Model Building Auto ML

Jacob Holmshaw

June 2020

1 Artificial Neural Networks

1.1 The Mathematics

A neural network is defined as a directed graph whose nodes and edges respectively correspond to neurons and links between them. Each node, representing a neuron, receives as input a weighted sum of the outputs of the neurons connected to its incoming edges.

A **feedforward neural network** is a directed acyclic graph, $G = (V, E)$, equipped with a weight function $w : E \to \mathbb{R}$ over it's edges. Each node, which corresponds to a single neuron, is modelled as a scalar function, $\sigma : \mathbb{R} \to \mathbb{R}$, called the "activation" function of the neuron. Three possible options of σ are:

- the sign function, $\sigma(a) = sign(a)$,
- the threshold function, $\sigma(a) = 1\{a > 0\}$
- the sigmoid function, $\sigma(a) = \frac{1}{1+\exp{(-a)}}$, which is a smooth approximation to the threshold function.

The edges of the graph link the outputs of the neurons to the inputs of other neurons on the graph, according to the structure of the graph. The input of a neuron is obtained by taking a weighted sum of the outputs of all the neurons connected to it, where the weighting is according to w. A further simplification is due to the assumption that the network is organised in layers. This means that the set of nodes V is a union of non-empty, disjoint subsets, $V_0, V_1, V_2, ..., V_T$ for some $T \in \mathbb{N}$, such that every edge in E connects some node in V_{t-1} to another in V_T for some $t \in 1...T$. The number T of layers in the network excluding V_O is also called the "depth" of the network. The size of the network is the number of nodes $|V|$. The "width" of the network is $\max_t |V_t|$. Layers $V_1, ..., V_{T-1}$ are called the hidden layers, and V_0 and V_T are the input and the output layers respectively. The input layer contains $m + 1$ neurons, where $m \in \mathbb{N}$, is the dimensionality of the data. That is, as before, each data-point is some m-dimensional real-valued vector $x \in \mathbb{R}^m$ for a fixed $m \in \mathbb{N}$. The last neuron in V_0 is the "constant" neuron which always outputs 1. We let $v_{t,i}$ denote the i^{th} neuron of the t^{th} layer and by $o_{t,i}(x)$ the output of $v_{t,i}$ when the network is fed with the vector $x \in \mathbb{R}^m$ as input . This means that for any $i \in 1...m$ we have $o_{0,i}(x) = x_i$ and for $i = m + 1$ we have $o_{0,i}(x) = 1$.

Let us now consider calculation layer by layer. Suppose we have calculated the outputs of the neurons at layer t. Then, we can calculate the outputs of the neurons at layer $t + 1$ as follows. Fix some $v_{t+1,j} \in V_{t+1}$. Let $a_{t+1,j}(x)$ denote the input to $v_{t+1,j}$ when the network is fed with the input vector x. Then,

$$a_{t+1,k}(x) = \sum_{r:(v_{t,r},v_{t+1,j})\in E} w((v_{t,r}, v_{t+1,j})) \times o_{t,r}(x) \tag{1.1}$$

and

$$o_{t+1,j}(x) = \sigma(a_{t+1,j}(x))$$

That is, the input to $v_t{+}1, j$ is a weighted sum of the outputs of the neurons in V_t that are connected to $v_{t+1,j}$. The weights are calculated according to the weight function w, and the output of $v_{t+1,j}$ is simply the application of the activation function σ on it's input. Note that, the highlighted condition in (1.1), namely $r : (v_{t,r}, v_{t+1,j}) \in E$, enumerates over all edges between the nodes in the t^{th} layer and node j in layer $t + 1$.

To specify a neural network, we fix a graph (V, E) along with an activation function σ. Given a weight function $w : E \to \mathbb{R}$, a neural network predictor is defined as a function $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0|-1} \to \mathbb{R}^{|V_T|}$. Given the triplet (V,E, σ), the architecture of the neural network, a corresponding hypothesis class

$$\mathcal{H}_{V,E,\sigma} := \{w \text{ is a mapping from } E \text{ to } \mathbb{R}\}. \tag{1.2}$$

This means that the parameters which specify a hypothesis $h_{V,E,\sigma,w}$ in the hypothesis class $\mathcal{H}_{V,E,\sigma}$ are the weights over the edges of the network. Since E is a finite set, we can think of the weight function as a vector $w \in \mathbb{R}^{|E|}$. Therefore, the problem of learning using a given neural network hypothesis class $\mathcal{H}_{V,E,\sigma}$ amounts to tuning the weights over the edges of the network on the basis of training data.

2 Summary

- A neural network has an input layer, V_0, an output layer, V_T, and a number of hidden layers $V_1, ..., V_{T-1}$.
- Layers have nodes that are connected to other layers through edges. Each node, which corresponds to a single neuron, received as input the weighted sum of the outputs of the neurons connected to it's incoming edges.
- A neural network is modelled as a directed graph which a weight function w mapping the edges to real numbers. Each node is modelled as a scalar function σ which is called the activation function.
- There are 3 options for this which are the sign, threshold or sigmoid function.
- The network is organised in layers, so the set of nodes V is a union of non-empty, disjoint subsets $V_0, ..., V_T$, such that every edge in E connects some node in $V_{t-1} to V_t$.
- The depth of the network is the number of layers T.
- The size of the network is the number of nodes, $|V|$.
- The width of the network is the maximum size of a layer, $\max_t |V_t|$.

- The input layer contains m+1 neurons, where m is the dimensionality of the data. The last neuron in the input layer always outputs 1.
- Given we have calculated the output of the neurons at layer t, we can calculate the outputs of the neurons at layer t + 1 as the application of the activation function σ on it's input. Which is the weighted sum of the outputs of the neurons in V_t that are connected to $V_{t+1,j}$.
-

3 Support vector machines

3.1 The Mathematics

SVM is a classification algorithm that works by constructing an optimal separating hyperplane between two perfectly separated classes.

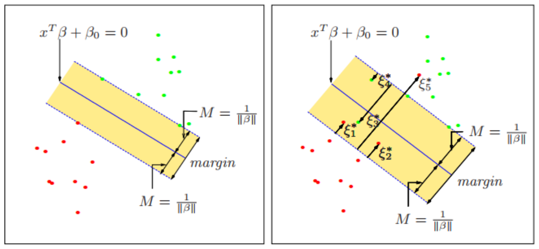


Figure 1: Support vector classifiers. The left panel showing the separable case where the decision boundary is the solid line, while broke lines bound the shaded maximal margin of width 2M. The right panel shows the nonseperable case.

If we have training data $(x_1, y_1), ..., (x_N, y_N)$ with x real values and y being -1 or 1, we can define a hyperplane by;

$$\{x : f(x) = x^T\beta + \beta_0 = 0\}, \tag{3.1}$$
$$\tag{3.2}$$

where β is a unit vector: $||\beta|| = 1$. A classification rule induced by f(x) is

$$G(x) = sign[x^T\beta + \beta_0] \tag{3.3}$$

f(x) in (1.1) gives the signed distance from a point x to the hyperplane $f(x) = x^T\beta + \beta_0 = 0$. As the classes are separable, we can find a function $f(x) = x^T\beta + \beta_0$ with $y_i f(x_i) > 0, \forall i$. Hence we can find the hyperplane that creates the biggest margin between training points for the classes -1 and 1. Therefore the optimisation problem is;

$$\max_{\beta,\beta_0,||\beta||=1} M, \text{ subject to } y_i(x_i^T\beta + \beta_0) \geq M, i = 1, ..., N, \tag{3.4}$$

Here the band is M units away from the hyperplane on either side and hence 2M wide, it is called the margin. The problem can be rephrased as;

$$\min_{\beta,\beta_0} ||\beta||, \text{ subject to } y_i(x_i^T\beta + \beta_0) \geq 1, i = 1, ..., N, \tag{3.5}$$

The usual support vector criterion is written as $M = 1/||\beta||$.

Suppose the classes overlap in feature space. In this case, we still maximise M but allow for some points to be on the wrong side of the margin. The slack variables are defined as $\xi = (\xi_1, ..., \xi_N)$. The constraint used is $y_i f(x) \geq M(1 - \xi_i)$, the proportional amount by which the prediction is on the wrong side of the margin. By bounding the sum $\sum \xi_i$, we bound the total proportional amount by which predictions fall on the wrong side of their margin. Miss-classifications occur when $\xi_i > 1$, so bounding the sum at a value K will bound the total number of training miss-classifications.

Therefore,

$$\text{we maximise } M = 1/||B|| \text{ by minimising } ||B||, \text{ subject to } \begin{cases} y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i, \forall i \\ \xi_i \geq 0, \sum \xi_i \leq constant. \end{cases} \tag{3.6}$$

This means that points well inside their boundary do not play a big role in shaping the boundary.

3.2 Computing the Support Vector Classifier

The problem above is a convex problem that can be solved using Lagrange Multipliers. (3.8) above can be expressed equivalently as:

$$\min_{\beta,\beta_0} \frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \xi_i \tag{3.7}$$

$$\text{subject to } \xi_i \geq 0, \; y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i \forall i, \tag{3.8}$$

The cost parameter C has replaced the constant from (3.8). If we rearrange the second constraint, we see that:

$$\xi_i \geq 1 - y_i(x_i^T\beta + \beta_0) \tag{3.9}$$

Therefore the minimum value of ξ_i is either 0 or $1 - y_i(x_i^T\beta + \beta_0)$, which we can now incorporate into (3.7) to gain;

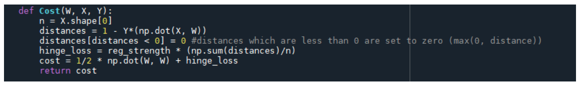
$$\frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \max(0, (1 - y_i(x_i^T\beta + \beta_0)) \tag{3.10}$$

β_0 can additionally be removed from the equation if we include an intercept column in the matrix X. $x_i^T\beta + \beta_0 = X^T W$ where $W = (\beta_0, \beta), X = (1, x_i)$, W can be achieved simply by including the intercept column of 1's in X. The final equation is known as the hinge loss.

3.3 Summary of the mathematics

- SVM is a supervised machine learning algorithm for binary classification.
- It's trained using a dataset with labels. The x_i is the n dimensional feature vector, with y_i the labels.
- SVM defines a hyperplane and creates separation between labels of -1 or +1 for the labels.
- The hyperplane is defined by $f(x) = x^T\beta + \beta_0 = 0$.
- The classification rule is $G(x) = sign[x^T\beta + \beta_0]$.
- The system finds the hyperplane by finding the optimal β weights, with intercept β_0. This is done by minimising the cost function using gradient descent. This has been achieved using stochastic gradient descent.
- This cost function is $\frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \max(0, (1 - y_i(x_i^T\beta))$.

4 The SVM Code



```
def Cost(W, X, Y):
    n = X.shape[0]
    distances = 1 - Y*(np.dot(X, W))
    distances[distances < 0] = 0 #distances which are less than 0 are set to zero (max(0, distance))
    hinge_loss = reg_strength * (np.sum(distances)/n)
    cost = 1/2 * np.dot(W, W) + hinge_loss
    return cost
```

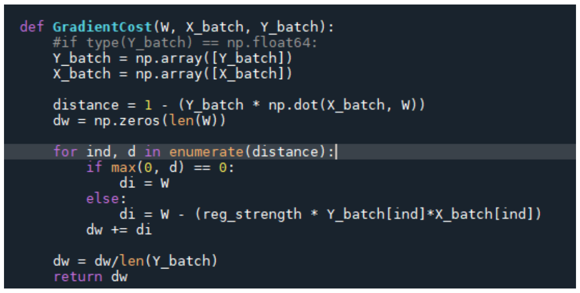
Figure 2: The Cost code

4.1 Cost function

- The Cost function first sets n as the length of the X vector.
- The cost function aims to calculate $\frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \max(0, (1 - y_i(x_i^T\beta + \beta_0))$. The init function includes a command to input a column of ones into x, therefore we are instead calculating $\frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \max(0, (1 - y_i(x_i^T\beta))$
- The function calculates the distances as $1 - y_i(x_i^T\beta)$.
- To then calculate $\max(0, (1 - y_i(x_i^T\beta))$, the function checks the vector distances for any values that are below 0, then sets these values to zero.
- $C\sum_{i=1}^{N} \max(0, (1 - y_i(x_i^T\beta))$ is then calculated by summing these values and multiplying by C/regression strength.
- The full equation is then calculated by adding the previous value to $\frac{1}{2}||\beta||^2$, where W is used to represent β.

4.2 Gradient of the cost function

If we differentiate the cost function with respect to the weights/β, we obtain;



```python
def GradientCost(W, X_batch, Y_batch):
    #if type(Y_batch) == np.float64:
    Y_batch = np.array([Y_batch])
    X_batch = np.array([X_batch])

    distance = 1 - (Y_batch * np.dot(X_batch, W))
    dw = np.zeros(len(W))

    for ind, d in enumerate(distance):
        if max(0, d) == 0:
            di = W
        else:
            di = W - (reg_strength * Y_batch[ind]*X_batch[ind])
        dw += di

    dw = dw/len(Y_batch)
    return dw
```

Figure 3: Gradient Cost Function

$$\frac{\delta}{\delta\beta}CF = \begin{cases} \beta, & \text{if } \max(0, 1 - y_i x_i^T) = 0 \\ \beta - \frac{C}{N}\sum_{i=1}^{N} y_i x_i, & \text{otherwise} \end{cases} \tag{4.1}$$

$$= \begin{cases} \frac{1}{N}\sum_{i=1}^{N} \beta, & \text{if } \max(0, 1 - y_i x_i^T) = 0 \\ \\ \frac{1}{N}\sum_{i=1}^{N} \beta - C y_i x_i, & \text{otherwise} \end{cases} \tag{4.2}$$

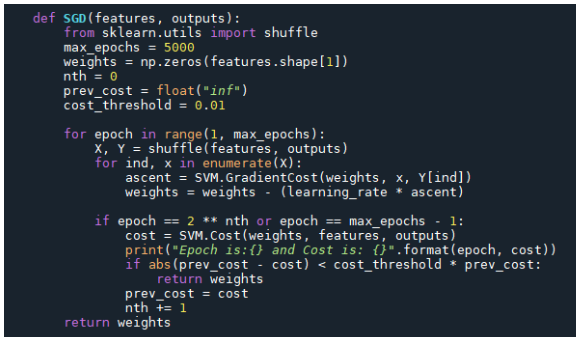
The function code performs as follows;

- The code first checks whether y is in a float64 form and converts it into a numpy array otherwise the following code would not work.
- The function calculates the distances as before, and sets dw as a vector of zeros the length of vector W.
- For the indices and d, in the numbers of distance (i.e. the length of W), the function checks for each distance whether it is bigger than zero, if it is not then it sets di as the vector W, if it is, then it sets di as the equation in the case (4.2)ii.
- dw is then set as the vector of the sum of all the di's.
- This vector is then divided by n to obtain the output.
- As you can see, what the function does is calculate (4.2) above.

4.3 Stochastic Gradient Descent

- Gradient descent means descending a slope to reach the lowest point on that surface.
- The algorithm first finds the slope of the objective function with respect to each parameter or feature. It computes the gradient of the function.
- It picks a random initial value for the parameters, then updates the gradient function by plugging in the parameter values.
- It then calculates the step sizes for each feature as: **step size = gradient * learning rate**.
- It then calculates new parameters as **new parameters = old parameters - step size** and continues this until the gradient is almost 0.

- The **learning rate** is a parameter that heavily influences the convergence of the algorithm. Larger rates make the algorithm take huge steps down the slope and it might jump across the minimum point and miss it, so it is good to have a low learning rate.
- Stochastic gradient descent comes in as normal gradient descent requires alot of computations so instead we use stochastic which means random.
- SGD selects data points at random from the whole dataset to calculate the derivatives, saving huge amounts of computations.
- A slight change on this is mini batch gradient descent which takes a small sample of data points instead of just one point.



```python
def SGD(features, outputs):
    from sklearn.utils import shuffle
    max_epochs = 5000
    weights = np.zeros(features.shape[1])
    nth = 0
    prev_cost = float("inf")
    cost_threshold = 0.01

    for epoch in range(1, max_epochs):
        X, Y = shuffle(features, outputs)
        for ind, x in enumerate(X):
            ascent = SVM.GradientCost(weights, x, Y[ind])
            weights = weights - (learning_rate * ascent)

        if epoch == 2 ** nth or epoch == max_epochs - 1:
            cost = SVM.Cost(weights, features, outputs)
            print("Epoch is:{} and Cost is: {}".format(epoch, cost))
            if abs(prev_cost - cost) < cost_threshold * prev_cost:
                return weights
            prev_cost = cost
            nth += 1
    return weights
```

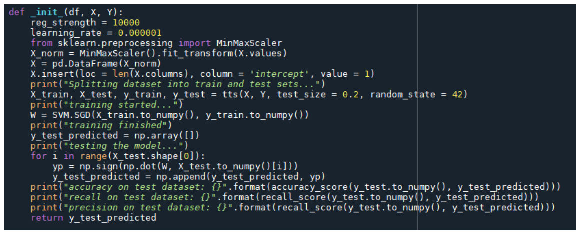
Figure 4: SGD function

The SGD code performs as follows;

- Sets a maximum to the epochs of 5000. Epoch stands for one cycle through the full training dataset, so the algorithm will perform a maximum 5000 cycles through the full training dataset. This is an upper bound to stop the algorithm going on forever.
- Sets the initial weights as an array of zeros the length of the features vector.
- Sets nth as zero to later refer to.
- Sets previous cost to a inf64 variable.
- Sets the cost threshold to 1%, to use as part of the stoppage criterion.
- For epoch in the range 1 to 5000, we shuffle the features and outputs, this is our random element described above.
- For the indices and x in enumerate (X), we calculate the gradient at the point, and then return the weights as the old parameters - step size as explained above.
- The second if statement is the stoppage criterion, this exists as each iteration costs time and extra computations that we dont need if the criterion is met. The loop stops if the current cost hasn't decreased much compared to the previous cost.

4.4 The initialise function: Tying everything together

The initialise function ties everything together to gain the model output from the dataset input. It



```python
def _init_(df, X, Y):
    reg_strength = 10000
    learning_rate = 0.000001
    from sklearn.preprocessing import MinMaxScaler
    X_norm = MinMaxScaler().fit_transform(X.values)
    X = pd.DataFrame(X_norm)
    X.insert(loc = len(X.columns), column = 'intercept', value = 1)
    print("Splitting dataset into train and test sets...")
    X_train, X_test, y_train, y_test = tts(X, Y, test_size = 0.2, random_state = 42)
    print("training started...")
    W = SVM.SGD(X_train.to_numpy(), y_train.to_numpy())
    print("training finished")
    y_test_predicted = np.array([])
    print("testing the model...")
    for i in range(X_test.shape[0]):
        yp = np.sign(np.dot(W, X_test.to_numpy()[i]))
        y_test_predicted = np.append(y_test_predicted, yp)
    print("accuracy on test dataset: {}".format(accuracy_score(y_test.to_numpy(), y_test_predicted)))
    print("recall on test dataset: {}".format(recall_score(y_test.to_numpy(), y_test_predicted)))
    print("precision on test dataset: {}".format(recall_score(y_test.to_numpy(), y_test_predicted)))
    return y_test_predicted
```

Figure 5: The initialise function.

performs as follows;

- Sets the reg strength and learning rate.
- Inputs a column of 1's into X for the intercept β_0 as explained earlier.
- Splits the data into train and test sets.
- Trains the model using the stochastic gradient descent function.
- Predicts the y values using the classification equation from (3.3) $sign[x^T\beta]$ (again recall the intercept is included in x) and returns the accuracy of the model.

5 Random Forest

5.1 The Mathematics

Random forests are built of the ideas of bagging;

- Bagging or bootstrap aggregation is a technique for reducing the variance of an estimated prediction function.
- Bagging works well for high variance, low bias procedures, such as trees.
- For regression, we fit the same regression tree many times to bootstrap sampled versions of the training data.
- For classification, a committee of trees each cast a vote for the predicted class.

Random forests are defined as;

- The aim of bagging is to average many noisy but approximately unbiased models to reduce the variance. Trees are ideal for bagging as they capture complex interaction structures in the data, and if grown sufficiently deep, have relatively low bias.
- Trees are notoriously noisy, so benefit from averaging.
- Since each tree generated by bagging id ID, the expectation of an average of B such trees is the same as the expectation of any one of them. So the bias of the bagged trees is the same as that of the individual trees.

- Therefore, the only improvements can be made by reducing the variance.
- The variance of the average is

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \tag{5.1}$$

- As B increases, the second term disappears, but the first remains, so the size of correlation of pairs of bagged trees limit the benefits of averaging.
- The idea of random forests is to improve the variance reduction by bagging by reducing the correlation between trees, without increasing the variance too much. This is achieved in the tree growing process through random selection of input variables.
- Specifically, when growing a tree on a bootstrapped dataset: *Before, each split, select* $m \leq P$ *of the input variables at random as candidates for splitting.*
- Typical values for m are $\sqrt{p}$ or as low as 1.
- After B such trees $\{T(x;\theta_b)\}_1^B$ are grown, the random forest (regression) predictor is

$$\hat{f}_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T(x;\theta_b). \tag{5.2}$$

- θ_b characterises the bth random forest tree in terms of split variables, cut-points at each node, and terminal- node values.
- Reducing m will reduce the correlation between any pairs of trees in the ensemble, thus, reducing the variance of the average.
- The default of m is $\sqrt{p}$ with $n_{min} = 1$ for classification and m is $p/3$ with $n_{min} = 5$ for regression. Note the best values depend on the problem and require hyperparameter optimisation.
- RF uses out of bag samples, meaning for each observation z_i, it constructs its random forest predictor by averaging only those trees corresponding to bootstrap samples in which z_i did not appear.
- In general, random forests build multiple decision trees and merge their predictions together to get a more accurate and stable prediction.

The generic random forest algorithm is;

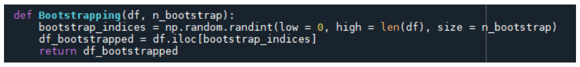
```
1. For b = 1 to B:
   (a) Draw a bootstrap sample $\mathbf{Z^*}$ of size N from the training data.
   (b) Grow a random- forest tree $T_b$ to the bootstrapped data, by recursively repeating the
       following steps for each terminal node of the tree, until the minimum node size $n_{min}$ is
       reached.
       i. Select m variables at random from the p variables.
       ii. Pick the best variable/split point among the m.
       iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.
```

To made a prediction at a new point x;
Regression: $\hat{f}^B_{rf}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$.
Classification: Let $\hat{C}_b(x)$ be the class prediction of the bth random forest tree. Then $\hat{C}^B_{rf}(x) =$ *majority vote* $\{\hat{C}_b(x)\}_1^B$.

---

6 The code

6.1 Bootstrap code



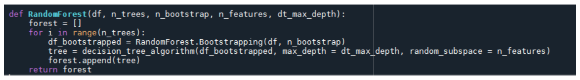
```
def Bootstrapping(df, n_bootstrap):
    bootstrap_indices = np.random.randint(low = 0, high = len(df), size = n_bootstrap)
    df_bootstrapped = df.iloc[bootstrap_indices]
    return df_bootstrapped
```

Figure 6: The bootstrap function.

The bootstrap code works as follows;

- Creates a vector called bootstrap indices, which is a set of random numbers between 0 and the length of the dataset, the size of which is specified by the user.
- The bootstrapped dataframe returns the dataframe which rows determined by the bootstrap indices just specified, therefore giving a random sample of the dataset.

6.2 Random Forest function



```
def RandomForest(df, n_trees, n_bootstrap, n_features, dt_max_depth):
    forest = []
    for i in range(n_trees):
        df_bootstrapped = RandomForest.Bootstrapping(df, n_bootstrap)
        tree = decision_tree_algorithm(df_bootstrapped, max_depth = dt_max_depth, random_subspace = n_features)
        forest.append(tree)
    return forest
```

Figure 7: The initialise function.

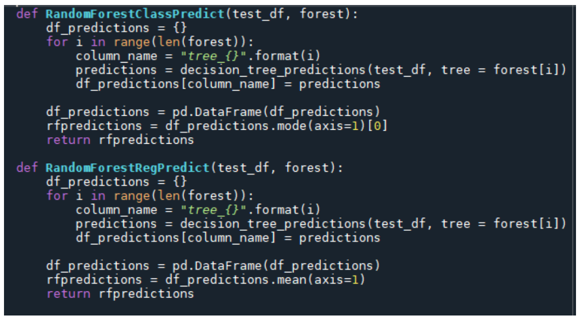
The random forest function works as follows;

- Creates a random sample of the dataset using the bootstrap function.
- Creates multiple decision tree using a specified decision tree algorithm, using the random sample, with the max depth of the tree specified by the user, as well as the random subspace.

6.3 Predict function

The predict function depends on whether the problem is regression or classification, the only difference is that regression uses the mean and classification uses the mode. It works as follows;

- Takes each tree in the forest, and uses the predict function from the decision trees algorithm to make a prediction on the test dataframe based on the given tree as the time.
- The function now has a dataframe of predictions based on multiple different trees, it returns the mode of the predictions for classification, and the mean for regression.



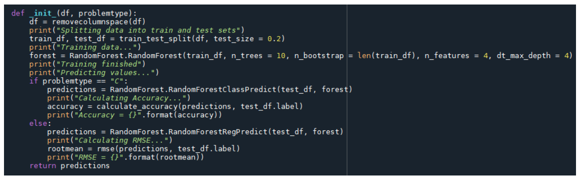
```python
def RandomForestClassPredict(test_df, forest):
    df_predictions = {}
    for i in range(len(forest)):
        column_name = "tree_{}".format(i)
        predictions = decision_tree_predictions(test_df, tree = forest[i])
        df_predictions[column_name] = predictions

    df_predictions = pd.DataFrame(df_predictions)
    rfpredictions = df_predictions.mode(axis=1)[0]
    return rfpredictions

def RandomForestRegPredict(test_df, forest):
    df_predictions = {}
    for i in range(len(forest)):
        column_name = "tree_{}".format(i)
        predictions = decision_tree_predictions(test_df, tree = forest[i])
        df_predictions[column_name] = predictions

    df_predictions = pd.DataFrame(df_predictions)
    rfpredictions = df_predictions.mean(axis=1)
    return rfpredictions
```

Figure 8: The initialise function.

6.4 Initialise function

The initialise function performs as follows;

- Inputs: df - the dataframe/dataset, problemtype - either "C" for classification or "R" for regression.
- The function first removes any feature name white space as this results in an error with the predict function, as this splits the tree decision based on white space, any white space in the feature name would give us more splits than we need and cause an error.
- The function then splits the dataset into test and train sets.
- The function build the random forest model using the function listed earlier.
- The function then performs different based on problem type.
- For classification, the function predicts using the classification function problem, then calculates and returns the accuracy of the predictions.
- For regression, the function does the same using the regression predict function, then calculates and returns the RMSE of the predictions.



```python
def _init_(df, problemtype):
    df = removecolumnspace(df)
    print("Splitting data into train and test sets")
    train_df, test_df = train_test_split(df, test_size = 0.2)
    print("Training data...")
    forest = RandomForest.RandomForest(train_df, n_trees = 10, n_bootstrap = len(train_df), n_features = 4, dt_max_depth = 4)
    print("Training finished")
    print("Predicting values...")
    if problemtype == "C":
        predictions = RandomForest.RandomForestClassPredict(test_df, forest)
        print("Calculating Accuracy...")
        accuracy = calculate_accuracy(predictions, test_df.label)
        print("Accuracy = {}".format(accuracy))
    else:
        predictions = RandomForest.RandomForestRegPredict(test_df, forest)
        print("Calculating RMSE...")
        rootmean = rmse(predictions, test_df.label)
        print("RMSE = {}".format(rootmean))
    return predictions
```

Figure 9: The initialise function.

7 K nearest neighbours

Given a query point x_0, we find the k training points $x_{(R)}, r = 1, ..., k$ closest in distance to x_0, then classify using the majority vote among the k neighbors.

If we have pairs of data $(X_1, Y_1), ..., (X_n, Y_n)$ where X is the training data and Y is the label, given x_0, we reorder the data based on the euclidean distances from the query point, that is $||X_1 - x|| \leq ... \leq ||X_n - x||$.

Algorithm In classification, k is a user defined constant, and an unlabeled vector is classificed by assigning the label which is most frequent among the k training samples.

The classification accuracy can be imporved by using a distance metric that is learned with specialised algorithms such as large margin nearest neighbor, or neighbourhood component anlysis.

Parameters: k, the initial constant, can be selected using hyperparameter optimisation.

Regression Compute the euclidean distances from the query example. Order the labeled examples by increasing distance. Find a heuristically optimal number k of nearest neighbors, based on RMSE. Calculate an inverse distance weighted average with the k nearest multivariate neighbours.

Psuedo code

- Initialise the value of k, this will later be optimised as a hyperparameter.
- Calculate the euclidean distance between the test data and each row of the training data.
- Sort the distances in ascending order based on distance values.
- Get the top k rows for the sorted distances.
- Get the most frequent class of these rows.
- Return this class as the predicted class.

8 Gradient Boosting

9 Model Selection: AIC

AIC $= 2k - 2ln(\hat{L})$.
K is the number of parameters whereas $\hat{L}$ is the maximum value of the likelihood function. It works for model selection as increasing the number of parameters in a model always improves the goodness of fit, but there is a penalty that comes with it by measuring overfit, which is the loss term.

10 Hyperparameter optimisation

In short, **Bayesian optimisation** builds a probability model of the objective function and uses it to select the most promising hyperparameters to evaluate the true objective function.

It works by building a probability model of the objective function that maps input values to a probability of a loss p. The model, called the response surface, is easier to optimise than the actual objective function. Bayesian methods select the next values to evaluate by applying a criteria such as expected improvement to the response. The concept is to limit evals of the objective function by spending more time choosing the next values to try.

Bayesian reasoning means updating a model based on new evidence, and, with each eval, the surrogate is re calculated to incorporate the latest information. The objective function is constructed using either Gaussian Process, Random forest regression or Tree parzen estimator.

Hyperparameters are represented in equation form as;

$$x* = \arg\min_{x} \in \chi f(x)$$

$f(x)$ represents an objective score to minimise, such as RMSE or error rate, which is evaluated on the validation set. $x*$ is the set of hyperparameters that yields the lowest value of the score, and x can take on any value in the domain X.

The problem with this is that evaluating the objective function is expensive as it take time to continuously train models with different hyperparameters.

Bayesian optimisation keeps track of past results, which is used to form a probabilistic model which maps hyperparameters to a probability of a score on the objective function.

$$P(score|hyperparameters)$$

This is called the surrogate for the objective and is represented as $p(y|x)$. The surrogate is much easier to optimise than the objective and Bayesian methods work by finding the next set of hyperparameters to evaluate on the actual objective function by selecting hyperparameters that perform best on the surrogate function.

1. Build a surrogate probability model of the objective function.
2. Find the hyperparameters that perform best on the surrogate.
3. Apply these hyperparameters to the true objective function.
4. Update the surrogate model incorporating the new results.
5. Repeat steps 2-4 until max iterations or time is reached.

Bayesian optimisation are efficient as they choose the next hyperparameters in an informed manner. They spend a little more time selecting the next hyperparameters in order to make fewer calls to the objective function.

10.1 Sequential model based optimisation

SMBO runs trials one after another, each time trying better hyperparameters by applying bayesian reasoning and updating a probability model/surrogate.

There are five aspects;

1. A domain of hyperparameters over which to search.
2. An objective function which takes in hyperparameters and outputs a score that we wish to either minimise or maximise.
3. The surrogate model of the objective function.
4. The selection function/criteria for evaluating which hyperparameters to choose next from the surrogate model.

5. A history consisting of (score, hyperparameter) pairs used by the algorithm to update the surrogate model.

Common choices for surrogate model are Gaussian processes, Random forest regression and tree parzen estimators. Step 4 commonly uses expected improvement.

Domain
We first have a domain of possible values for the hyperparameters. For example KNN might have possible values of k, whereas random forest would have possible values for number of estimators, max depth and min samples leaf. This is for grid or random search.

For a model based approach, the domain consists of probability distributions. This lets us encode domain knowledge into the search process by placing greater probability in regions where we think the true best hyperparameters lie.

Objective function
This function takes in hyperparameters and outputs a single real valued score that we either want to minimise or maximise. For example, in random forest regression, we would have hyperparameters and the single real valued score would be Root Mean Squared Error that we wish to minimise. This includes going through and building the model with hyperparameters and then calculating the rmse, which is very expensive. To reduce the amount of times this happens, a surrogate probability model is used.

Surrogate function
This is the probability representation of the objective function built using previous evaluations. This is called a response surface as it is a high dimensional mapping of hyperparameters to the probability of a score on the objective function. This function often takes the form of gaussian processes, random forest regression or tree parzen estimator.

Selection function
This is the criteria by which the next set of hyperparameters are chosen from the surrogate. It is commonly Expected improvement;

$$EI_{y*}(x) = \int_{-\infty}^{y*} (y^* - y) p(y|x) dy$$

$y*$ is a threshold value of the objective function, x is the proposed set of hyperparameters, y is the actual value of the objective function using hyperparameters x, and $p(y|x)$ is the surrogate probability model expressing the probability of y given x. The aim is to maximise Expected improvement with respect to x. This means finding the best hyperparameters under the surrogate function $p(y|x)$. If this is zero whenever $y < y*$, then the hyperparameters x are not expected to yield any improvement. If the integral is positive, then it means the hyperparameters are expected to yield a better result than the threshold value.

TPE
TPE builds a model by applying Bayes rule. Using;

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

where,

$$p(x|y) = \begin{cases} \ell(x) & \text{if } y < y* \\ g(x) & \text{if } y \geq y* \end{cases} \tag{10.1}$$

This means we have two different distributions for the hyperparameters, one where the value of the objective function is less than the threshold, and one where the value is greater than the threshold.

Now we construct two probability distributions for the number of estimators, one using the estimators that yielded values under the threshold, and one using the estimators that yielded the values above the threshold.

We want to draw values of x from $\ell(x)$ because this distribution is based only on values of x that yielded lower scores than the threshold, and this is shown by performing substitutions on the EI equation;

$$EI_{y*}(x) \approx (\gamma + \frac{g(x)}{\ell(x)}(1-\gamma))^{-1}$$

This says that expected improvement is proportional to hte ratio $\ell(x)/g(x)$ and so to maximise EI, we maximise this ratio.

TPE works by drawing sample hyperparameters from $\ell(x)$, evaluating them in terms of the ratio, then returning the set that yields the highest value under this ratio, corresponding to the greatest expected improvement. These hyperparameters are then evaluated on the objective function, if the surrogate function is correct, then these hyperparameters should yield a better value when evaluated.

History The algorithm stores the scores and hyperparameters to form a history record. The algorithm builds $\ell(x), g(x)$ from the history to come up with a probability model of the objective function that improves with each iteration.

In summary, this method is better than others because it gains better hyperparameters, and spends less time finding the next set of hyperparameters than other algorithms spend evaluating the objective function.

Potential libraries

- Spearmint, MOE: Gaussian Process
- Hyperopt: TPE
- SMAC: Random Forest regression