# Prediction using Bagging with Random Forest

# Concept Session

## Demo - 6.1: Bagging with Random Forest

**We will use the Ensemble method: Bagging with Random Forest. To compare the results, we will also evaluate a simple Decision Tree.**

In [ ]:
```
# general imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from numpy import mean
from numpy import std

import warnings
warnings.filterwarnings("ignore")
```

In [ ]:
```
# data imports
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import LabelEncoder
```

In [ ]:
```
# evaluation imports
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
```

### 1. Load the Data

We use the UCI breast cancer dataset to classify tumors as being malignant or benign.

We use the scikit-learn API to import the dataset into our program.

In [ ]:
```
# load data
breast_cancer = load_breast_cancer()

X = pd.DataFrame(breast_cancer.data, columns=breast_cancer.feature_names)
y = pd.Categorical.from_codes(breast_cancer.target, breast_cancer.target_names)
```

In [ ]:
```
print(y.describe)
```

**Since the label is categorical, it must be encoded as numbers. As the result, malignant is set to 1 and benign to 0.**

In [ ]:
```
# encode data
encoder = LabelEncoder()
binary_encoded_y = pd.Series(encoder.fit_transform(y))
```

### Learn Ensembles

We will evaluate all models using repeated stratified k-fold cross-validation, with three repeats and 10 folds.

We will report the mean and standard deviation of the F1-Score of the model across all repeats and folds.

### 2. Baseline: Decision Tree Classifier

In [ ]:
```
from sklearn.tree import DecisionTreeClassifier
```

In [ ]:
```
# define the model
model = DecisionTreeClassifier()

# evaluate the model
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, binary_encoded_y, scoring='f1', cv=cv, n_jobs=-1, error_score='raise')

# report performance
print('F1-Score: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

### 3. Bagging with Random Forest

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
```

In [ ]:
```
# define the model
model = RandomForestClassifier()

# evaluate the model
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, binary_encoded_y, scoring='f1', cv=cv, n_jobs=-1, error_score='raise')

# report performance
print('F1-Score: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

### Conclusion

For this dataset, we can see that the Ensemble method Bagging with default hyperparameters achieves a higher classification F1-Score than a simple Decision Tree Classifier.

### 4. Using Hyperparameters - Random Forest

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=42)
X_train.shape, X_test.shape
```

In [ ]:
```
classifier_rf = RandomForestClassifier(random_state=42, n_jobs=-1, max_depth=5, n_estimators=100, oob_score=True)
```

In [ ]:
```
%%time
classifier_rf.fit(X_train, y_train)
```

**Out of bag (OOB) score is a way of validating the Random forest model.**

In [ ]:
```
# checking the oob score
classifier_rf.oob_score_
```

### 5. Hyperparameter tuning for Random Forest using GridSearchCV and fit the data.

In [ ]:
```
rf = RandomForestClassifier(random_state=42, n_jobs=-1)
```

In [ ]:
```
params = {
    'max_depth': [2,3,5,10,20],
    'min_samples_leaf': [5,10,20,50,100,200],
    'n_estimators': [10,25,30,50,100,200]
}
```

***Grid Search is an exhaustive search on a manually specified hyperparameter search region.***

1. Select hyperparameters to optimize.
2. Select values to try out.
3. For each combination of the hyperparameter, build and test the model.
4. Choose the hyperparameter combination with the best performance.

In [ ]:
```
from sklearn.model_selection import GridSearchCV
```

In [ ]:
```
# Instantiate the grid search model
grid_search = GridSearchCV(estimator=rf,
                           param_grid=params,
                           cv = 4,
                           n_jobs=-1, verbose=1, scoring="accuracy")
```

**scoring - Strategy to evaluate the performance of the cross-validated model on the test set.**

**verbose - Controls the verbosity: the higher, the more messages.**

greater than 1 value : the computation time for each fold and parameter candidate is displayed;

greater than 2 value : the score is also displayed;

greater than 3 value : the fold and candidate parameter indexes are also displayed together with the starting time of the computation.

In [ ]:
```
%%time
grid_search.fit(X_train, y_train)
```

In [ ]:
```
grid_search.best_score_
```

In [ ]:
```
rf_best = grid_search.best_estimator_
rf_best
```

### 6. Visualization

In [ ]:
```
from sklearn.tree import plot_tree
model = RandomForestClassifier()
plt.figure(figsize=(80,40))
plot_tree(rf_best.estimators_[5], feature_names = X.columns,class_names=['benign', "malignant"],filled=True);
```

In [ ]:
```
from sklearn.tree import plot_tree
model = RandomForestClassifier()
plt.figure(figsize=(80,40))
plot_tree(rf_best.estimators_[7], feature_names = X.columns,class_names=['benign', "malignant"],filled=True);
```

The trees created by estimators*[5] and estimators*[7] are different. Thus we can say that each tree is independent of the other.

### 7. Sort the data with the help of feature importance

In [ ]:
```
rf_best.feature_importances_
```

In [ ]:
```
imp_df = pd.DataFrame({
    "Varname": X_train.columns,
    "Imp": rf_best.feature_importances_
})
```

In [ ]:
```
imp_df.sort_values(by="Imp", ascending=False)
```