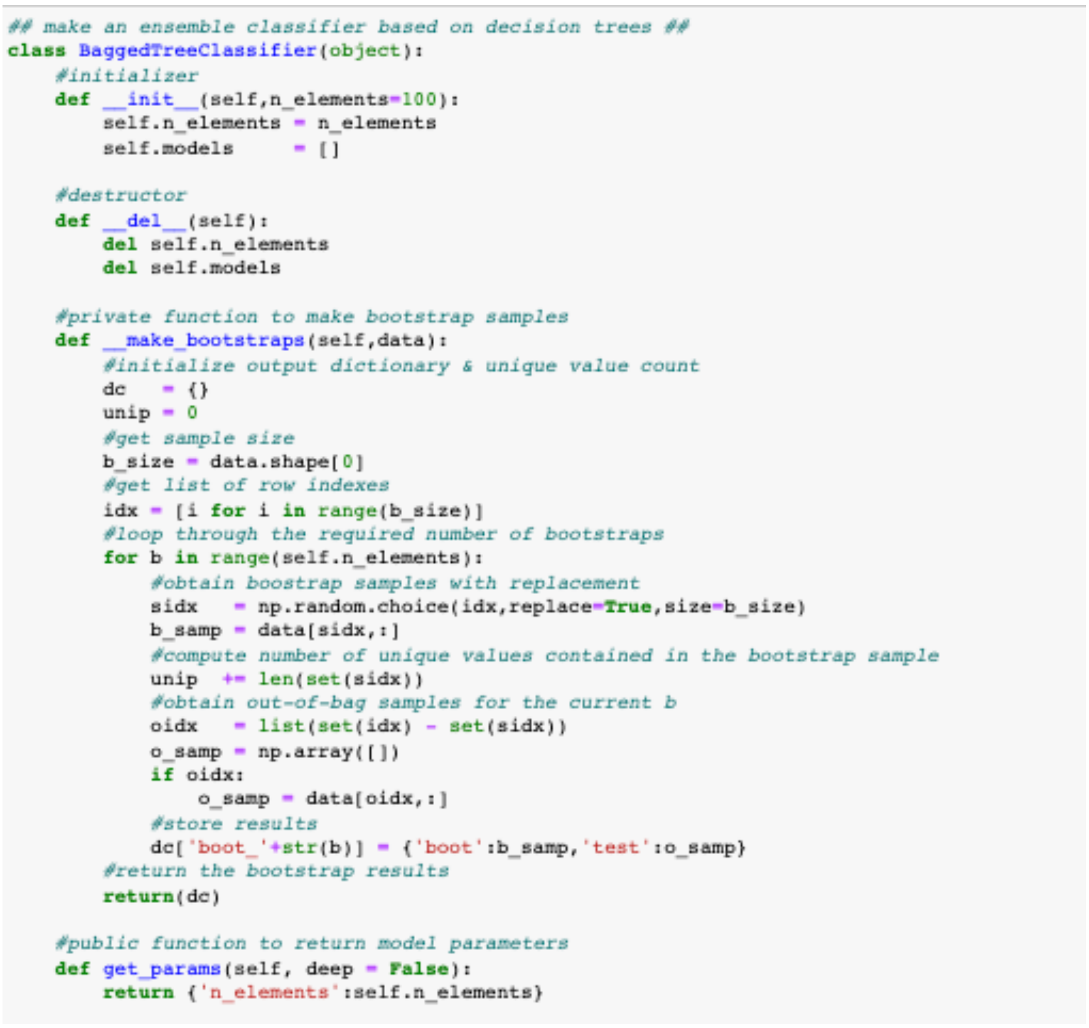
Ensemble methods Bagging

This code defines a Python class called BaggedTreeClassifier for creating an ensemble classifier based on decision trees. The ensemble is created using a technique called bootstrap aggregating or bagging. In bagging, multiple decision trees are trained on different bootstrap samples of the training data, and the ensemble's predictions are made by aggregating the predictions of individual trees.

1. Custom Bagging Tree **Classifier: Class** to encapsulate the functionality required for bagging ensemble. To generate the bootstrap samples. As we'll be investigating a classification dataset, the weak learner we will use here, to build our ensemble, is the decision tree classifier.

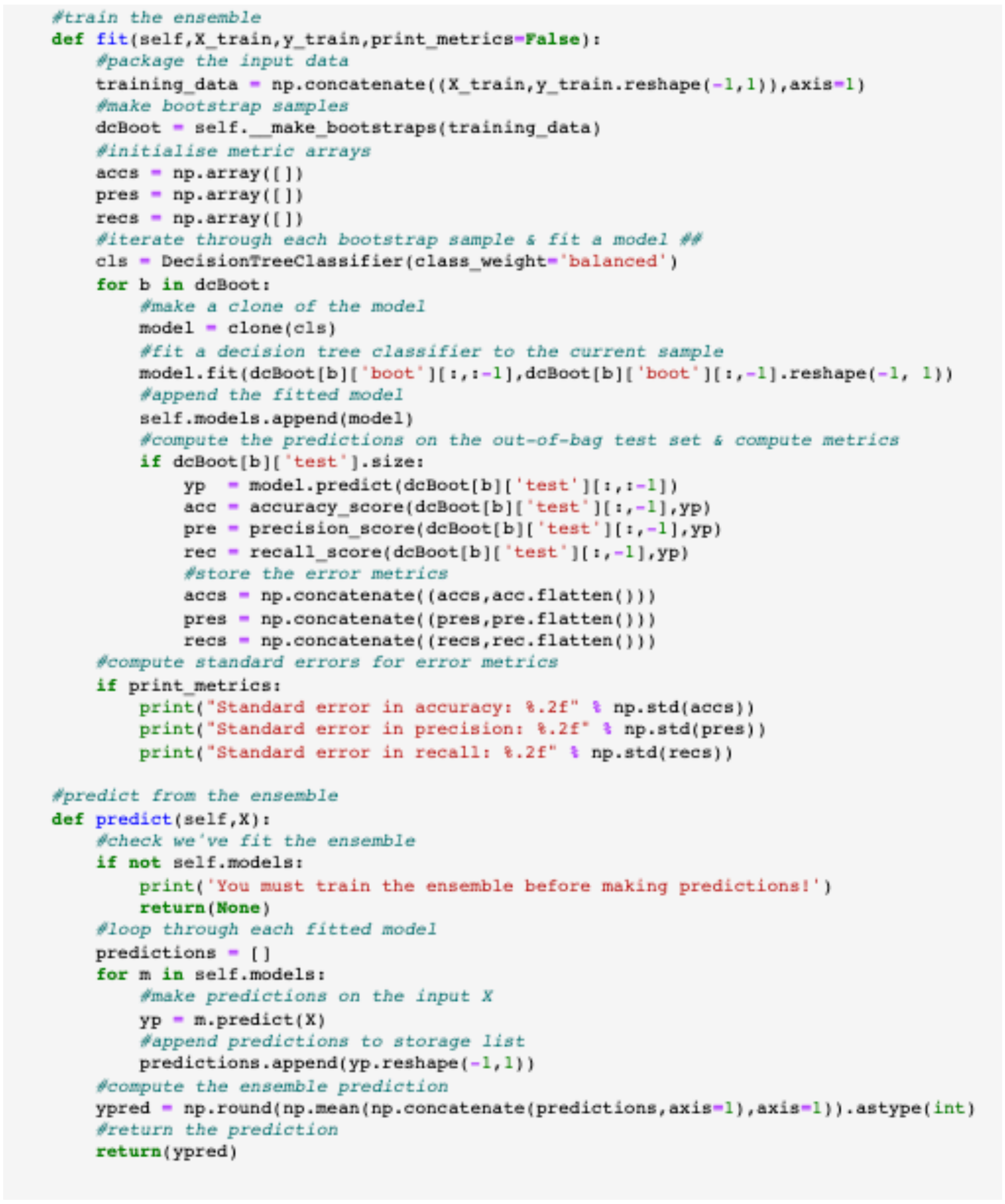


```
## make an ensemble classifier based on decision trees ##
class BaggedTreeClassifier(object):
    #initializer
    def __init__(self,n_elements=100):
        self.n_elements = n_elements
        self.models     = []

    #destructor
    def __del__(self):
        del self.n_elements
        del self.models

    #private function to make bootstrap samples
    def __make_bootstraps(self,data):
        #initialize output dictionary & unique value count
        dc   = {}
        unip = 0
        #get sample size
        b_size = data.shape[0]
        #get list of row indexes
        idx = [i for i in range(b_size)]
        #loop through the required number of bootstraps
        for b in range(self.n_elements):
            #obtain boostrap samples with replacement
            sidx   = np.random.choice(idx,replace=True,size=b_size)
            b_samp = data[sidx,:]
            #compute number of unique values contained in the bootstrap sample
            unip  += len(set(sidx))
            #obtain out-of-bag samples for the current b
            oidx   = list(set(idx) - set(sidx))
            o_samp = np.array([])
            if oidx:
                o_samp = data[oidx,:]
            #store results
            dc['boot_'+str(b)] = {'boot':b_samp,'test':o_samp}
        #return the bootstrap results
        return(dc)

    #public function to return model parameters
    def get_params(self, deep = False):
        return {'n_elements':self.n_elements}
```



```python
    #train the ensemble
    def fit(self,X_train,y_train,print_metrics=False):
        #package the input data
        training_data = np.concatenate((X_train,y_train.reshape(-1,1)),axis=1)
        #make bootstrap samples
        dcBoot = self.__make_bootstraps(training_data)
        #initialise metric arrays
        accs = np.array([])
        pres = np.array([])
        recs = np.array([])
        #iterate through each bootstrap sample & fit a model ##
        cls = DecisionTreeClassifier(class_weight='balanced')
        for b in dcBoot:
            #make a clone of the model
            model = clone(cls)
            #fit a decision tree classifier to the current sample
            model.fit(dcBoot[b]['boot'][:,:-1],dcBoot[b]['boot'][:,-1].reshape(-1, 1))
            #append the fitted model
            self.models.append(model)
            #compute the predictions on the out-of-bag test set & compute metrics
            if dcBoot[b]['test'].size:
                yp  = model.predict(dcBoot[b]['test'][:,:-1])
                acc = accuracy_score(dcBoot[b]['test'][:,-1],yp)
                pre = precision_score(dcBoot[b]['test'][:,-1],yp)
                rec = recall_score(dcBoot[b]['test'][:,-1],yp)
                #store the error metrics
                accs = np.concatenate((accs,acc.flatten()))
                pres = np.concatenate((pres,pre.flatten()))
                recs = np.concatenate((recs,rec.flatten()))
        #compute standard errors for error metrics
        if print_metrics:
            print("Standard error in accuracy: %.2f" % np.std(accs))
            print("Standard error in precision: %.2f" % np.std(pres))
            print("Standard error in recall: %.2f" % np.std(recs))

    #predict from the ensemble
    def predict(self,X):
        #check we've fit the ensemble
        if not self.models:
            print('You must train the ensemble before making predictions!')
            return(None)
        #loop through each fitted model
        predictions = []
        for m in self.models:
            #make predictions on the input X
            yp = m.predict(X)
            #append predictions to storage list
            predictions.append(yp.reshape(-1,1))
        #compute the ensemble prediction
        ypred = np.round(np.mean(np.concatenate(predictions,axis=1),axis=1)).astype(int)
        #return the prediction
        return(ypred)
```

-get_params(self,deep=False) : this function returns the input model parameters, and is only included here so that we can make use of scikit-learns cross-validation functionality -fit(self,X_train,y_train,print_metrics=False) : this function trains the ensemble using the provided arrays of predictors, X_train, and labels, y_train. If the print_metrics argument is set to True, the out-of-bag samples will be used to compute the standard error on our evaluation metrics. Note however that these data are unbalanced, and as such we will have to address this fact while modelling (Each individual decision tree classifier has been set with class_weight = 'balanced' for just this reason).

-predict(self,X) : this function produces predictions from the trained ensemble using the input X array of predictors.

2. Dataset: We can now proceed to load in the data set. Here we use the same breast cancer data set previously analysed in the logistic regression & decision tree posts. As such, I will not do any data exploration here.

```
## load classification dataset ##
data = load_breast_cancer()
X    = data.data
y    = data.target
```

3. Training: An instance of our ensemble class can now be made and trained. To reveal the standard errors in our evaluation metrics,

```
## declare an ensemble instance with default parameters ##
ens = BaggedTreeClassifier()

## train the ensemble & view estimates for prediction error ##
ens.fit(X,y,print_metrics=True)
```

4. Evaluation: Let's now measure the performance of our ensemble! However instead of doing a simple train-test split, we will be using k-fold cross validation. This involves making a series of train-test splits on the data, and evaluating the ensemble on each partition. The results from each evaluation can then be combined to yield measurements that are less sensitive to any specific splitting of the data. Luckily, scikit-learn provides a function to take care of most of the work for us:

Let's now measure the performance of our ensemble! However instead of doing a simple train-test split, we will be using k-fold cross validation. This involves making a series of train-test splits on the data, and evaluating the ensemble on each partition. The results from each evaluation can then be combined to yield measurements that are less sensitive to any specific splitting of the data. Luckily, scikit-learn provides a function to take care of most of the work for us:

```
## use k fold cross validation to measure performance ##
scoring_metrics = ['accuracy','precision','recall']
dcScores        = cross_validate(ens,X,y,cv=StratifiedKFold(10),scoring=scoring_metrics)
print('Mean Accuracy: %.2f' % np.mean(dcScores['test_accuracy']))
print('Mean Precision: %.2f' % np.mean(dcScores['test_precision']))
print('Mean Recall: %.2f' % np.mean(dcScores['test_recall']))
```

5. Comparing with scikit learn: Let's compare how our custom-built ensemble compares to the one available through scikit-learn:

```python
## import the scikit-learn model ##
from sklearn.ensemble import BaggingClassifier

## declare a bagging classifier instance ##
ens = BaggingClassifier(base_estimator=DecisionTreeClassifier(class_weight='balanced'),n_estimators=100)

## use k fold cross validation to measure performance ##
scoring_metrics = ['accuracy','precision','recall']
dcScores        = cross_validate(ens,X,y,cv=StratifiedKFold(10),scoring=scoring_metrics)
print('Mean Accuracy: %.2f' % np.mean(dcScores['test_accuracy']))
print('Mean Precision: %.2f' % np.mean(dcScores['test_precision']))
print('Mean Recall: %.2f' % np.mean(dcScores['test_recall']))
```