Understanding AdaBoost

1. Access “Elements of Statistical Learning” by Trevor Hastie, Robert Tibshirani, Jerome Friedman
    a. https://solo.bodleian.ox.ac.uk/permalink/44OXF_INST/ao2p7t/cdi_askewsholts_vlebooks_9780387848587
2. Review Chapter 10 “Boosting and Additive Trees”
3. Review the online article: “AdaBoost from Scratch : Build your Python implementation of one of the most popular “off-the-shelf” algorithms in Data Science”, Alvaro Corrales Cano
    a. https://towardsdatascience.com/adaboost-from-scratch-37a936da3d50
4. Access the code for that blog here:
    a. https://github.com/AlvaroCorrales/AdaBoost
5. Access the dataset here:
    a. https://archive.ics.uci.edu/dataset/94/spambase
6. The above is a better source of reference / learning than my notes .. but FYI, here are my personal notes:
    a. The overall model is collection (ensemble) of (typically) weak classifiers $G_m(x)$
    b. Given a vector of predictor (input) variables X a classifier G(X) produces a prediction, and the error is measured as follows:

$$\overline{\text{err}} = \frac{1}{N}\sum_{i=1}^{N} I(y_i \neq G(x_i)),$$

    c. The algorithm repeatedly applies new, (weak) classifier, to updated versions of the data
        i. The data is modified by applying a weight to each observation in the training data
            1. Observations that are misclassified have their weights increased
            2. Observations that are correctly classified have their weights reduced
            3. In other words, the data set starts to ‘favour’ difficult cases ..the difficult cases become more important
            4. (How are the weights actually used .. see below)
    d. The algorithm also records the overall error rate for the classifier (called ‘alpha’ in the Hastie book) .. this is used in the final step when combining the set of weak classifiers into an overall ensemble model:

$$\text{Output } G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right].$$

e. How are the observation (sample) weights (W_i) actually used?

   i. Generally, sklearn classifiers accept a 'sample weight' parameter – a vector of weights for each sample. Those weights are used somewhat differently depending on the classifier. But generally have the same impact.

   ii. For example, sklearn 'decisionTreeClassifier' accepts 'sample weight' as a parameter:

      1. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier.fit

fit(*X, y, sample_weight=None, check_input=True*) [source]

Build a decision tree classifier from the training set (X, y).

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*

The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y : *array-like of shape (n_samples,) or (n_samples, n_outputs)*

The target values (class labels) as integers or strings.

sample_weight : *array-like of shape (n_samples,), default=None*

Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

   iii. As does LogisticRegression

fit(*X, y, sample_weight=None*) [source]

Fit the model according to the given training data.

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features)*

Training vector, where `n_samples` is the number of samples and `n_features` is the number of features.

y : *array-like of shape (n_samples,)*

Target vector relative to X.

sample_weight : *array-like of shape (n_samples,) default=None*

Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Added in version 0.17: *sample_weight* support to LogisticRegression.

   iv. As does SVM:

`fit(X, y, sample_weight=None)` [source]

Fit the SVM model according to the given training data.

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features) or (n_samples, n_samples)*

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y : *array-like of shape (n_samples,)*

Target values (class labels in classification, real numbers in regression).

sample_weight : *array-like of shape (n_samples,), default=None*

Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

V

v.