***Architecture (UML):***

|  |
| --- |
| Dataset |
| +df: pandas.DataFrame  +description: str  + attributes: dict() holding all the different possible values for each attribute (metadata) |
| +\_\_init\_\_(): Class constructor  + traintestsets(self, train\_size: float , test\_size: float): returns list of equally partitioned sets of main dataframe to preform k-fold cross\_validation  + decode(self): decodes all string columns from byte string to python str  + fill\_missing\_values(self, byClass: bool ): fills missing values for dataframe, has option to take class into consideration for mean and mode, or not  + map\_discretization(self, discretized\_df): maps discretized\_df bins for numerical columns to numeric columns in self.DataFrame  + entropy(self, sorted\_values: np.array) : calculates entropy of given sorted array and returns value  + entropy\_discretization\_helper(self, attribute, sorted\_values, threshold) : recursively calculates entropies, until meeting given threshold and returns minimum entropy breakpoints for given sorted\_values  + discretize( self, attribute, method, threshold) : discretizes continuous data based on given method. Entropy-based discretization is referred by method = ‘entropy’, and uses threshold as stopping criteria. Method = ‘binning’ bins values into 10 equally sized bin  + normalize(self, attribute, method): normalizes the attribute passed to it, throws error if not numerical, also has ‘z-score’ set to default method  + encode(self): encodes the dataset by hot one encoding each categorical attribute |

|  |
| --- |
| Evaluator |
| + actual: list() of actual class labels for given test set  + predicted: list() of predicted class labels for given test set  + n: number of evaluated observations  + correct: number of correctly evaluated observations  + class\_labels: list() of possible class\_labels  + confusion\_matrices: dict() of confusion matrix for each class label |
| + \_\_init\_\_(): Class constructor  + init(): initializes class variables if not passed as arguments to constructor  + precision(): calculates and returns the precision given a confusion matrix  + recall(): calculates and returns the recall given a confusion matrix  + f1(): calculates and returns f1 score given confusion\_matrix  + micro\_f1\_score(): calculates and returns micro\_f1\_score for given actual and predicted values  + macro\_f1\_score(): calculates and returns macro f1 score for given actual and predicted values  + micro\_precision(): calculates and returns micro precision given actual and predicted values  + macro\_precision(): calculates and returns macro precision given actual and predicted values  + micro\_recall(): calculates and returns micro recall given actual and predicted values  + macro\_recall(): calculates and returns macro recall given actual and predicted values  + accuracy(): calculates and returns accuracy of actual vs predicted values  + add\_predicition(self, actual, predicted): populates appropriate confusion matrices given a predicted and actual value of an observation  + add(self, actual, predicted): populates confusion matrices given vectors of actual and predicted values of an entire dataset |

|  |
| --- |
| BackProp |
| + hiddenLayerCount: integer representing number of hidden layers  + hiddenLayerSizes: List of sizes for different hidden Layers  + learningRate: learning rate for neural network  + hiddenLayers: list of hiddenLayer objects holding all hidden layers  + outputLayerWeights: numpy array holding weights going into output layer  + minimumError: minimum error for convergence  + maxEpochs: maximum epochs for convergence |
| + \_\_init\_\_(): Class constructor  + init\_weights(self, inputLayerNodes, outputLayerNodes): randomly initiailizes all the weights across the network  + feed\_forward (self, row, outputLayerNodes): feeds an example(row) through the network and returns output  + calculate\_error\_vectors(self, y, output): Calculates error vectors and returns them as deltao for output layer, and a list of numpy arrays (deltah) for each hidden layer  + adjust\_weights(self, row, deltah, deltao): adjusts weights across the network given error vectors  + train(self, ds: DataSet): feeds examples through networks until minimumError is reached or epochs have elapsed  + classify(self, ds: DataSet ): classifies dataset based on trained model and returns Evaluator object |

|  |
| --- |
| Hidden Layer |
| + size: number of nodes in hidden layer  + previousLayerSize: number of nodes in previous layer  + nodes: numpy array holding values of each node in layer  + weights: numpy matrix holding all weights of edges coming into layers’ nodes (on left side of layer) |
| + \_\_init\_\_(): Class constructor  + init\_weights (): initializes the weights for the layer |

Also implemented a `core.py` script to run 5-fold cross validation, pre-processing, modelling, and evaluation for this model

***Considerations:***

*Pre-processing:*

Filling missing values:

* Filled missing values for training set according to mean and mode (numeric or categorical respectively) of given class label
* Filled missing values for testing set based on overall mean and mode (numeric or categorical respectively) for given observations

*Back Propagating Neural Network:*

* Instead of having a distinct object to hold the biases, I attached the bias at the end of each layer, except the output. In this way, I would have an edge to store the bias for each node in the weights coming into that layer. Visual representation below:

Bias

The weight of the incoming edge from the last node in the previous layer is the bias.

* Created BackProp object to implement relevant functions and store relevant data for out network
  + Abstracted hidden layers as hidden layer objects and stored them in list in BackProp object so that this model could be used with more than one hidden layer, and have varying hidden layer sizes too.
* Training
  + Encoded dataset at beginning of training using dataset metadata (in .encode() method)
  + Use dictionaries to iterate through data instead of iterating through pandas DataFrame row by row in an attempt to optimize performance.
  + Abstracted the functions necessary to train into:
    - Feed-forward
    - Calculating error
    - Adjusting weights
  + This would also help me classify, since I just need to feed forward through network
  + Stopped training when reached the max number of epochs or when error rate (SSE) was lower than threshold
* Classifying
  + Encoded dataset at beginning of classifying using dataset metadata ( in encode() method )