Decision Tree Classifier

It is a widely used simple classifier which has many applications in the field of machine learning. The simplicity lies in the theoretical background of the machine learning classification techniques. Decision Tree Classifier poses a series of carefully crafted questions (decision questions) about the attributes of the test record. Each time time it receive an answer, a follow-up question is asked until a conclusion about the class label of the record is reached.
The decision tree classifiers organised a series of test questions and conditions in a tree structure.

Building the Decision Tree / Training

Build an optimal decision tree is key problem in decision tree classifier. In general, may decision trees can be constructed from a given set of attributes. While some of the trees are more accurate than others, finding the optimal tree is computationally infeasible because of the exponential size of the search space.
Hence to make the decision tree a GREEDY approach is applied at each decision step based on the metric used for split - ‘mean’ or ‘median'. The greedy strategy is based on a metric for split. These metrics correspond to the impurity at the following step. It can be based on the following:

1. $Gini : Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$
2. $Entropy : H(E) = -\sum_{j=1}^{c} p_j \log p_j$

The information gain is calculated for the node before and after split

$$IG(Y, X) = E(Y) - E(Y|X)$$

Information Gain from X on Y

The nodes are split until the the nodes are pure or information gain has some minimum value.

The following results were obtained on titanic dataset:

```
dt = DecisionTree(max_depth=6)
dt.train(X_train,Y_train)
```

```
pred = dt.predict(X_test)
```

```
from sklearn.metrics import accuracy_score
acc = accuracy_score(y_true=Y_test, y_pred=pred)
```

```
print(acc)
```

```
0.8461538461538461
```

For sklearn lib:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
```

```
DT = DecisionTreeClassifier(max_depth=6)
DT.fit(X_train,Y_train)
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=6,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

```
pred = DT.predict(X_test)
from sklearn.metrics import accuracy_score
acc = accuracy_score(y_true=Y_test, y_pred=pred)
print(acc)
```

```
0.8111888111888111
```

Documentation:

DecisionTree():

max_depth = 10000 : The maximum depth uptill which the decision tree is formed. INT value expected.

split_val_metric = 'median' : The metric on which to split the current node.
Expects values - ‘mean’ or ‘median’.
min_info_gain = 1e-7 : The minimum information gain below which the node is made a leaf. Float value expected.
split_node_criterion = 'gini' - Criteria used for information gain calculation.
Expects ‘gini’ or ‘entropy’.

train(X_train, Y_train):
X_train - The dataset containing the attributes and rows in form of numpy array or panda dataset to be trained on
Y_train. Label value for each row of the dataset. Numpy array or Panda dataset expected.

predict(X_test):
X_test: The dataset containing the attributes and rows in form of numpy array or panda dataset to be tested.
Returns: numpy array with predicted labels corresponding to each row of dataset passed.

Random Forest Classifier

Ensemble method is a way to aggregate less predictive base models to produce a better predictive model. Random forest is the prime example of ensemble machine learning method. Random forests ensembles various decision trees to produce a more generalized model by reducing the notorious over-fitting tendency of decision trees.
The two basic steps of Random Forest are as follows:

1) Feature bagging: bootstrap aggregating or bagging is a method of selecting a random number of samples from the original set with replacement. In feature bagging the original feature set is randomly sampled and passed onto different trees (without replacement since having redundant features makes no sense). This is done to decrease the correlation among trees. A feature with unmatched great importance will cause every decision tree to choose it for the first and possible consequent splits, this will make all the trees behave similarly and ultimately more correlated which is undesirable. The aim here is to make highly uncorrelated decision trees.
2) Aggregation: The core concept that makes random forests better than decision trees is aggregating uncorrelated trees. The idea is to create several crappy model trees (low depth) and average them out to create a better random forest. Mean of some random errors is zero hence we can expect generalized predictive results from our forest. In case of regression we can average out the prediction of each tree (mean) while in case of classification problems we can simply take the majority of the class voted by each tree (mode).

The performance on titanic dataset:

```
rf = RandomForest(n_trees=800, n_cores=4)
rf.train(X_train,Y_train)
```

```
800
```

```
pred = rf.predict(X_test)
from sklearn.metrics import accuracy_score
acc = accuracy_score(y_true=Y_test, y_pred=pred)
print(acc)
```

```
0.8111888111888111
```

Using sklearn:

```
rf = RandomForestClassifier(n_estimators=800, n_jobs=4)
rf.fit(X_train,Y_train)
pred = rf.predict(X_test)
from sklearn.metrics import accuracy_score
acc = accuracy_score(y_true=Y_test, y_pred=pred)
print(acc)
```

```
0.8531468531468531
```

Documentation:

RandomForest() :

n_tress =100. Number of estimator trees in the forest. Expected: INT values.

split_val_metric = 'median' : The metric on which to split the current node. Expects values - ‘mean’ or ‘median’.

min_info_gain = 1e-7 : The minimum information gain below which the node is made a leaf. Float value expected.

split_node_criterion = 'gini' - Criteria used for information gain calculation. Expects ‘gini’ or ‘entropy’.

max_features =’auto’

The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If “auto”, then max_features=sqrt(n_features).
- If “sqrt”, then max_features=sqrt(n_features) (same as “auto”).
- If “log2”, then max_features=log2(n_features).
- If None, then max_features=n_features.

bootstrap : boolean, optional (default=True)

Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

n_cores = 1:

The number of jobs to run in parallel for both fit and predict. Expects integer value greater than or equal to 1.

train(X_train, Y_train):

X_train - The dataset containing the attributes and rows in form of numpy array or panda dataset to be trained on

Y_train. Label value for each row of the dataset. Numpy array or Panda dataset expected.

predict(X_test):

X_test: The dataset containing the attributes and rows in form of numpy array or panda dataset to be tested.

Returns: numpy array with predicted labels corresponding to each row of dataset passed.

Logistic Regression:

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. Unlike linear regression which outputs continuous number values, logistic regression transforms its output using the logistic sigmoid function to return a probability value which can then be mapped to two or more discrete classes.

Logistic regression uses an equation as the representation, very much like linear regression. Input values (x) are combined linearly using weights or coefficient values (referred to as the Greek capital letter Beta) to predict an output value (y). A key difference from linear regression is that the output value being modeled is a binary values (0 or 1) rather than a numeric value.

Performance on titanic dataset:

```
In [25]: titanic = pd.read_csv('processed_titanic.csv')
         y = titanic['survived']
         x = titanic.drop(['survived'],axis = 1)

In [26]: from sklearn.model_selection import train_test_split
         xTrain, xTest, yTrain, yTest = train_test_split(x, y, test_size = 0.3, random_state = 0)

In [27]: from math import sqrt
         from sklearn.metrics import mean_squared_error
         model = LogisticRegression(regulariser='l2')
         model.train(xTrain,yTrain)
         y_pred = model.predict(xTest)
         rms = sqrt(mean_squared_error(yTest, y_pred))
         rms

         C:\Users\Kush Jain\Anaconda3\lib\site-packages\ipykernel_launcher.py:12: RuntimeWarning: overflow encountered in exp
           if sys.path[0] == '':

Out[27]: 0.5091750772173156
```

Performance using sklearn:

```
In [29]: from sklearn.linear_model import LogisticRegression as LR
         lr = LR(penalty='l2')
         lr.fit(xTrain,yTrain)

Out[29]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)

In [30]: from sklearn.metrics import mean_squared_error
         from math import sqrt

         y_pred1 = lr.predict(xTest)
         rms1 = sqrt(mean_squared_error(yTest, y_pred1))
         rms1

Out[30]: 0.4714045207910317
```

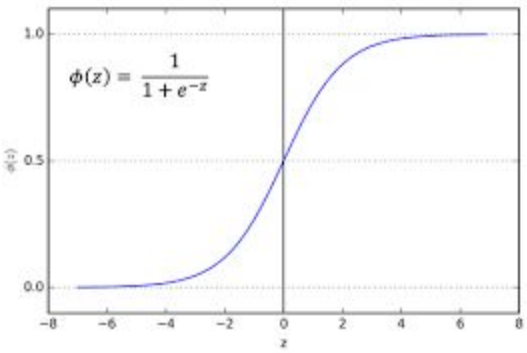
Documentation:

In the Logistic Regression classifier, the first function, **__init__()** is used to initialize the various parameters used in the class:

- regulariser parameter can take the values l1 or l2, depending on the regularization type to be used, with the default value being l2.
- lmbda is the hyperparameter of the regularization term, with the default value being 0.01.
- Num_steps is the number of steps for which the gradient descent will run for updating the weights, with the default value being 50.
- learning_rate is the size of the step taken in the gradient descent algorithm, with the default value being 0.01.
- initial_wts is the list of weights assigned for the updating of weights, with the default values for all the weights given as 1, but given as

Nonetype in the function, as it gave a RMSE score almost same as that of the sklearn library's Logistic Regression model.

The **sigmoid(z)** function returns the value of the hypothesis function of logistic regression, the sigmoid function, on a parameter z. This value lies between 0 and 1.



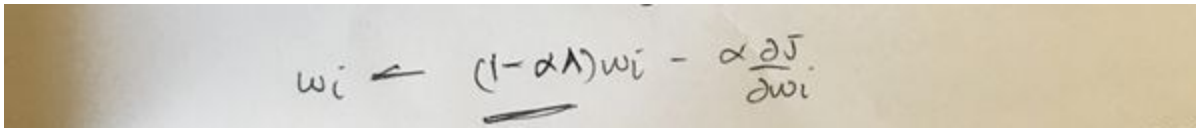
The **normalize(self)** function finds the maximum, minimum and the range (maximum – minimum) values of the weights, and normalizes the weights within the range [0,1], returning the initial_wts as an array of type float.

Derivative of logistic regression cost function:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i$$

The **updated_weights(self, X_tr, y_tr)** function takes the X and y training data points, appends a column to X having all values as 1 (because the initial_wts is a list of n+1 values), calculates the sigmoid of the product of the weights and X points to calculate the derivative of the cost function. We use this derivative in updating the weights according to the value of the regulariser parameter given.

Weights Update Equation:



The **train(self, X_tr, y_tr) function** sets the value of all the initial_wts as 1 if it is not given by the user (since the default value in __init__() is Nonetype), else normalizes the weights if they are given by the user. It runs the updated_weights() function for the value of num_steps given by the user, updating the weights in each iteration.

The **predict(self, X_test)** function first appends a column to the test dataframe with all elements having value 1 (because the initial_wts is a list of n+1 values), calculates the probability of y from the sigmoid function of the product of the test dataframe and the updated weights. Based on the value of the sigmoid function, the value for each row in the test data is set as 0 or 1, setting them 1 if the value is greater than 0.5 and 0 otherwise.

Stacking:

Stacked Generalization or stacking is an ensemble technique that uses a new model to learn how to best combine the predictions from two or more models trained on your dataset.
The predictions from the existing models or submodels are combined using a new model, and as such stacking is often referred to as blending, as the predictions from sub-models are blended together.

Performance on titanic dataset:

```
In [31]: titanic = pd.read_csv('processed_titanic.csv')
         y = titanic['survived']
         x = titanic.drop(['survived'],axis = 1)

In [32]: from sklearn.model_selection import train_test_split
         xTrain, xTest, yTrain, yTest = train_test_split(x, y, test_size = 0.3, random_state = 0)

In [33]: a = ([(LogisticRegression(), 2), (LogisticRegression(), 3)])
         s = Stack(a)
         s.train(xTrain,yTrain)

         C:\Users\Kush Jain\Anaconda3\lib\site-packages\ipykernel_launcher.py:12: RuntimeWarning: overflow encountered in exp
           if sys.path[0] == '':

In [34]: from sklearn.metrics import mean_squared_error
         from math import sqrt
         y_pred = s.predict(xTest)
         rms2 = sqrt(mean_squared_error(yTest, y_pred))
         rms2

         C:\Users\Kush Jain\Anaconda3\lib\site-packages\ipykernel_launcher.py:12: RuntimeWarning: overflow encountered in exp
           if sys.path[0] == '':

Out[34]: 0.8240220541217403
```

Documentation:

In the Stacking ensemble method, the first function, **__init__()** is used to initialize the various parameters used in the class:

- args is a list of tuples, where the first element of each tuple contains a classifier object and the second element contains the number of times it is to be considered, putting this data in allclf

The **train(self, X_tr, y_tr)** function trains each model with the previous features and appends the values of the prediction to the training dataset. This dataset is used to train the next classifier in allclf, and the next prediction column is appended to the current training dataset and so on.

The **predict(self, x_test)** function predicts each model with the previous features and appends the values of the prediction to the testing dataset. This dataset is used to predict the next classifier in allclf, and the next prediction column is appended to the current testing dataset and so on.

Naive Bayes Classifier :

The algorithm has been implemented to predict the classes of categorical datasets and numerical datasets using Multinomial Naive Bayes and Gaussian Naive Bayes respectively.
This classification is based on the probabilities of a particular instance belonging to a particular class based on Bayes' Rule which is given as follows :

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

The algorithm first learns likelihood of the data from the training set and then goes on to predict the classes for a given test set with the use of some user defined priors (based on some prior knowledge).
For Numerical datasets, P(X|C) is calculated as follows :

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The algorithm first divides the data according to the respective classes and then calculates the mean and standard deviation for each class and each attribute. Using these means and standard deviations it learns the probability of a particular instance occurs given a class label.

For categorical datasets, the same is calculated using simple counting. The algorithm follows the same procedure, only differing in the procedure for calculating the probability.

When the algorithm is run on the titanic dataset using only categorical variables and MultinomialNB we obtain an accuracy of 80.44

When this algorithm is run on the same dataset after encoding all attributes and using GaussianNB, we obtain an accuracy of 80.419

Documentation:
NaiveBayes()
Parameters : prior : dictionary-type. Contains probability of each class.
Type : string type. ‘Multinomial’ for categorical data.
‘Gaussian’ for numerical data.
Attributes : maxnum : Stores the maximum attribute value for each attribute of a categorical dataset

minnum : Stores the minimum value for each attribute of a categorical dataset.

ClassDiv : Stores the data as a dictionary corresponding to each class.

probablities : Stores the probability of each attribute for each class/

Functions : __init__(typec, prior)

mean(values):

Parameters : values: array-like

Returns: mean of the set of values passed.

variance(values):

Parameters : values: array-like

Returns: variance of the set of values passed.

train(X, y):

Train model on the passed data.

Parameters : X: array-like. Training sample

y: array-like. Training labels.

Returns: None

DatabyClass(X, y):

Parameters : X: array-like. Training sample

y: array-like. Training labels.

Attributes : divclass : dictionary type. Separate and store X by class label

Returns: divclass : dictionary type

Counts(values):

Parameters : values: array-like

Returns: Counts of each unique value in values as a dictionary.

CountVals(instances):
Parameters : instances: array-like. Data corresponding to each class
Returns: counts for each attribute for the given data.

CalcClassprob(ClassDiv):
Parameters : ClassDiv: dictionary-like. Data corresponding to each class
Returns: Dictionary containing the probability for each attribute corresponding to each class

calcPred(row):
Parameters : row: array-like. Data corresponding to a row in the test data
Returns: prediction of class for that row.

predict(X_test):
Parameters : X_test: array-like. Data corresponding to the test data
Returns: prediction for given test data.

AttrProb(instances):
Parameters : instances: array-like. Data corresponding to each class
Returns: probability value of each attribute

calcPGNB(x, avg, var):
Returns the probablity as per the formula given a particular attribute value.

calcPredGNB(row):
Predictions for Gaussian Naive Bayes.

Parameters : row: array-like. Data corresponding to a row in the test data

Returns: prediction of class for that row.

AdaBoost

This is an adaptive boosting technique that combines multiple weak classifiers to form a strong classifier. It starts with a weak classifier then calculates the misclassification of values. Based on this misclassification the score of the estimator varies and correspondingly the sample weights. The weights of the misclassified tuples increase so that further classifiers will concentrate on learning how to classify these. These tuples basically represent tuples that have been hard to classify. So now each classifier tries to classify these. The final predictions is done by majority voting, with the prediction of each classifier weighted with the estimator weights.

1. *Initialize the observation weights* $w_i = 1/n,\ i = 1, 2, \ldots, n.$
2. *For* $m = 1$ *to* M*:*
   (a) *Fit a classifier* $T^{(m)}(\boldsymbol{x})$ *to the training data using weights* w_i.
   (b) *Compute*
   $$err^{(m)} = \sum_{i=1}^{n} w_i \mathbb{I}\left(c_i \neq T^{(m)}(\boldsymbol{x}_i)\right) / \sum_{i=1}^{n} w_i.$$
   (c) *Compute*
   $$\alpha^{(m)} = \log \frac{1 - err^{(m)}}{err^{(m)}} + \log(K - 1).$$
   (d) *Set*
   $$w_i \leftarrow w_i \cdot \exp\left(\alpha^{(m)} \cdot \mathbb{I}\left(c_i \neq T^{(m)}(\boldsymbol{x}_i)\right)\right),\ i = 1, \ldots, n.$$
   (e) *Re-normalize* w_i.
3. *Output*
   $$C(\boldsymbol{x}) = \arg\max_k \sum_{m=1}^{M} \alpha^{(m)} \cdot \mathbb{I}(T^{(m)}(\boldsymbol{x}) = k).$$

The above algorithm shows exactly how AdaBoost SAMME works.

DOCUMENTATION :

AdaBoost():

Parameters : n_trees : int values. Maximum number of trees/classifiers.

Learning_rate : learning rate for changing weights.

Attributes : base_estimator : base Decision Tree

n : int. The number of samples given.

Labels : array-like. Contains all the classes in the sample.

Errors : array-like. Stores the error value of each estimator

Eweight : array-like. Stores the estimator weights of each estimator.

Classifiers : list. Stores all the weak classifier models.

Functions : __init__()

fit(X, y):

Parameters : X: array-like. Training sample

y: array-like. Training labels.

Create and train the ensemble of Decision Trees on the dataset.

adaboost(X, y, s_wt):

Parameters : X: array-like. Training sample

y: array-like. Training labels.

S_wt : array-like. Stores the sample weights for the model.

Returns : new sample weights, error and weights of the estimator.

predict(X):

Parameters : X: array-like. Test data

Predict the class label based on majority voting of the predictions of weak classifiers weighted with classifier weights.