WHITE PAPER

cool clinical

Consortium for Clinical Science and AI

A Comparison of Machine Learning Algorithms for Breast Cancer Classification: Part 2

August 2020

Felix Beacher, PhD

About Cool Clinical

Cool Clinical is a non-profit consortium of clinical and computational scientists. Cool Clinical publishes original articles and reports for businesses, the public sector, NGOs and the general public. Our goal is to advance the conversation on AI applications to clinical science.

To learn more about the research of Cool Clinical, visit www.coolclinical.com

Contents

Abstract

Machine learning (ML) can address various problems related to medical diagnosis and is central to the development of personalized medicine. Breast cancer is one of the most common causes of death among women in the United States. The aim of this paper is to compare the results of various machine learning algorithms used in Part 1 on a comparable dataset of histological features of breast tumour samples, the Wisconsin Breast Cancer Dataset.

Background

This paper uses the second of two datasets published by Dr. William H. Wolberg, Madison, Wisconsin, the Wisconsin Breast Cancer Dataset. This dataset contains measurements of cytological features of samples taken from fine needle aspirate (Mangasarian and Wolberg, 1990). These cells features were produced by a program called Xcyt on the basis of digital images of those cells.

This dataset contains samples from 699 patients. There are nine features, each of which is assigned a score of 1 to 10, where 1 is most benign and 10 is most malignant:

1. Clump Thickness
   This is a measure of whether cells are mono- or multi-layered. Benign cells tend to be grouped in mono-layers, while malignant cells are often grouped in multi-layers.
2. Uniformity of Cell Size
   Malignant cells tend to have greater variations in size than benign cells.
3. Uniformity of Cell Shape
   Malignant cells tend to have greater variations in shape than benign cells.
4. Marginal Adhesion
   This is a measure of how much cells outside the epithelium stick together (cell adhesion). Normal cells stick together more than cancer cells.
5. Single Epithelial Cell Size
   Epithelia are cell 'linings' that separate different parts of the body. Enlarged epithelial cells may indicate malignancy.
6. Bare Nuclei
   - Bare nuclei are nuclei not surrounded by cytoplasm. Bare nuclei may correspond to cell degeneration, including cancer.
7. Bland Chromatin
   Chromatin is a complex of DNA and protein which packages long DNA molecules into compact structures. 'Bland Chromatin' is a measure of nucleic uniformity. 'Bland' nuclei are typical of benign cells, coarse nuclei tend to be coarse.
8. Normal Nucleoli
   Nucleoli are small structures in the nucleus which create ribosomes (RNA particles). In cancer cells nucleoli may be enlarged, or more numerous.
9. Mitoses
   Mitosis is cell division and replication. The number of mitoses indicates the grade of a cancer, i.e. how fast-growing it is.

(See http://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+%28diagnostic%29)

The dataset is unbalanced in its classes, with 458 samples being benign and only 241 samples being malignant. This means that figures for accuracy can be misleading, as apparently reasonable accuracy could be achieved simply by assigning all unknown labels to the benign class. To address this, full confusion matrices are presented and specficity and sensitivity values are given.

As in Part 1, the algorithms used here were:

1. Logistic Regression
2. K-Nearest Neighbor
3. Multilayer perceptron ('vanilla' neural network)
4. Support Vector Machine
5. Naïve Bayes
6. Decision Tree
7. Random Forest
8. Gradient Boosting
9. XGBoost
10. Adaboost

The Aim of this Paper

The aim of this paper is to make a second comparison of the implementations of a selection of ML algorithms for breast cancer, and to compare results to those obtained in Part 1.

Results

Results are summarized in Table 2, with best performing algorithms for each metric highlighted in yellow.

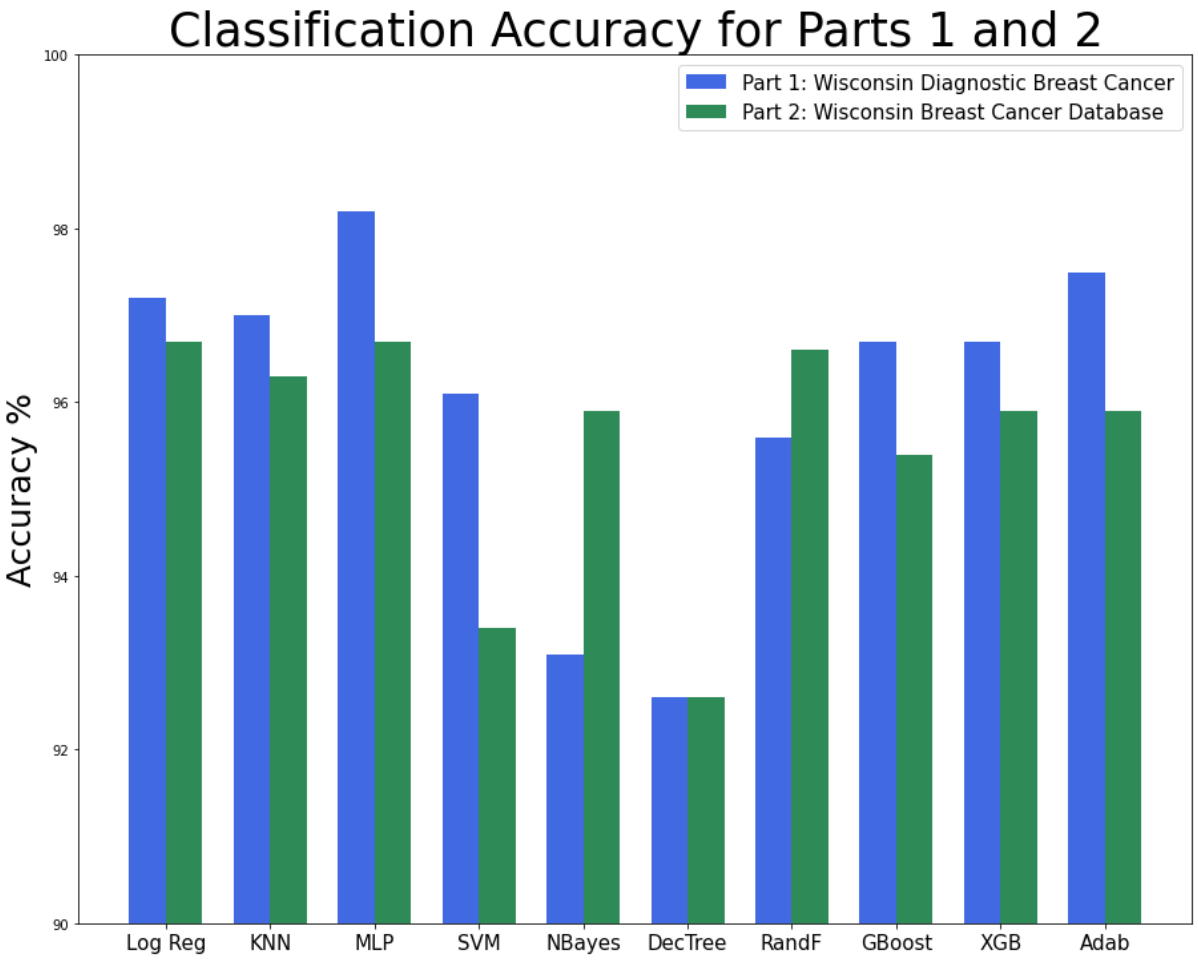
Two algorithms had an accuracy of 97%, which is slightly higher than the highest of 95.90% reported by the UCI Machine Learning team. All algorithms were associated with very high levels on the performance metrics, particularly logistic regression. This indicates that the data do not present a difficult classification problem for which the more advanced ML algorithms have an advantage.

Table 1: Key results for the Wisconsin Breast Cancer Database

	Accuracy	Sensitivity /Recall	Precision	F1 Score	AUC
Logistic Regression	96.7%	95.9%	95.2%	97.1%	95.7%
K-Nearest Neighbor	96.3%	95.9%	94.5%	97.2%	95.2%
MLP	96.7%	95.1%	95.3%	96.6%	94.5%
SVM	93.4%	91.8%	91.5%	93.8%	84.5%
Naïve Bayes	95.9%	94.5%	94.2%	96.1%	91.6%
Decision Tree	92.6%	95.7%	88.5%	97.1%	92.1%
Random Forest	96.6%	96.7%	95.1%	97.7%	94.1%
Gradient Boosting	95.4%	100%	93.2%	100%	94.3%
XGBoost	95.9%	96.6%	94%	97.6%	94.8%
Adaboost	95.9%	96.2%	93.8%	97.4%	95.1%

Accuracy: the proportion of all correctly predicted cases to the total number of cases
Sensitivity /Recall: the proportion of true positives
Precision: is the proportion of correctly predicted positives /tot predicted positives
F1 Score: is the weighted average of Sensitivity and Precision
AUC: is the ROC area under the curve

The key results on accuracy are presented alongside results from the Part 1 paper.



Overall Conclusions

The dataset is fairly small (n=699) relative to most datasets used in ML. Nonetheless, the models presented here are associated with accuracy of between 94% and 97%. Overall, the results are broadly consistent with those for Part 1, even for different types of histological measurement. The results further underline the potential of ML to support the diagnosis of breast cancer.

References

1. Mangasarian and Wolberg, 1990, "Cancer diagnosis via linear programming", SIAM News, Volume 23, Number 5, pp 1 & 18.

Appendix: Programming Steps for Developing a Diagnostic for Breast Cancer

Data Exploration

#imports

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

#define data

```
dataset = pd.read_csv('C:/Users/felix/OneDrive/Documents/01 computing/1 ML/04 projects/1 cancer/2 breast/1 wisconsin features/data/2 Wisconsin Breast Cancer Database/Wisconsin Breast Cancer Database.data.csv',header=None)
```

Inspect the dataset

```
dataset.head()
```

```
      0  1  2  3  4  5  6  7  8  9  10
0  1000025  5  1  1  1  2  1  3  1  1  2
1  1002945  5  4  4  5  7 10  3  2  1  2
2  1015425  3  1  1  1  2  2  3  1  1  2
3  1016277  6  8  8  1  3  4  3  7  1  2
4  1017023  4  1  1  3  2  1  3  1  1  2
```

Give the dataset column labels

```
dataset.columns=['ID', 'ClumpThickness', 'Unif_Cell_Size', 'Unif_Cell_Shape', 'Marg_Adhesion', 'Single_Epith_Cell_Size', 'Bare_Nuclei', 'Bland_Chromatin', 'Normal_Nucleoli', 'Mitoses', 'label']
```

find the dimensions of dataset

```
dataset.shape
        (699, 11)
```

I.e. 699 rows and 11 columns.

Get frequencies of each tumour type

```
(unique, counts) = np.unique(dataset['label'], return_counts=True)

counts
        [458, 241]
```

I.e. 458 2s (benign) and 241 4s (malignant)

Check for Missing Data

```
dataset.isna().sum()
```

```
0    0
1    0
2    0
3    0
4    0
5    0
6    0
7    0
8    0
9    0
10   0
```

There are no NaNs or missing cells. However, visual inspection shows that there are '?'s in the dataset.

```
(dataset == '?').astype(int).sum(axis=0)
```

```
ID                          0
ClumpThickness              0
Unif_Cell_Size              0
Unif_Cell_Shape             0
Marg_Adhesion               0
Single_Epith_Cell_Size      0
Bare_Nuclei                16
Bland_Chromatin             0
Normal_Nucleoli             0
Mitoses                     0
label                       0
dtype: int64
```

Let's check the distribution of values in 'Bare_Nuclei'

```
dataset['Bare_Nuclei'].value_counts()
```

```
1     402
10    132
5      30
2      30
3      28
8      21
4      19
?      16
9       9
7       8
6       4
```

- We can see there is a somewhat weird bimodal distribution with 1 and 10 being much more frequent than the other values.
- This suggests mode is a better imputation method than mean.
- The easiest way to do this is by finding and replacing in the csv file and reloading

Count number of zeros in each column

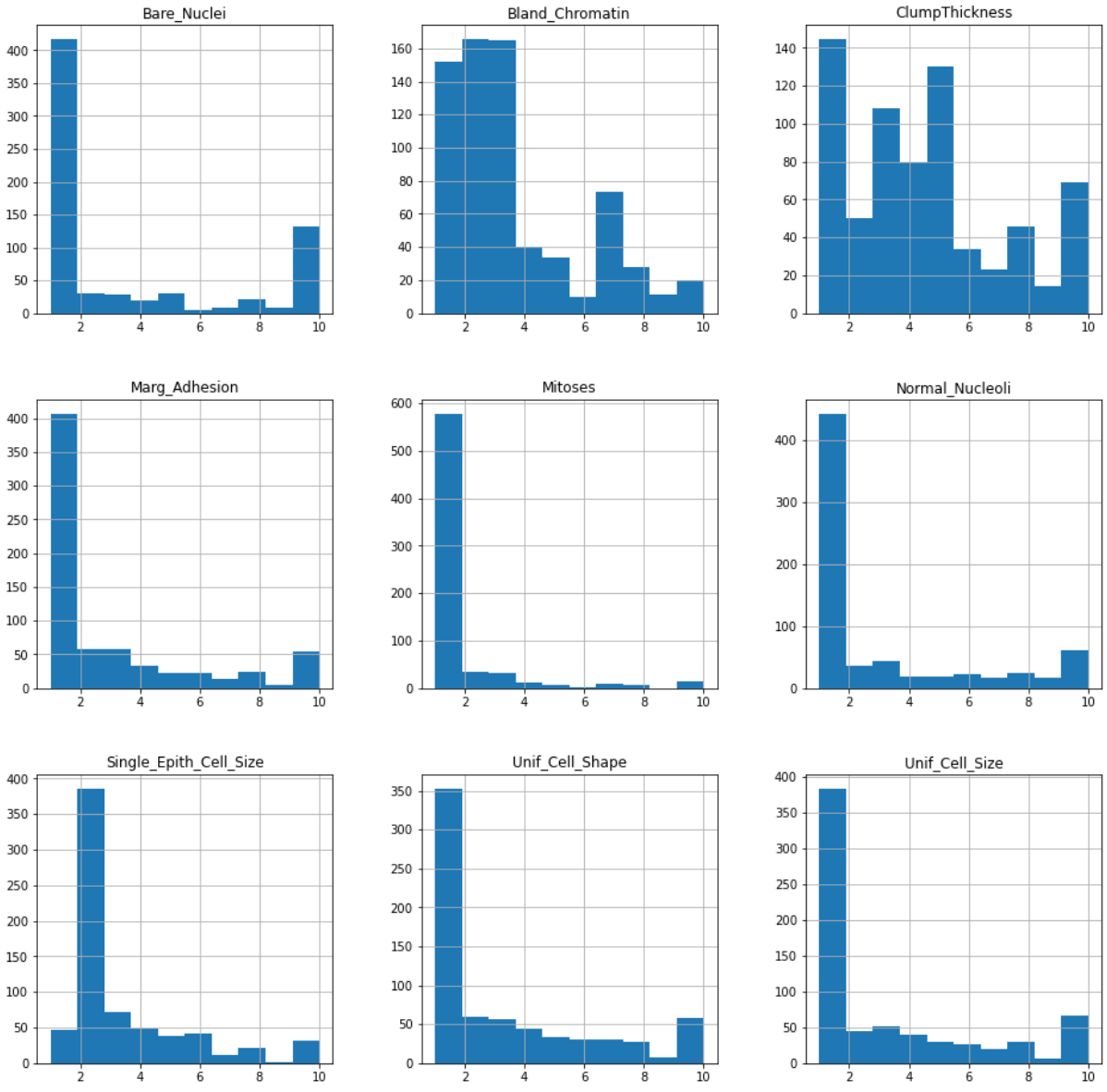
```
(dataset == 0).astype(int).sum(axis=0)
```

```
0    0
1    0
2    0
3    0
4    0
5    0
6    0
7    0
8    0
9    0
10   0
```

We see there are no zero values – we do not have to consider this as a problem.

Plot histogram of distributions of all variables

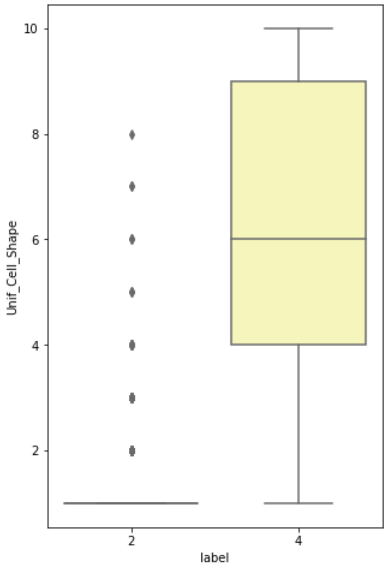
```
relev_data = dataset.loc[:,'ClumpThickness':'Mitoses']
hist = relev_data.hist(bins=10, figsize=(16,16))
```

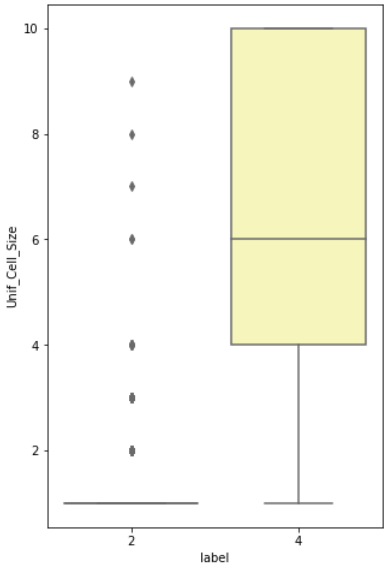


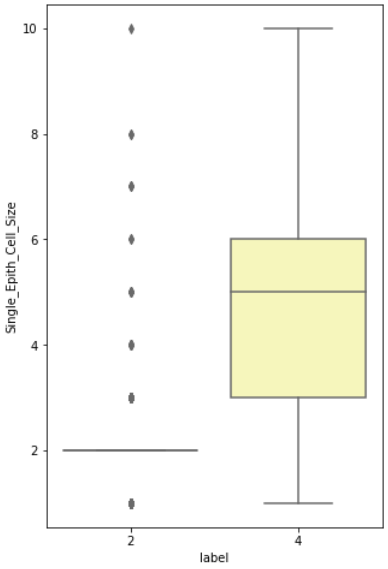
We can see that all but two of the variables are highly skewed to the lower end. This may limit the discriminatory value of the dataset.

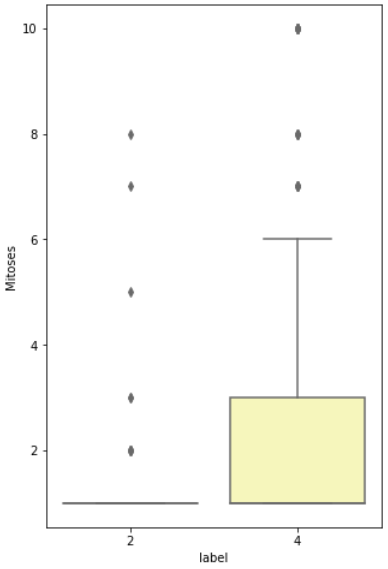
Plot boxplots of major variables by diagnosis

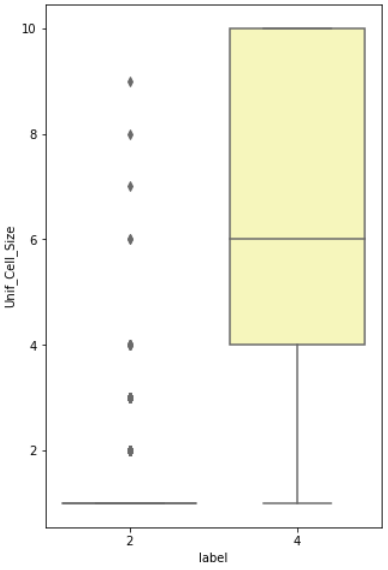
```
import seaborn as sns

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['ClumpThickness'], data= dataset, palette="Set3")

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['Unif_Cell_Size'], data= dataset, palette="Set3")

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['Unif_Cell_Shape'], data= dataset, palette="Set3")

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['Marg_Adhesion'], data= dataset, palette="Set3")

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['Single_Epith_Cell_Size'], data= dataset, palette="Set3")

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['Bare_Nuclei'], data= dataset, palette="Set3")

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['Bland_Chromatin'], data= dataset, palette="Set3")

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['Normal_Nucleoli'], data= dataset, palette="Set3")

plt.figure(figsize=(5,8))
ax = sns.boxplot(x=dataset['label'], y= dataset['Mitoses'], data= dataset, palette="Set3")
```

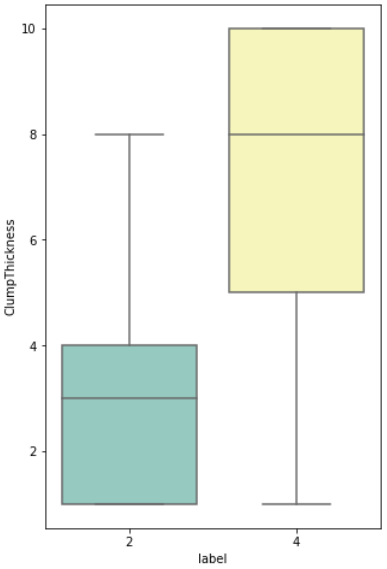


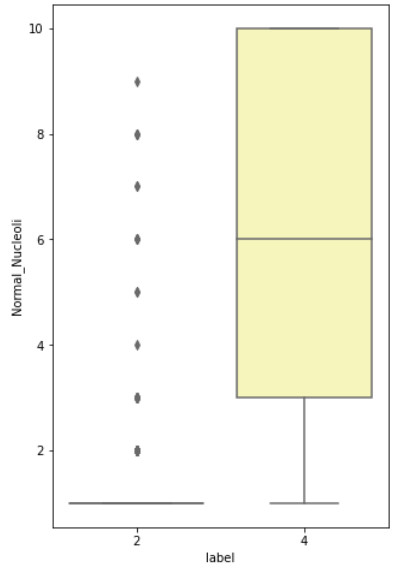


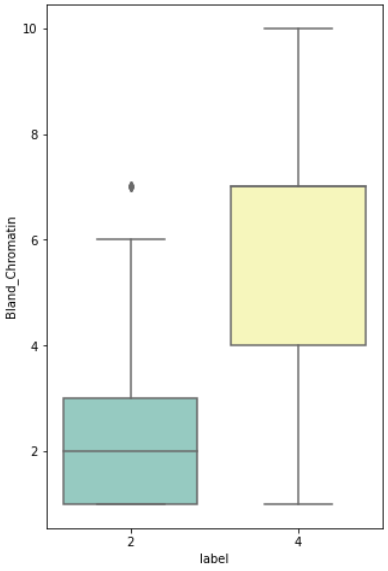


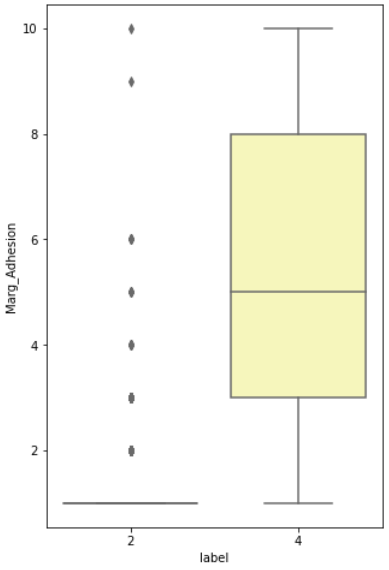


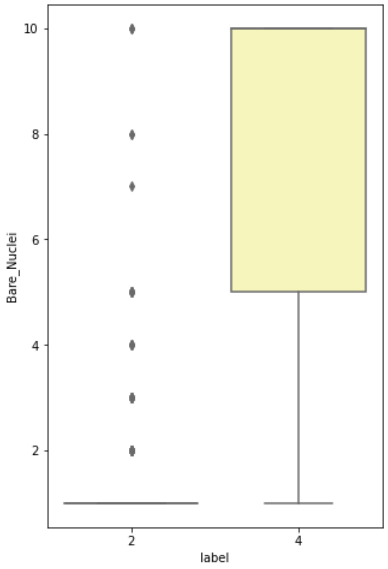












These plots give us an idea of how difficult the classification problem is and which variables are likely to be of most use in discriminating between classes.

We can see that this is not a difficult classification problem because many variables (such as uniform cell shape and uniform cell size) are very different between the two classes. Uniform cell shape and uniform cell size are usually small for benign cells and take a range of non-small values in the malignany class.

Normalise data

- Most ML algorithms use Euclidian distances between data points
- However, the variables are scaled in very different ways
- To avoid biases, we need to convert the variables into the same dimensions
- Here we use StandardScaler to yield a zero mean and variance of 1

```
X= dataset.loc[:,'ClumpThickness':'Mitoses']

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)

Y = dataset['label']
Y = Y.to_numpy()

Y = np.where(Y==2, 0, Y)
Y = np.where(Y==4, 1, Y)
```

Define and run Models

We will use various classification algorithms:

1. Logistic Regression
2. K-Nearest Neighbor
3. Multilayer perceptron ('vanilla' neural network)
4. Support Vector Machines
5. Naïve Bayes
6. Decision Tree
7. Random Forest
8. Gradient Boost
9. XGBoost
10. Adaboost

Imports for all models

```
# For hyperparameter tuning
    from sklearn.model_selection import GridSearchCV

# The sample is fairly small, so cross-validation is better than test-train-split
    from sklearn.model_selection import cross_validate
```

LogisticRegression

It is good practice, based on Occam's principle, to begin with a simple algorithm like regression and to see whether more complex ML procedures can improve on it.

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()

# get optimal hyperparameters
logreg_grid_values = {'penalty': ['l1', 'l2'],'C':[0.001,0.01,0.1,1,5,10,25]}
        # penalty specifies the norm used in the penalization.
        # C is the Inverse of regularization strength (smaller values mean stronger regularization)

# perform GridSearch, using recall as the metric to optimize
logreg_gridsearch = GridSearchCV(logreg, param_grid = logreg_grid_values, scoring = 'recall')
logreg_best_params = logreg_gridsearch.fit(X, Y)
best_penalty = logreg_best_params.best_estimator_.get_params()['penalty']
best_c = logreg_best_params.best_estimator_.get_params()['C']

# instantiate tuned model
logreg_tuned = LogisticRegression(penalty=best_penalty, C= best_c)

# fit model and get scores
logreg_scores = cross_validate(logreg_tuned, X, Y, cv=5, scoring=('accuracy',
'f1','precision','recall','roc_auc'), return_train_score=True)

logreg_mn_acc = np.mean(list(logreg_scores.values())[2])
logreg_mn_f1 = np.mean(list(logreg_scores.values())[3])
logreg_mn_precis = np.mean(list(logreg_scores.values())[4])
logreg_mn_recall = np.mean(list(logreg_scores.values())[5])
logreg_mn_auc = np.mean(list(logreg_scores.values())[6])
```

KNN

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()

# get optimal hyperparameters – use this lists format so we can specify a range
leaf_size = list(range(1,50,2))
n_neighbors = list(range(1,30,2))
p=[1,2]
knn_grid_values = dict(leaf_size=leaf_size, n_neighbors=n_neighbors, p=p)

knn_gridsearch = GridSearchCV(knn, knn_grid_values, cv=10,scoring = 'recall')
knn_best_params = knn_gridsearch.fit(X,Y)     # this takes time to run!
best_leaf_size = knn_best_params.best_estimator_.get_params()['leaf_size']
best_p = knn_best_params.best_estimator_.get_params()['p']
best_NN = knn_best_params.best_estimator_.get_params()['n_neighbors']

# tuned model
knn_tuned = KNeighborsClassifier(n_neighbors = best_NN, metric = 'minkowski', p = best_p,
leaf_size=best_leaf_size)

# fit model and get cross validation scores
knn_scores1 = cross_validate(knn_tuned, X, Y, cv=5, scoring=('accuracy',
'f1','precision','recall','roc_auc'), return_train_score=True)

knn_mn_acc = np.mean(list(knn_scores1.values())[2])
knn_mn_f1 = np.mean(list(knn_scores1.values())[3])
knn_mn_precis = np.mean(list(knn_scores1.values())[4])
knn_mn_recall = np.mean(list(knn_scores1.values())[5])
knn_mn_auc = np.mean(list(knn_scores1.values())[6])
```

MLP

```
from sklearn.neural_network import MLPClassifier
mlp= MLPClassifier()

# get optimal hyperparameters
mlp_grid_values = {
   'hidden_layer_sizes': [(20,), (6,6), (10,30,10)],
   'activation': ['tanh', 'relu'],
   'solver': ['sgd', 'adam'],
   'alpha': [0.0001, 0.05],
   'learning_rate': ['constant','adaptive']}
mlp_gridsearch = GridSearchCV(mlp, mlp_grid_values, n_jobs=-1, cv=5, scoring = 'recall')
mlp_best_params = mlp_gridsearch.fit(X, Y)

best_hiddenlayersize = mlp_best_params.best_estimator_.get_params()['hidden_layer_sizes']
best_activ_func = mlp_best_params.best_estimator_.get_params()['activation']
best_solver = mlp_best_params.best_estimator_.get_params()['solver']
best_alpha = mlp_best_params.best_estimator_.get_params()['alpha']
best_learning_rt = mlp_best_params.best_estimator_.get_params()['learning_rate']

# tuned model
mlp_tuned = MLPClassifier(
        hidden_layer_sizes=best_hiddenlayersize,
        alpha= best_alpha,
        beta_2=0.9,
        solver= best_solver,
        learning_rate = best_learning_rt,
        activation = best_activ_func,
        verbose=True)

# fit model and get cross validation scores
mlp_scores = cross_validate(mlp_tuned, X, Y, cv=5, scoring=('accuracy', 'f1','precision','recall','roc_auc'),
return_train_score=True)

mlp_mn_acc = np.mean(list(mlp_scores.values())[2])
mlp_mn_f1 = np.mean(list(mlp_scores.values())[3])
mlp_mn_precis = np.mean(list(mlp_scores.values())[4])
mlp_mn_recall = np.mean(list(mlp_scores.values())[5])
mlp_mn_auc = np.mean(list(mlp_scores.values())[6])
```

SVM

```
from sklearn.svm import SVC
svm = SVC()

# get optimal hyperparameters
svm_grid_values = {'C': [0.1,1, 10, 100], 'gamma': [1,0.1,0.01,0.001],'kernel': ['rbf', 'poly', 'sigmoid']}

svm_gridsearch = GridSearchCV(svm, svm_grid_values,refit=True,verbose=2, scoring = 'recall')
svm_best_params = svm_gridsearch.fit(X,Y)
best_C = svm_best_params.best_estimator_.get_params()['C']
best_gamma = svm_best_params.best_estimator_.get_params()['gamma']
best_kernel = svm_best_params.best_estimator_.get_params()['kernel']

# tuned model
svm_tuned =SVC(C= best_C, gamma = best_gamma, kernel = best_kernel)

# fit model and get cross validation scores
svm_scores = cross_validate(svm_tuned, X, Y, cv=5, scoring=('accuracy',
'f1','precision','recall','roc_auc'), return_train_score=True)

svm_mn_acc = np.mean(list(svm_scores.values())[2])
svm_mn_f1 = np.mean(list(svm_scores.values())[3])
svm_mn_precis = np.mean(list(svm_scores.values())[4])
svm_mn_recall = np.mean(list(svm_scores.values())[5])
svm_mn_auc = np.mean(list(svm_scores.values())[6])
```

Naïve Bayes

```
from sklearn.naive_bayes import GaussianNB
# Naive Bayes doesn't have any hyperparameters to tune.

nb = GaussianNB()

nb_scores = cross_validate(nb, X, Y, cv=5, scoring=('accuracy', 'f1','precision','recall','roc_auc'),
return_train_score=True)

nb_mn_acc = np.mean(list(nb_scores.values())[2])
nb_mn_f1 = np.mean(list(nb_scores.values())[3])
nb_mn_precis = np.mean(list(nb_scores.values())[4])
nb_mn_recall = np.mean(list(nb_scores.values())[5])
nb_mn_auc = np.mean(list(nb_scores.values())[6])
```

Decision Tree

```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier()

dt_grid_values = {'max_depth': [3, None], 'max_features': [np.random.randint(1, 9)], 'min_samples_leaf':
[np.random.randint(1, 9)], 'criterion': ['gini', 'entropy']}

dt_gridsearch = GridSearchCV(dt, dt_grid_values, refit=True, verbose=2, scoring = 'recall')
dt_best_params = dt_gridsearch.fit(X,Y)

best_max_depth = dt_best_params.best_estimator_.get_params()['max_depth']
best_max_features = dt_best_params.best_estimator_.get_params()['max_features']
best_min_samples_leaf= dt_best_params.best_estimator_.get_params()['min_samples_leaf']
best_criterion= dt_best_params.best_estimator_.get_params()['criterion']

# tuned model
dt_tuned = DecisionTreeClassifier(max_depth = best_max_depth, max_features= best_max_features,
min_samples_leaf = best_min_samples_leaf, criterion = best_criterion)

# fit model and get cross validation scores
dt_scores = cross_validate(dt_tuned, X, Y, cv=5, scoring=('accuracy', 'f1','precision','recall','roc_auc'),
return_train_score=True)

dt_mn_acc = np.mean(list(dt_scores.values())[2])
dt_mn_f1 = np.mean(list(dt_scores.values())[3])
dt_mn_precis = np.mean(list(dt_scores.values())[4])
dt_mn_recall = np.mean(list(dt_scores.values())[5])
dt_mn_auc = np.mean(list(dt_scores.values())[6])
```

Random Forest

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()

# Create hyperparameter grid
rf_grid_values ={
        "n_estimators":[25,50,100,500,1000],
        "max_features":[1,2,3,4,5,6],
        "min_samples_leaf":[1,5,10],
        "min_samples_split":[2,5,10,15],
        "criterion" : ["gini", "entropy"]}

rf_gridsearch = GridSearchCV(rf, rf_grid_values, cv=3, n_jobs=-1, scoring = 'recall')

rf_best_params = rf_gridsearch.fit(X, Y)

best_n_estimators = rf_best_params.best_params_['n_estimators']
best_max_feat = rf_best_params.best_params_['max_features']
best_min_samples_leaf = rf_best_params.best_params_['min_samples_leaf']
best_minsamples_split = rf_best_params.best_params_['min_samples_split']
best_crit = rf_best_params.best_params_['criterion']

rf_tuned=RandomForestClassifier(
        n_estimators= best_n_estimators,
        max_features=best_max_feat,
        min_samples_leaf= best_min_samples_leaf,
        min_samples_split= best_minsamples_split,
        criterion= best_crit,
        oob_score = True)

rf_scores = cross_validate(rf_tuned, X, Y, cv=5, scoring=('accuracy', 'f1','precision','recall','roc_auc'),
return_train_score=True)

rf_mn_acc = np.mean(list(rf_scores.values())[2])
rf_mn_f1 = np.mean(list(rf_scores.values())[3])
rf_mn_precis = np.mean(list(rf_scores.values())[4])
rf_mn_recall = np.mean(list(rf_scores.values())[5])
rf_mn_auc = np.mean(list(rf_scores.values())[6])
```

Gradient Boosting

```
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier()

gb_grid_values ={'learning_rate':[0.15,0.1,0.05,0.01,0.005,0.001],
        'n_estimators':[100,250,500,750,1000,1250,1500,1750]}

gb_gridsearch = GridSearchCV(gb, gb_grid_values, cv=3, n_jobs=-1, scoring = 'recall')

gb_best_params = gb_gridsearch.fit(X, Y)

best_n_estimators = gb_best_params.best_params_['n_estimators']
best_learning_rate= gb_best_params.best_params_['learning_rate']

gb_tuned = GradientBoostingClassifier(
        n_estimators= best_n_estimators,
        learning_rate= best_learning_rate)

gb_scores = cross_validate(gb_tuned, X, Y, cv=5, scoring=('accuracy', 'f1','precision','recall','roc_auc'),
return_train_score=True)

gb_mn_acc = np.mean(list(gb_scores.values())[2])
gb_mn_f1 = np.mean(list(gb_scores.values())[3])
gb_mn_precis = np.mean(list(gb_scores.values())[4])
gb_mn_recall = np.mean(list(gb_scores.values())[5])
gb_mn_auc = np.mean(list(gb_scores.values())[6])
```

xgboost

```
from xgboost import XGBClassifier
xgb=XGBClassifier()

xgb_grid_values = {'max_depth':[4,5,6], 'min_child_weight':[4,5,6]}

xgb_gridsearch = GridSearchCV(xgb, xgb_grid_values, cv=3, n_jobs=-1, scoring = 'recall')

xgb_best_params = xgb_gridsearch.fit(X, Y)

best_max_depth = xgb_best_params.best_params_['max_depth']
best_min_child_weight = xgb_best_params.best_params_['min_child_weight']

xgb_tuned= XGBClassifier (max_depth = best_max_depth, min_child_weight= best_min_child_weight)

xgb_scores = cross_validate(xgb_tuned, X, Y, cv=5, scoring=('accuracy', 'f1','precision','recall','roc_auc'),
return_train_score=True)

xgb_mn_acc = np.mean(list(xgb_scores.values())[2])
xgb_mn_f1 = np.mean(list(xgb_scores.values())[3])
xgb_mn_precis = np.mean(list(xgb_scores.values())[4])
xgb_mn_recall = np.mean(list(xgb_scores.values())[5])
xgb_mn_auc = np.mean(list(xgb_scores.values())[6])
```

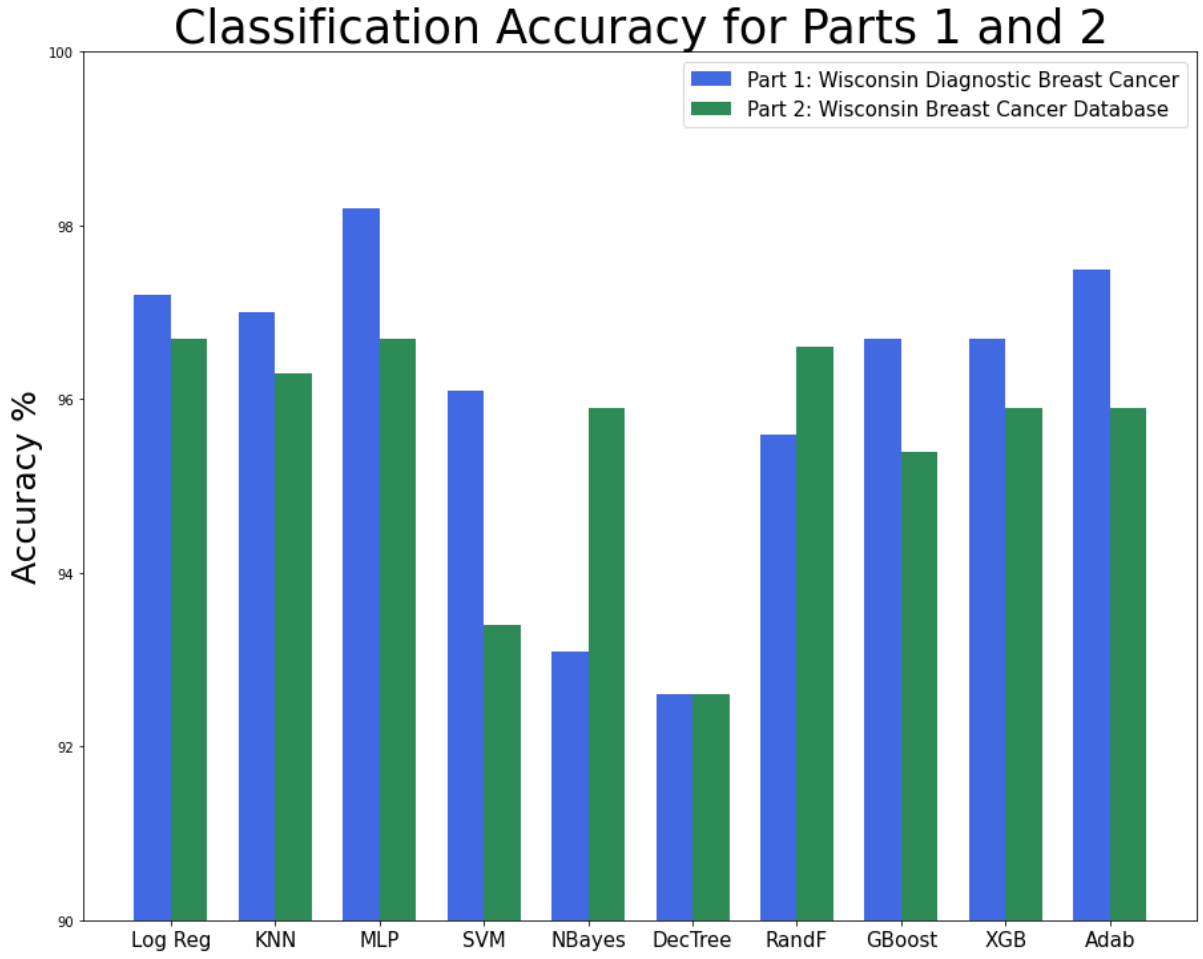
adaboost

```
from sklearn.ensemble import AdaBoostClassifier
ada= AdaBoostClassifier(n_estimators=50, learning_rate=1)

ada_grid_values = {'n_estimators':[ 10, 50, 100, 500,1000,2000],'learning_rate':[0.0001, 0.001, 0.01, 0.1, 1]}

ada_gridsearch = GridSearchCV(estimator=ada, param_grid= ada_grid_values, cv=3, n_jobs=-1, scoring = 'recall')
ada_best_params = ada_gridsearch.fit(X, Y)

best_n_estimators= ada_best_params.best_params_['n_estimators']
best_learning_rate= ada_best_params.best_params_['learning_rate']

ada_tuned = AdaBoostClassifier(n_estimators = best_n_estimators, learning_rate = best_learning_rate)

ada_scores = cross_validate(ada_tuned, X, Y, cv=5, scoring=('accuracy', 'f1','precision','recall','roc_auc'), return_train_score=True)

ada_mn_acc = np.mean(list(ada_scores.values())[2])
ada_mn_f1 = np.mean(list(ada_scores.values())[3])
ada_mn_precis = np.mean(list(ada_scores.values())[4])
ada_mn_recall = np.mean(list(ada_scores.values())[5])
ada_mn_auc = np.mean(list(ada_scores.values())[6])
```

Summarise all results

```
summary = np.empty((10,5))
summary[:] = np.NaN

summary[0,0:5] = np.array([logreg_mn_acc, logreg_mn_recall, logreg_mn_precis, logreg_mn_f1,logreg_mn_auc])
summary[1,0:5] = np.array([knn_mn_acc, knn_mn_recall, knn_mn_precis, knn_mn_f1,knn_mn_auc])
summary[2,0:5] = np.array([mlp_mn_acc, mlp_mn_recall, mlp_mn_precis, mlp_mn_f1,mlp_mn_auc])
summary[3,0:5] = np.array([svm_mn_acc, svm_mn_recall, svm_mn_precis, svm_mn_f1,svm_mn_auc])
summary[4,0:5] = np.array([nb_mn_acc, nb_mn_recall, nb_mn_precis, nb_mn_f1,nb_mn_auc])
summary[5,0:5] = np.array([dt_mn_acc, dt_mn_recall, dt_mn_precis, dt_mn_f1,dt_mn_auc])
summary[6,0:5] = np.array([rf_mn_acc, rf_mn_recall, rf_mn_precis, rf_mn_f1,rf_mn_auc])
summary[7,0:5] = np.array([gb_mn_acc, gb_mn_recall, gb_mn_precis, gb_mn_f1,gb_mn_auc])
summary[8,0:5] = np.array([xgb_mn_acc, xgb_mn_recall, xgb_mn_precis, xgb_mn_f1,xgb_mn_auc])
summary[9,0:5] = np.array([ada_mn_acc, ada_mn_recall, ada_mn_precis, ada_mn_f1,ada_mn_auc])

summary = 100*np.round(summary,3)
```

```
percs_part1=[97.2,97, 98.2, 96.1,93.1,92.6,95.6,96.7,96.7,97.5]

gp_xlocs = np.arange(10)
barwidth = 0.35

fig = plt.figure(figsize = (15,12))
ax = fig.add_subplot(111)
rects1 = ax.bar(gp_xlocs, percs_part1, barwidth, color='royalblue')

percs_part2 = [96.7,96.3,96.7,93.4,95.9,92.6,96.6,95.4,95.9,95.9]

rects2 = ax.bar(gp_xlocs + barwidth, percs_part2, barwidth, color='seagreen')

ax.set_ylabel('Accuracy %', fontsize=25)
ax.set_title('Classification Accuracy for Parts 1 and 2', fontsize=35)
ax.set_xticks(gp_xlocs + barwidth/ 2)
ax.set_xticklabels(('Log Reg','KNN', 'MLP', 'SVM', 'NBayes', 'DecTree','RandF', 'GBoost', 'XGB', 'Adab') ,
fontsize=15)
axes = plt.gca()
axes.set_ylim([90,100])

ax.legend( (rects1[0], rects2[0]), ('Part 1: Wisconsin Diagnostic Breast Cancer', 'Part 2: Wisconsin Breast
Cancer Database') , fontsize=15)

plt.show()
```



About the Author

Felix Beacher | felix@coolclinical.com

Felix Beacher is the founder of Cool Clinical. Felix has a PhD in neuroscience and has worked in drug development and various therapy areas including neurodegeneration and cancer.

Legal disclaimer

This publication has been written in general terms and is not intended to be relied on to cover specific situations. Any application of the information given in this publication will depend upon the particular circumstances involved. As such, we recommend that professional advice is sought before acting or refraining from acting on any of the contents of this publication. This publication and the information contained herein is provided “as is”. Cool Clinical makes no express or implied warranties that this publication is error-free or meet any particular criterion of performance or quality.

Cool Clinical accepts no duty of care or liability for any loss to any person acting or refraining from action as a result of any material in this publication.

