# Tumor Prediction using ML

A comparison test for generic classifiers

By Siddhant Mahapatra – n9840371 and Alec Gurman – n9160531

## Objective

The aim of this project is to build some classifiers and evaluate their performance. The classification task is to predict whether a tumour is malignant (M) or benign (B). Using the sklearn and numpy libraries, we have built:
• a naive Bayes classifier
• a nearest neighbours classifier
• a decision tree classifier
• a support vector machine classifier

## Technical Specifications

### Datasets :

The records are stored in a text file named "medical_records.data". Each row corresponds to a patient record. The diagnosis is the attribute predicted. In this dataset, the diagnosis is the second field and is either B (benign) or M (malignant). There are 32 attributes in total (ID, diagnosis, 30 real-valued input features). The first two attributes are the ID number and the Diagnosis (M = malignant, B = benign). The rest from 3 to 32 constitute of ten real-valued features

## Code Analysis:

**Libraries and used :** *Pandas , sklearn , numpy*

**Preprocessing:** Firstly, the function prepare_dataset loads the data records from the text file, and converts the information to numpy arrays using Pandas's *read_csv* and *as_matrix* functions respectively. The second column of the matrix, the class labels that depicts the Diagnosis (M = malignant, B = benign), is stored separately using Pandas's *as_matrix* function. The *as_matrix* function converts the frame to its Numpy-array representation. Using a combination of *concatenate* and *normalize* functions, columns 2,3,4,5,13,14,15,22,23,24,25 of the original matrix are normalized as SVM does not yield favourable results with values larger than 1. The scikit-learn's *normalize* function scales input vectors individually to unit norm (vector length). Using numpy's *unique* function, class labels B,M are converted to 0,1, respectively.
Data from 387 to 569 are to be treated as the validation set, especially just for the purpose of cross validation where applicable, hence in this case we'll call it the *cross-validation set*. Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a **test set** X_test, y_test. Using scikit-learn, the 1 to 387 of the original array is quickly split into training and test subsets *train_test_split* function, while holding out 40% of the data for testing (evaluating) our classifier.

### Why do I need Cross Validation, particularly for k-NN and SVM classifiers?

Another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.
However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

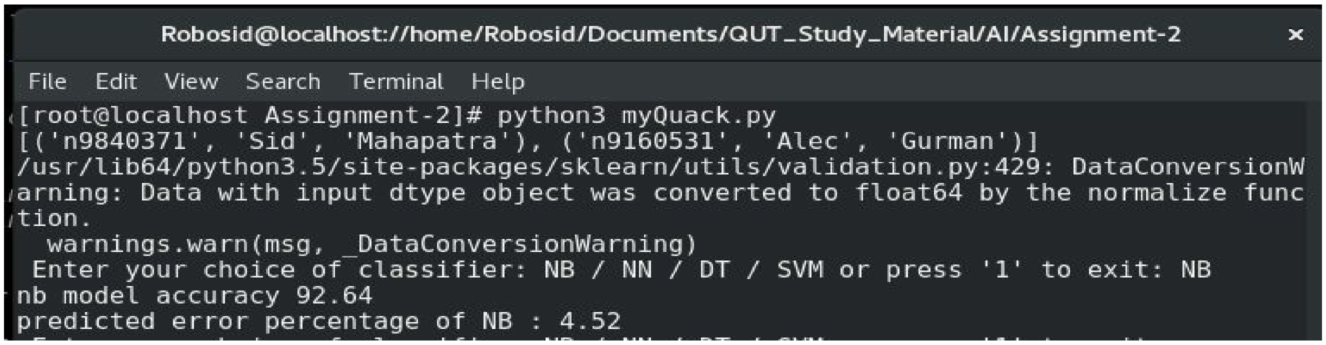
A solution to this problem is a procedure called ***cross-validation*** (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called *k*-fold CV, the **cross-validation set** is split into *k* smaller sets. The following procedure is followed for each of the *k* “folds”:

- A model is trained using $k-1$ of the folds as training data;
- the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The performance measure reported by *k*-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as it is the case when fixing an arbitrary test set), which is a major advantage in problem such as inverse inference where the number of samples is  small to moderate.

**Naive Bayes Classifier:**   NB Classifier is built using the *GaussianNB* of scikit-learn , which is a supervised learning method based on applying Bayes’ theorem with strong (naive) feature independence assumptions, and the likelihood of the features is assumed to be Gaussian. It’s *fit*  function fits Gaussian Naive Bayes according to X, y of the training subset. Using the testdata, Gaussian’s *predict*  function is used to perform classification on an array of test vectors X. Using the training data and training label, the *score* function of Gaussian returns the mean accuracy on the given test data and labels. This accuracy is nothing but the model accuracy of the Naive Bayes Classifier.

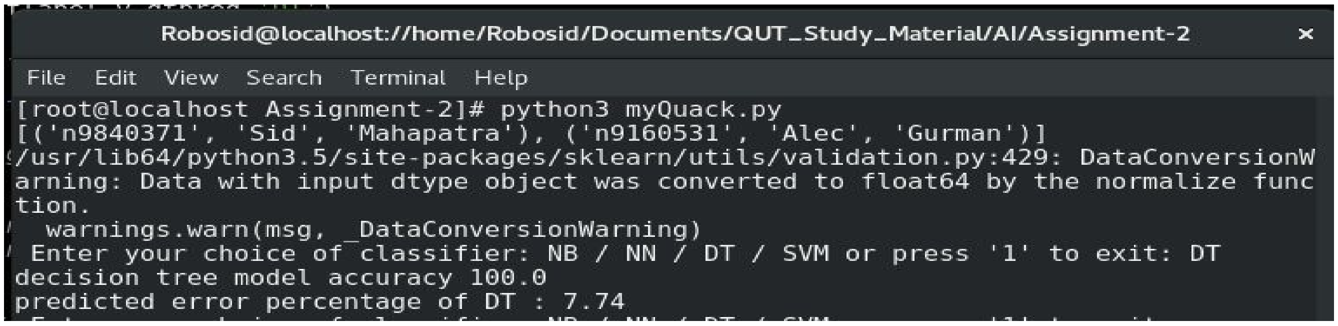
In the *predict_accuracy* function, the prediction error is calculated by pitching the testlabel against the predicted result (the result of the operation of *predict* function on the testdata).



```
[root@localhost Assignment-2]# python3 myQuack.py
[('n9840371', 'Sid', 'Mahapatra'), ('n9160531', 'Alec', 'Gurman')]
/usr/lib64/python3.5/site-packages/sklearn/utils/validation.py:429: DataConversionWarning: Data with input dtype object was converted to float64 by the normalize function.
  warnings.warn(msg, _DataConversionWarning)
 Enter your choice of classifier: NB / NN / DT / SVM or press '1' to exit: NB
nb model accuracy 92.64
predicted error percentage of NB : 4.52
```

**Decision Tree Classifier:** **Decision Trees (DTs)** are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

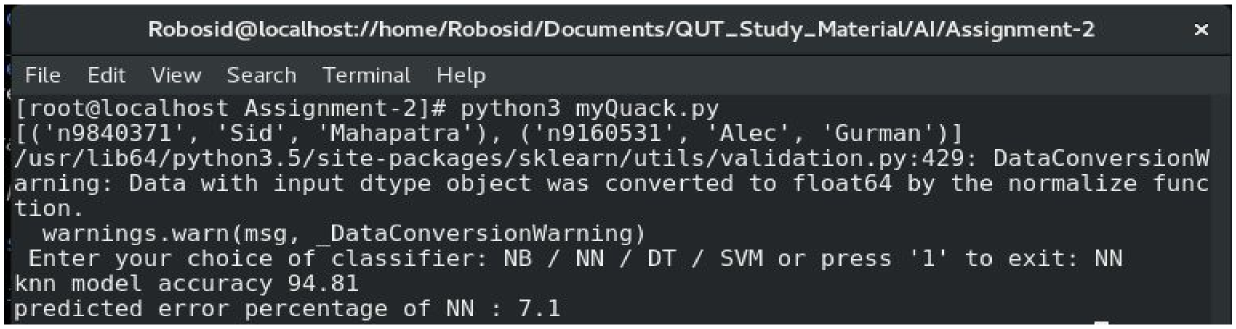
DT Classifier is built using the *DecisionTreeClassifier* of scikit-learn,which is a class capable of performing multi-class classification on a dataset. It takes as input two arrays: an array trainingdata, sparse or dense, holding the training samples, and an array traininglabel of integer values, holding the class labels for the training samples. After being fitted using its *fit* function on the trainingdata and traininglabel, the model can then be used to predict the class of samples using its *predict* function on the testdata.



```
[root@localhost Assignment-2]# python3 myQuack.py
[('n9840371', 'Sid', 'Mahapatra'), ('n9160531', 'Alec', 'Gurman')]
/usr/lib64/python3.5/site-packages/sklearn/utils/validation.py:429: DataConversionWarning: Data with input dtype object was converted to float64 by the normalize function.
  warnings.warn(msg, _DataConversionWarning)
 Enter your choice of classifier: NB / NN / DT / SVM or press '1' to exit: DT
decision tree model accuracy 100.0
predicted error percentage of DT : 7.74
```

Using the training data and training label, the *score* function of *DecisionTreeClassifier* returns the mean accuracy on the given test data and labels. This accuracy is nothing but the model accuracy of the Decision Tree Classifier. In the *predict_accuracy* function, the prediction error is calculated by pitching the testlabel against the predicted result (the result of the operation of *predict* function on the testdata).

**K Nearest Neighbours Classifier:** The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbors-based methods are known as *non-generalizing* machine learning methods, since they simply "remember" all of its training data (possibly transformed into a fast indexing structure such as a Ball Tree or KD Tree.).
kNN Classifier is built using the *KNeighborsClassifier* of scikit-learn, which is a Classifier implementing the k-nearest neighbors vote. In this case, it takes the k value and the algorithm type as parameters. In our case, BallTree is used, as they perform well for fast generalized N-point problems. *Fit* function fits the model using trainingdata and training label.
KNeighborsClassifier implements learning based on the k nearest neighbors of each query point, where k is an integer value specified or calculated. In our case, we're using cross validation to find the best k value. The simplest way to use cross-validation is to call the cross_val_score helper function on the estimator and the dataset (cross-validation subset). It evaluates a score by cross-validation using the *cross_val_score* function of sklearn. By default, the score computed at each CV iteration (10) is the *score* method of the estimator. We changed this by using the scoring parameter to 'accuracy', which is connected to the function *metrics.accuracy_score*. We run this algorithm for every odd value of 'k' as stats show that best value of 'k' has mostly been an odd number, and find the k value which produces the maximum accuracy.
We then use this value of 'k', in the *KNeighborsClassifier* function with 'ball_tree' algorithm, fit it with trainingdata and traininglabel, and deploy the predict function on testdata. Using the training data and training label, the *score* function of *KNeighborsClassifier* returns the mean accuracy on the given test data and labels. This accuracy is nothing but the model accuracy of the k-NN Classifier. In the *predict_accuracy* function, the prediction error is calculated by pitching the testlabel against the predicted result (the result of the operation of *predict* function on the testdata).

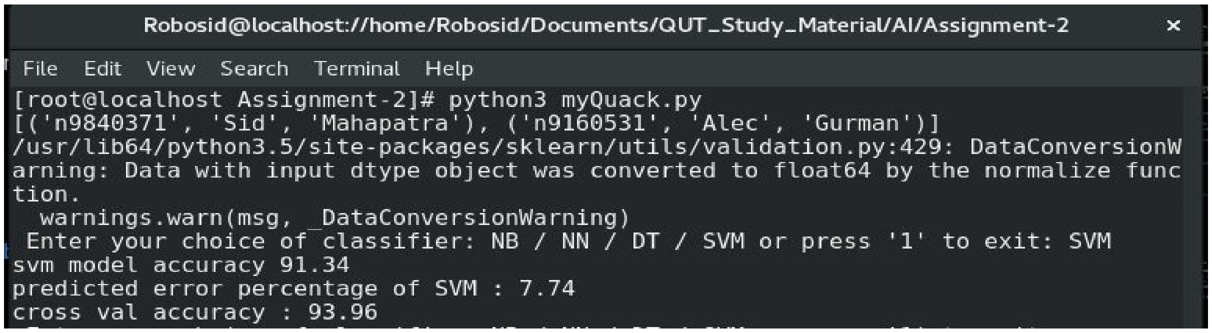

Robosid@localhost://home/Robosid/Documents/QUT_Study_Material/AI/Assignment-2
File Edit View Search Terminal Help

```
[root@localhost Assignment-2]# python3 myQuack.py
[('n9840371', 'Sid', 'Mahapatra'), ('n9160531', 'Alec', 'Gurman')]
/usr/lib64/python3.5/site-packages/sklearn/utils/validation.py:429: DataConversionW
arning: Data with input dtype object was converted to float64 by the normalize func
tion.
  warnings.warn(msg, _DataConversionWarning)
 Enter your choice of classifier: NB / NN / DT / SVM or press '1' to exit: NN
knn model accuracy 94.81
predicted error percentage of NN : 7.1
```

**Support Vector Machines Classifier:** Support vector machines (SVMs) are a set of supervised learning methods used for classification. We used the *LinearSVC* function of scikit-learn to implement SVM. It is implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples. This supports both dense and sparse input.
After being fitted using its *fit* function on the trainingdata and traininglabel, the model can then be used to predict the class of samples using its *predict* function on the testdata.
Using the training data and training label, the *score* function of *LinearSVC* returns the mean accuracy on the given test data and labels. This accuracy is nothing but the model accuracy of the SVM Classifier. In the *predict_accuracy* function, the prediction

error is calculated by pitching the testlabel against the predicted result (the result of the operation of *predict* function on the testdata).

SVM is also implemented via cross validation just to check the influence on its accuracy. We used the *SVC* function of scikit-learn to implement this with a linear kernel and the Penalty parameter C of the error term as 1. After fitting it with trainingdata and traininglabel, cross validation is performed using sklearn's *cross_val_score* function on the cross-validation subset and CV iteration of 5. After getting the score, the accuracy percentage of SVM using cross validation is printed.


Robosid@localhost://home/Robosid/Documents/QUT_Study_Material/AI/Assignment-2
File Edit View Search Terminal Help

```
[root@localhost Assignment-2]# python3 myQuack.py
[('n9840371', 'Sid', 'Mahapatra'), ('n9160531', 'Alec', 'Gurman')]
/usr/lib64/python3.5/site-packages/sklearn/utils/validation.py:429: DataConversionW
arning: Data with input dtype object was converted to float64 by the normalize func
tion.
  warnings.warn(msg, _DataConversionWarning)
 Enter your choice of classifier: NB / NN / DT / SVM or press '1' to exit: SVM
svm model accuracy 91.34
predicted error percentage of SVM : 7.74
cross val accuracy : 93.96
```

## Comparing and choosing a Machine Learning Classifier

If your training set is small, high bias/low variance classifiers (e.g., Naive Bayes) have an advantage over low bias/high variance classifiers (e.g., kNN), since the latter will overfit. But low bias/high variance classifiers start to win out as your training set grows (they have lower asymptotic error), since high bias classifiers aren't powerful enough to provide accurate models.

For classification, **Naive Bayes** is a good starter, as it has good performances, is highly scalable and can adapt to almost any kind of classification task, the only issue is the computation cost (quadratic because we need to compute the distance matrix, so it may not be a good fit for high dimensional data). Super simple, and just doing a bunch of counts. If the NB conditional independence assumption actually holds, a Naive Bayes classifier will converge quicker than discriminative models, so we need less training data. And even if the NB assumption doesn't hold, a NB classifier still often does a great job in practice. Its main disadvantage is that it can't learn interactions between features.

**Decision Trees:** They easily handle feature interactions and they're non-parametric, so I don't have to worry about outliers or whether the data is linearly separable. Disadvantage is that they easily overfit, but that's where ensemble methods like random forests (or boosted trees) come in, although not used here in this code.

**K-nearest neighbors** - **simplest** thing you can do, **often effective** but **slow** and requires **lots of memory**. Works well with cross validation and the 'ball tree' algorithm. The accuracy is not too compromising.

**SVM** - **Among the best** with **limited data**, but **losing against boosting** or **random trees** only when **large data sets** are available, although not proved in this code. High accuracy, nice theoretical guarantees regarding overfitting, and with an appropriate kernel they can work well even if our data isn't linearly separable in the base feature space. Should work where very high-dimensional spaces are the norm. Memory-intensive, hard to interpret, and kind of annoying to run and tune. Works even better with cross validation.

So, to sum it up, the **performance order** of the classifiers wrt **prediction error or test set**, used in this project is as follows: ***Naive Bayes > Support Vector Machine (with Cross Validation) > k-NN > Decision Trees > Support Vector Machine***