COMP0088
Introduction to Machine Learning
Lab Assignment 3

Matthew Caldwell

October 4, 2021

Task Summary

Week 3 Notes

The non-parametric classification models and ensembles in this week's assignment have losses that are not smooth and aren't suited to simple gradient-based optimisation. You are only asked to produce naïve, brute force implementations, but note that these can scale really badly, so be cautious about running the algorithms with large sample sizes. You may find it interesting to think about what strategies could be used to speed things up.

The assignment script is `week_3.py`, generating output file `week_3.pdf`. There are some potentially-useful command line options, which you can list using:

```
$ python week_3.py -h
```

Example output from a completed assignment is shown in Fig. 1. Plotting functionality is provided, so your output should look similar, though by default the resolution for the class maps will be significantly lower (to avoid scaling issues).

The script uses a synthetic dataset, provided in `week_3_data.csv`, with both binary and 3-class labels (Fig. 2).

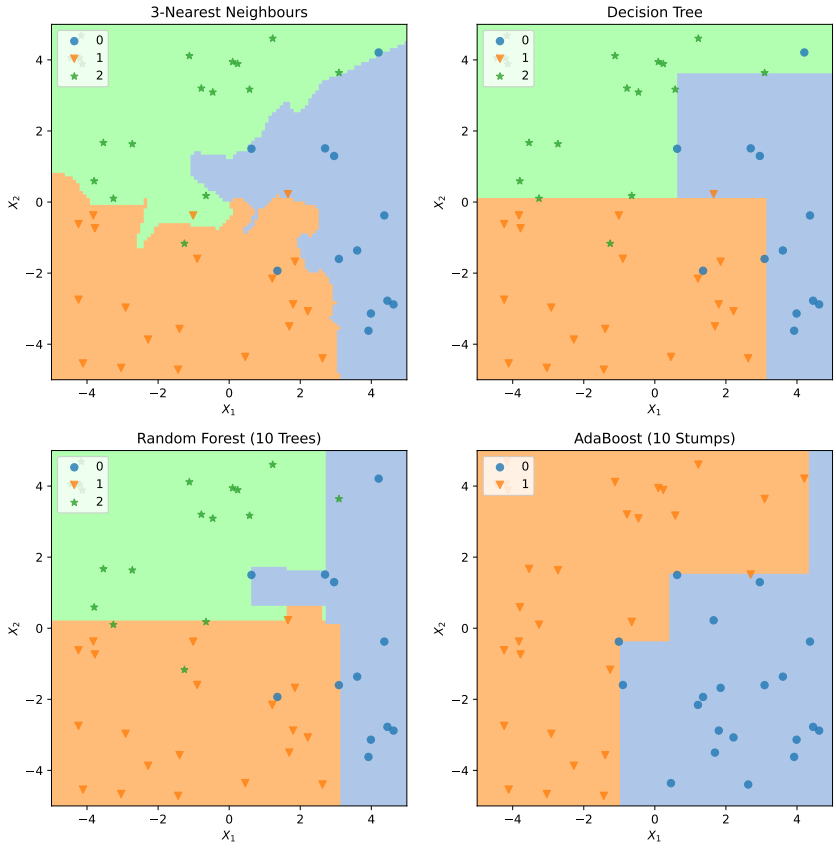

Figure 1: Example output from the `week_3.py` script

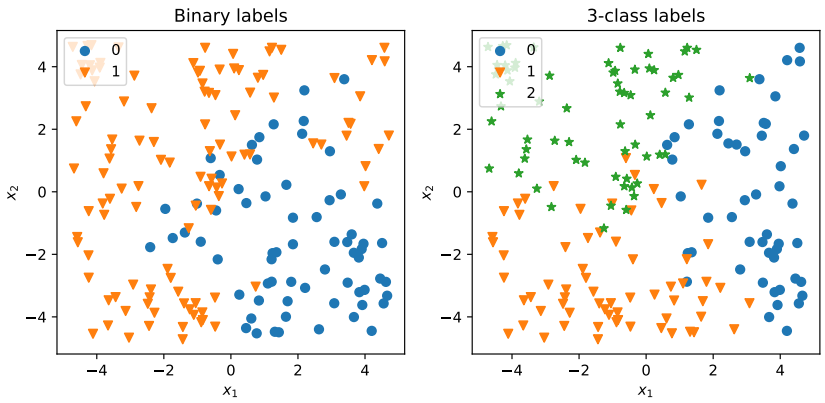

Figure 2: Week 3 dataset includes both binary (left) and 3-class (right) labels.

1 *k*-Nearest Neighbours

In a *k*-Nearest Neighbours classifier, test samples are compared directly against the samples in the training set and the *k* most similar—according to some chosen similarity or distance metric—vote on the class to predict. Training such a classifier consists simply of memorising the training set. For the implementation below, we will skip this step and just pass the training data directly to the prediction function.

Although other distance metrics may be useful for real problem classes, for the purposes of this exercise you can stick to a simple Euclidean distance.

You may find the function `vote` in the `utils` module useful.

1.1 Implement *k*-Nearest Neighbours prediction

Implement the body of the following function in `week_3.py`:

```
def nearest_neighbours_predict ( train_X, train_y, test_X, neighbours=1 ):
    """
    Predict labels for test data based on neighbourhood in
    training set.

    # Arguments:
        train_X: an array of sample data for training, where rows
            are samples and columns are features.
        train_y: vector of class labels corresponding to the training
            samples, must be same length as number of rows in X
        test_X: an array of sample data to generate predictions for,
            in same layout as train_X.
        neighbours: how many neighbours to canvass at each test point

    # Returns
        test_y: predicted labels for the samples in test_X
    """
    assert(train_X.shape[0] == train_y.shape[0])
    assert(train_X.shape[1] == test_X.shape[1])

    # TODO: implement the k-nearest neighbours algorithm
    return None
```

2 Decision Trees

Decision trees are both a learning model in their own right and an important constituent model for the ensemble methods in subsequent tasks. The model consists of a recursive sequence of binary tests, typically of inequalities on single feature values. These effectively partition the feature space into separate regions, each of which is then assigned the class that occurs most often among training samples within it.[1]

Trees can be trained by a greedy brute force search for the split that minimises some chosen loss. Various losses are possible, but for simplicity here (and compatibility with Task 4) we will use a **weighted misclassification error** throughout:

$$L(\mathbf{y}, \hat{\mathbf{y}}, \mathbf{w}) = \sum_i w_i \mathbf{1}(y_i \neq \hat{y}_i) \tag{1}$$

Trees are a naturally recursive data structure that almost cry out for a class-based implementation, but here we limit ourselves to using dicts to avoid adding syntactical distractions for students who may not be very familiar with Python classes.

2.1 Calculate misclassification error

Implement the body of the following function in `week_3.py`:

```
def misclassification ( y, cls, weights=None ):
    """
    Calculate (optionally-weighted) misclassification error for
    a given set of labels if assigned the given class.

    # Arguments
        y: a set of class labels
        cls: a candidate classification for the set
        weights: optional weights vector specifying relative
            importance of the samples labelled by y

    # Returns
        err: the misclassification error of the candidate labels
    """

    # TODO: implement weighted misclassification metric
    return 1
```

Note that if `weights` is not provided, you should default it to $\frac{1}{n}$.

2.2 Split a dataset to reduce misclassification

Implement the body of the following function in `week_3.py`:

```
def decision_node_split ( X, y, cls=None, weights=None, min_size=3 ):
    """
    Find (by brute force) a split point that best improves the weighted
    misclassification error rate compared to the original one (or not, if
```

[1] Equivalently, those training samples *vote* on the outcome. Once again, the `utils.vote` function may be useful here.

```
    there is no improvement possible).

    Features are assumed to be numeric and the test condition is
    greater-or-equal.

    # Arguments:
        X: an array of sample data, where rows are samples
            and columns are features.
        y: vector of class labels corresponding to the samples,
            must be same length as number of rows in X
        cls: class label currently assigned to the whole set
            (if not specified we use the most common class in y, or
            the lowest such if 2 or more classes occur equally)
        weights: optional weights vector specifying relevant importance
            of the samples
        min_size: don't create child nodes smaller than this

    # Returns:
        feature: index of the feature to test (or None, if no split)
        thresh: value of the feature to test (or None, if no split)
        c0: class assigned to the set with feature < thresh
            (or None, if no split)
        c1: class assigned to the set with feature >= thresh
            (or None, if no split)
    """
    assert(X.shape[0] == len(y))

    # TODO: implement this
    return None, None, None, None
```

Remember to account for the number of points in each child node—the weights should help with this. You should decide on some tie-break policy for when multiple splits produce the same loss improvement.

2.3 Train a decision tree classifier

Implement the body of the following function in `week_3.py`:

```
def decision_tree_train ( X, y, cls=None, weights=None,
                          min_size=3, depth=0, max_depth=10 ):
    """
    Recursively choose split points for a training dataset
    until no further improvement occurs.

    # Arguments:
        X: an array of sample data, where rows are samples
            and columns are features.
        y: vector of class labels corresponding to the samples,
            must be same length as number of rows in X
        cls: class label currently assigned to the whole set
            (if not specified we use the most common class in y, or
```

```
            the lowest such if 2 or more classes occur equally)
        weights: optional weights vector specifying relevant importance
            of the samples
        min_size: don't create child nodes smaller than this
        depth: current recursion depth
        max_depth: maximum allowed recursion depth

    # Returns:
        tree: a dict containing (some of) the following keys:
            'kind' : either 'leaf' or 'decision'
            'class' : the class assigned to this node (leaf)
            'feature' : index of feature on which to split (decision)
            'thresh' : threshold at which to split the feature (decision)
            'below' : a nested tree for when feature < thresh (decision)
            'above' : a nested tree for when feature >= thresh (decision)
    """
    # TODO: implement this
    return None
```

You should find that most of the hard work is already done by the `decision_node_split` function, and this is just responsible for managing the recursion.

2.4 Make predictions from a decision tree

Implement the body of the following function in `week_3.py`:

```
def decision_tree_predict ( tree, X ):
    """
    Predict labels for test data using a fitted decision tree.

    # Arguments
        tree: a decision tree dictionary returned by decision_tree_train
        X: an array of sample data, where rows are samples
            and columns are features.

    # Returns
        y: the predicted labels
    """
    # TODO: implement this
    return None
```

3 Random Forests

Random forests™ are an ensemble model aggregating the predictions from multiple decision trees. Diversity is introduced into the ensemble by training the trees on **bootstrap samples** from the training set, and also by restricting the subset of features used by each tree.

For the exercises below, we will forgo feature subsetting (we will only be using two features anyway) and focus on the **bagging** aspect.

3.1 Train a (simplified) random forest classifier

Implement the following function in `week_3.py`:

```
def random_forest_train ( X, y, k, rng, min_size=3, max_depth=10 ):
    """
    Train a (simplified) random forest of decision trees.

    # Arguments:
        X: an array of sample data, where rows are samples
            and columns are features.
        y: vector of binary class labels corresponding to the
            samples, must be same length as number of rows in X
        k: the number of trees in the forest
        rng: an instance of numpy.random.Generator
            from which to draw random numbers
        min_size: don't create child nodes smaller than this
        max_depth: maximum tree depth

    # Returns:
        forest: a list of tree dicts as returned by decision_tree_train
    """

    # TODO: implement this
    return None
```

Use the function you wrote in Task 2.3 to train the individual trees. The choice method of `numpy.random.Generator` will help with the bootstrap sampling.

3.2 Make predictions from a (simplified) random forest classifier

Implement the following function in `week_3.py`:

```
def random_forest_predict ( forest, X ):
    """
    Predict labels for test data using a fitted random
    forest of decision trees.

    # Arguments
        forest: a list of decision tree dicts
        X: an array of sample data, where rows are samples
            and columns are features.
```

```
    # Returns
        y: the predicted labels
    """
    # TODO: implement this
    return None
```

Once again, the `utils.vote` function may be useful.

4 AdaBoost

AdaBoost is a meta-algorithm that iteratively builds an ensemble of weak learners such that each new addition provides the best available marginal improvement in the ensemble performance. The new learner is chosen to minimise its weighted classification error (Eq. (1)) on the training set, with the sample weights updated at each iteration to prioritise misclassified points. The training procdure is shown in Algorithm 1.

AdaBoost is agnostic as to the class of weak learners used, but is commonly implemented using **decision stumps**—decision trees of depth 1—and that is what you should do here, using the decision tree functions you implemented in Task 2.

Algorithm 1 AdaBoost training for binary classification

Initialise sample weights $w_i = \frac{1}{n}, \quad i \in \{1, 2, ..., n\}$
for $t = 1$ to k **do**
  Fit classifier h_t to minimise misclassification error with weights w_i
  Set $\epsilon =$ the weighted misclassification error of h_t
  Compute prediction weight $\alpha_t = \log\left(\frac{1-\epsilon}{\epsilon}\right)$
  Update weights: $w_i \leftarrow w_i e^{\alpha_t \mathbf{1}(y_i \neq h_i(\mathbf{x}_i))}$
  Normalise weights: $w_i \leftarrow \frac{w_i}{\sum_j w_j}$
end for

Once the ensemble is trained, new samples are classified like this:

$$\hat{y} = \mathbf{1}\left(\sum_t \alpha_t h_t(\mathbf{x}) \geq 0\right) \tag{2}$$

Note that the training algorithm has been expressed in terms that don't require a particular binary labelling convention, but the prediction expression above assumes that the outputs of the classifiers h_t are $\{-1, 1\}$. This is *not* the case for the decision trees implemented in Task 2, nor for the data loaded by the `week_3.py` script. So you will need to convert the h_t outputs appropriately within the prediction sum.

4.1 Train an AdaBoost classifier

Implement the body of the following function in `week_3.py`:

```
def adaboost_train ( X, y, k, min_size=1, max_depth=1, epsilon=1e-8 ):
    """
    Iteratively train a set of decision tree classifiers
    using AdaBoost.

    # Arguments:
        X: an array of sample data, where rows are samples
            and columns are features.
        y: vector of binary class labels corresponding to the
            samples, must be same length as number of rows in X
        k: the maximum number of weak classifiers to train
        min_size: don't create child nodes smaller than this
        max_depth: maximum tree depth -- by default we just
            use decision stumps
        epsilon: threshold below which the error is considered 0
```

```
    # Returns:
        trees: a list of tree dicts as returned by decision_tree_train
        alphas: a vector of weights indicating how much credence to
            given each of the decision tree predictions
    """
    # TODO: implement this
    return None, None
```

4.2 Make predictions from an AdaBoost classifier

Implement the body of the following function in `week_3.py`:

```
def adaboost_predict ( trees, alphas, X ):
    """
    Predict labels for test data using a fitted AdaBoost
    ensemble of decision trees.

    # Arguments
        trees: a list of decision tree dicts
        alphas: a vector of weights for the trees
        X: an array of sample data, where rows are samples
            and columns are features.

    # Returns
        y: the predicted labels
    """
    # TODO: implement this
    return None
```

5 Further exploration

If you have exhausted the previous exercises, you might find it interesting to try out one or more of the following tasks. Doing so is entirely optional, but may help to cement or extend some of the ideas learned this week.

These tasks are not included in the `week_3.py` script skeleton. You can add them in or start from scratch in a new script of your own.

5.1 Adapt your decision trees to use a different loss

The decision trees for Task 2 use a weighted misclassification error in order to easily use them with AdaBoost. However, other loss functions may be preferable for some problems. Two common choices are the Gini impurity:

$$\sum_k p_k(1 - p_k) \tag{3}$$

and the entropy (or cross-entropy):

$$-\sum_k p_k \log p_k \tag{4}$$

where k ranges over all the classes present in the node and p_k is the fraction of samples in the node that are of class k.

Try modifying your implementation to use one or both of these losses and see what difference (if any) it makes to the splits chosen and the performance of the trees. (There is an implementation of `gini_impurity` in the `utils` module.)

5.2 Add feature subset selection to your random forests

For simplicity in Task 3 we omitted feature subsetting as a diversification mechanism in the ensemble. Try adding this and seeing if it makes much difference to your fits. You will need to adapt the data structure used for the forest to keep track of which trees are using which features so that the correct subsets can be used at test time.

You may want to find some higher-dimensional data to apply your modified, as the supplied data—having only two features—doesn't provide very much scope for variation. As mentioned last week, the `scikit-learn` datasets package can be a good source for data to play with.

5.3 Implement an ExtraTrees ensemble

Bagging and feature subsetting are not the only way to build a diverse ensemble of decision trees. In the ExtraTrees or **extremely randomised trees** approach, the training algorithm for the trees is modified instead. At each split point, rather than performing a brute force search over all possible splits, a random search is used instead: some specified (but relatively small) number of purely random candidate splits are evaluated, and the best of these is chosen. This can result in a high degree of variation in the ensemble that *may* better probe the structure of the data distribution.

Try implementing this fitting procedure and comparing its behaviour to that of the random forest. Is there any meaningful difference for our simple data?

5.4 Compare the behaviour of these models in `scikit-learn`

For practical applications, you will almost never need to implement classic ML models like decision trees and random forests yourself. Instead, you will typically use existing implementations such as those in the `scikit-learn` library. Because they are so widely used, these are likely to be more versatile, better optimised and better tested than your own code.

`scikit-learn` supports a wide range of ML models and algorithms, including all those in the lab exercises above: nearest neighbours, decision trees, AdaBoost and forests of randomised trees. Models have (mostly) consistent interfaces for training and evaluation, so it is usually straightforward to swap between different models and compare their behaviour.

We will see a bit more of this library in future weeks, but by all means get acquainted with it now if you have the time and inclination.