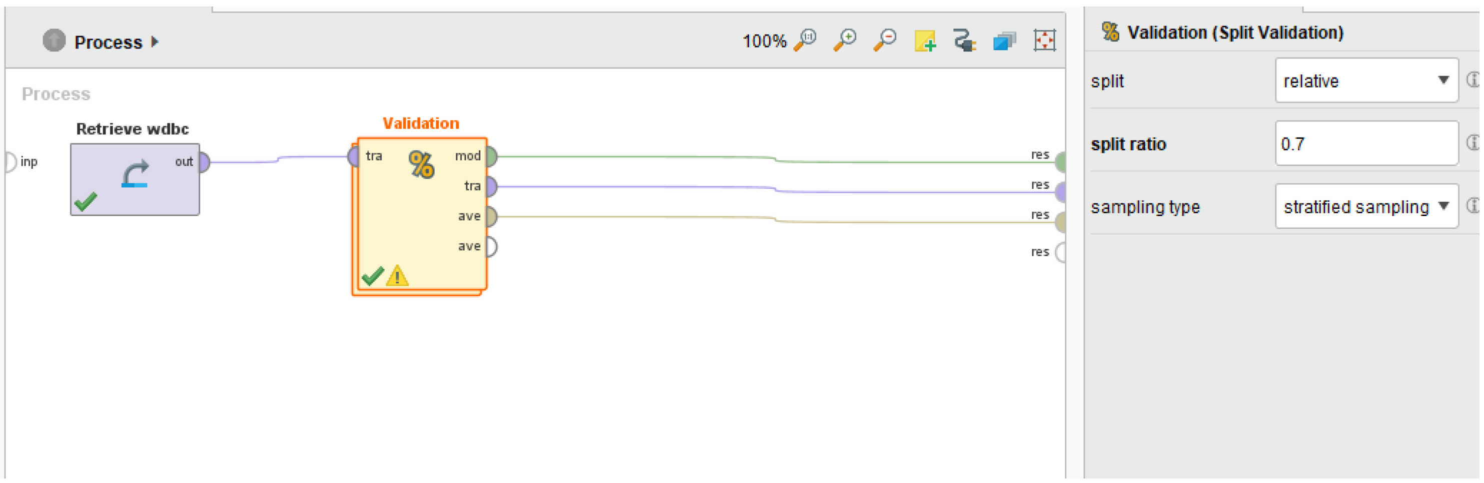


accuracy: 97.66%

	true M	true B	class precision
pred. M	60	0	100.00%
pred. B	4	107	96.40%
class recall	93.75%	100.00%	

precision: 96.40% (positive class: B)
recall: 100.00% (positive class: B)
f_measure: 98.17% (positive class: B)

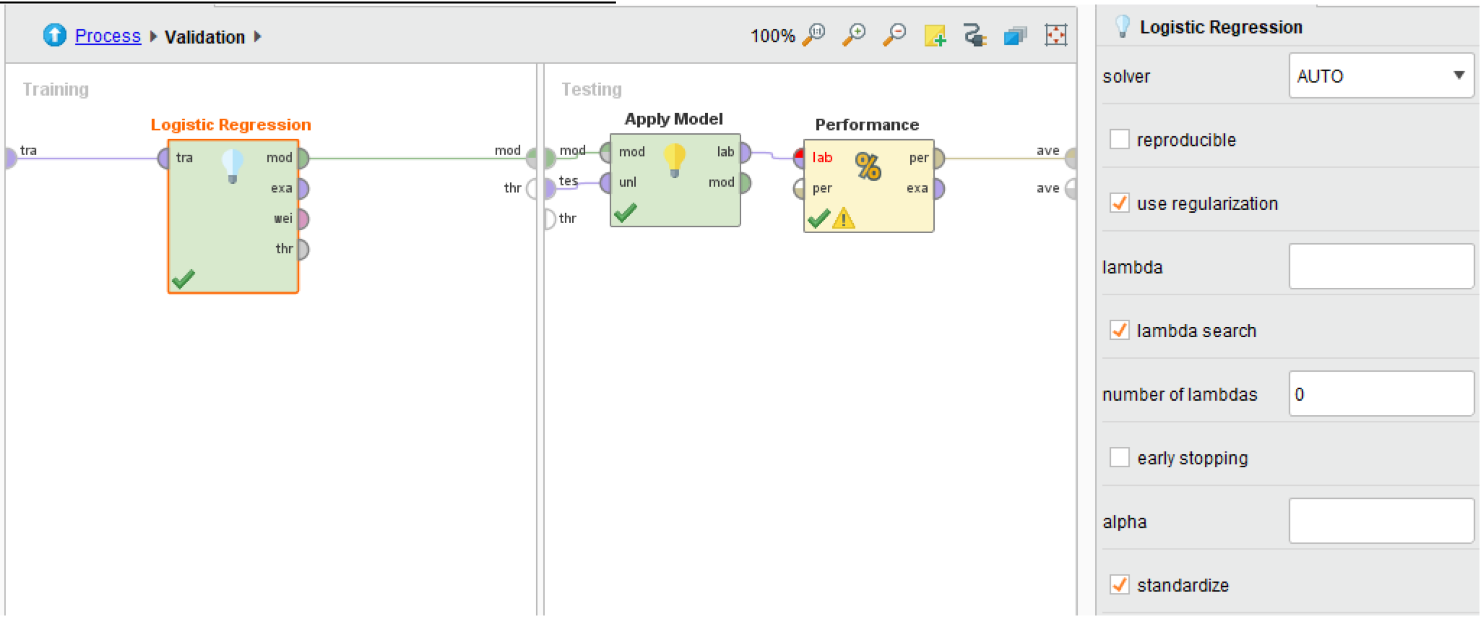
Process of my logistic regression Rapidminer Model. Notice the 0.7 split ratio and stratified sampling type.



Validation (Split Validation)

split	relative
split ratio	0.7
sampling type	stratified sampling

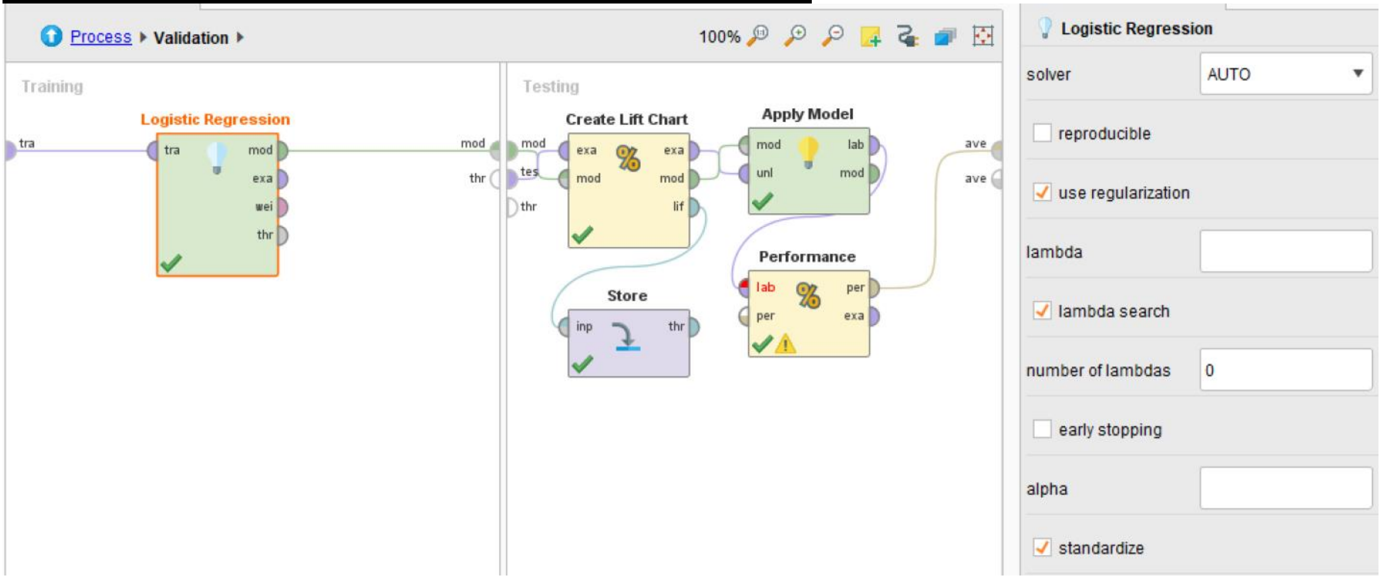
Details of the validation module. notice how we used regularization and used lambda search and made sure to standardize. Regularization and lambda search are used to reduce overfitting, while standardization ensures that our data columns have 0 mean and unit distance



Logistic Regression

solver	AUTO
reproducible	☐
use regularization	✓
lambda	
lambda search	✓
number of lambdas	0
early stopping	☐
alpha	
standardize	✓

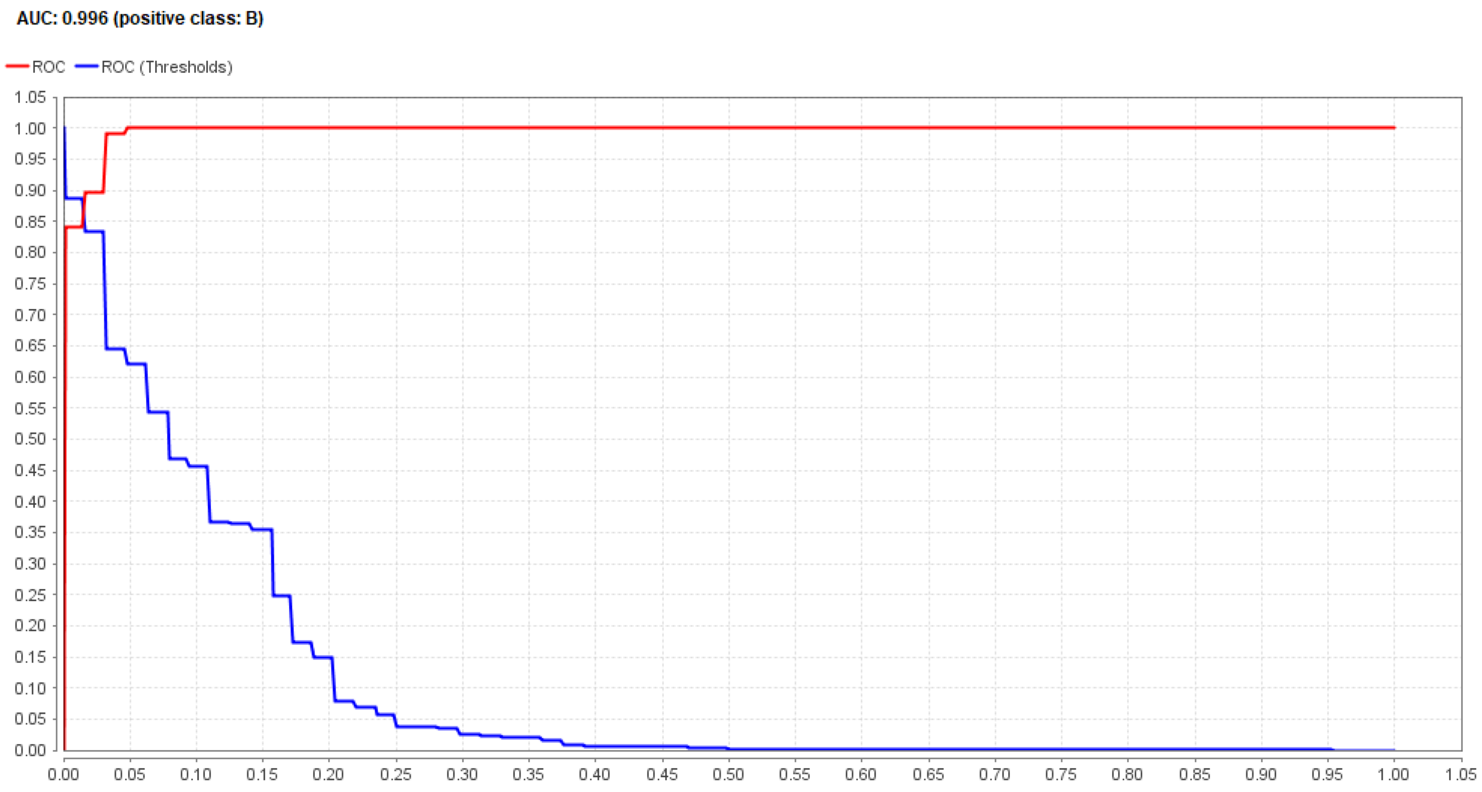
***I was originally going to use the most advanced version of Logistic Regression in class example available, but I do not understand the use of the Create Lift Chart and Store modules, so I revered back to the most simple version of Logistic Regression to keep things simple.**



Logistic Regression

solver	AUTO
reproducible	☐
use regularization	✓
lambda	
lambda search	✓
number of lambdas	0
early stopping	☐
alpha	
standardize	✓

Our AUC graph with the ROC curve



Appendix (Data Description)

1.Title: Wisconsin Diagnostic Breast Cancer (WDBC)
Results:
- predicting field 2, diagnosis: B = benign, M = malignant

2. Number of instances: 569
3. Number of attributes: 32 (ID, diagnosis, 30 real-valued input features)
4. Attribute information

1) ID number
2) Diagnosis (M = malignant, B = benign)
3-32)

Ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter)
b) texture (standard deviation of gray-scale values)
c) perimeter
d) area
e) smoothness (local variation in radius lengths)
f) compactness (perimeter^2 / area - 1.0)
g) concavity (severity of concave portions of the contour)
h) concave points (number of concave portions of the contour)
i) symmetry
j) fractal dimension ("coastline approximation" - 1)

The mean, standard error, and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 3 is Mean Radius, field 13 is Radius SE, field 23 is Worst Radius.
All feature values are recoded with four significant digits.

5. Missing attribute values: none
6. Class distribution: 357 benign, 212 malignant