Artificial Intelligence Project 3: Design Document

Group 11: Lisa Peters, Janette Rounds & Monica Thornton

1. Description of the Problem

Classification tasks are a common problem in machine learning, and as such, there are a number of algorithms that can be employed to classify data. The goal of classification is to construct a classifier that divides input data into a discrete number of classes described by a set of attributes, allowing the user to both learn from the input data and generalize about unseen inputs. In the software we are developing for this project, we will implement four well-established machine learning algorithms: k-nearest neighbor (k-NN), naïve Bayes, tree augmented naïve Bayes (TAN), and Iterative Dichotomiser 3 (ID3). These algorithms take vastly different approaches to classification, and to test these algorithms we will look at their ability to classify data in five different data sets. These data sets all come from the UCI Machine Learning repository, and vary in the number of data points, the number of attributes, the type of data, and the number of classes (Lichman, 2013). Details regarding the implementation of the learning algorithms can be found in Section 3 of this document, and our plan to evaluate the performance of these algorithms will be outlined in Section 4.

2. Software Architecture

A model of our software is provided in Figure 1. This requirements for this software were simpler than the requirements for the other two projects. We need to handle data imputation and discretization, and implement four classification algorithms. For this project, we chose to use the State software design pattern (Freeman, Robson, Bates, & Sierra, 2004). In the State pattern, there is a Context class and State interface. Several sub-classes inherit from the State interface. Although this may appear to be similar to other patterns we have previously used, there is a key difference in that the Context class maintains a pointer to the current State sub-class, which will change when the internal state of the State interface changes. Thus the State class appears to change its class.

In our implementation, the `RunModels` class corresponds to the Context class, and contains a pointer to an instance of the `Algorithm` class. When we switch to a different classification algorithm in our experiment, the pointer will be changed to the new subclass. The abstract `Algorithm` class corresponds to the State class. For software testing purposes, we have chosen to make methods such as *train*() and *test*() concrete methods in the `Algorithm` abstract class. These methods will be overridden in the concrete subclasses, but the initial methods will simply assign a class label to a point at random. This will allow us to test other parts of the software without needing a classification algorithm to be implemented. We can also use the "Random" algorithm as a baseline for our other algorithms when it comes to conducting experiments. `RunModels`, in addition to acting as the Context, will also call our `DataImputer` and `DataDiscretizer` classes, in order to prepare the data for classification experiments. The `ValueDiff` class holds the methods for the difference metric, while the

`Parser` class reads in and preprocesses the dataset. The transitions between States in the State pattern (or between algorithms in our implementation) can either be defined in the Context class or by the State classes themselves. We have chosen to define the transitions in the Context class because we will have no need to add additional classification algorithms to this software, and we can write a single method to handle transitions rather than one for each algorithm.

Two of our classification algorithms use a tree structure, the `DecisionTree` algorithm and the `TreeAugmented NB` algorithm. An expansion of the `Tree` data structure from Figure 1 is shown in Figure 2. We expect that the two algorithms will have different requirements for the tree, but there are some methods common to all trees. Therefore, we can write some methods in the `Tree` class and the abstract `TreeNode` class, leaving the unique operations and information to be written in the `DecisionTreeNode` and `BayesTreeNode` classes.

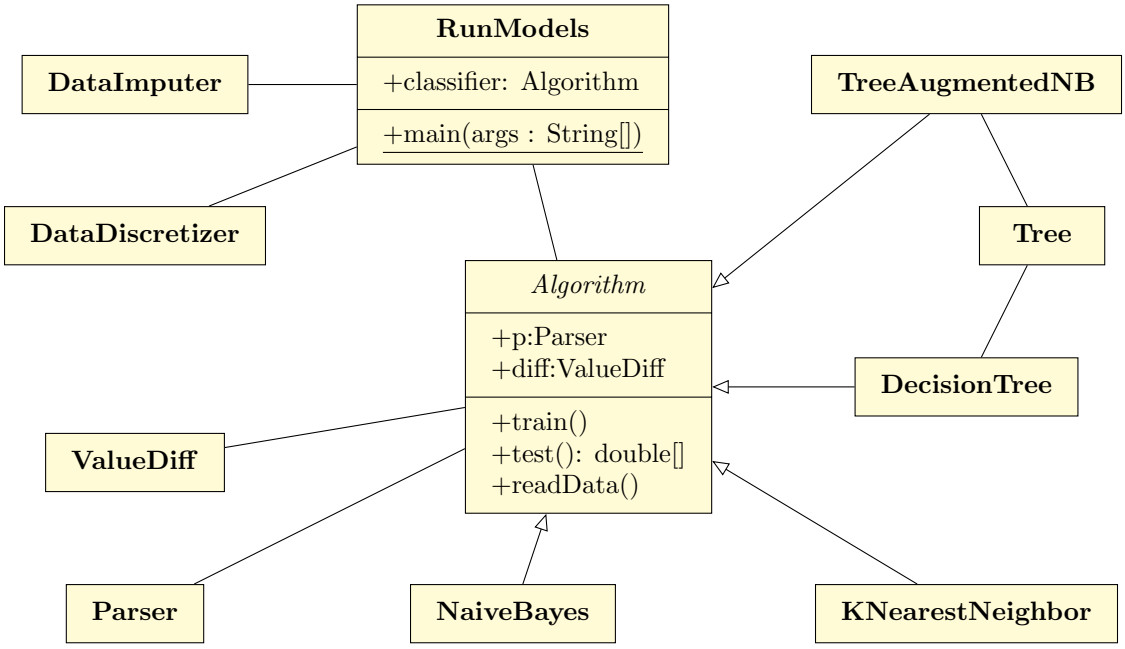

Figure 1: UML diagram that outlines the architecture of the described software.

3. Design Decisions

In the following subsections, we outline a number of the key design decisions we have made when implementing our solution. The decisions more directly related to our experimental design are presented in Section 4 of this document. Additionally, although not depicted in Figure 1, our software will also include classes for running experiments and printing results. Time permitting we will also include unit tests for all of the major functions, to ensure that the software continues to perform as expected.

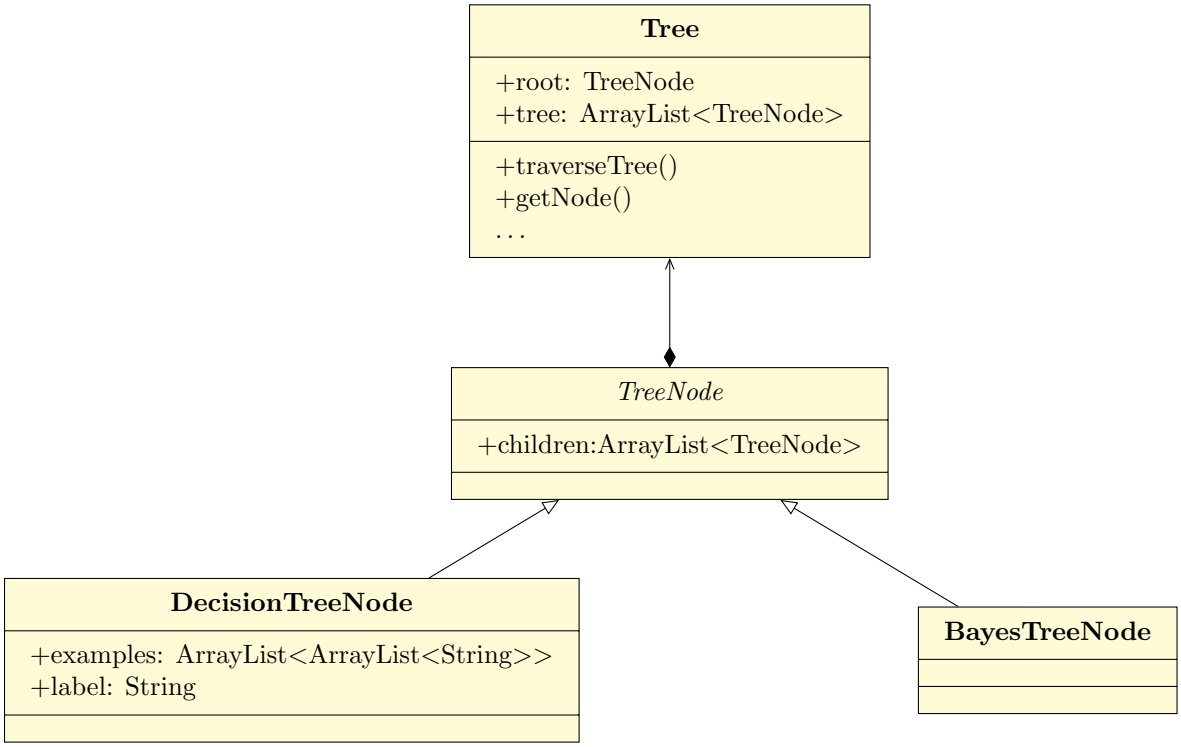

Figure 2: UML diagram that outlines the architecture of the Tree data structure

3.1 Data Imputation

It is not uncommon for data sets to have missing attribute values, and this missing information can be problematic when trying to use machine learning algorithms to classify data. With respect to classification problems, there are two main methods of dealing with missing attribute values, the first strategy involves removing the associated data points before classification, and the second strategy employs data imputation as a method to replace the missing attribute values with reasonable values. There are a number of common data imputation strategies, including random imputation based on observed data for that feature, or "fractioning" the examples based on the prior probability of each of the values as determined by the remaining examples in the data set (Sheppard, 2016). The latter is the approach we will take in this work, as we feel it does the best job retaining the relationship between the attributes and the corresponding class label.

Of the five datasets we will be using to test the implemented learning algorithms, only two of them (Breast Cancer and Vote), have attribute values corresponding to '?' (UCI, 2013a, 2013b). Generally, this is used to indicate missing attribute values in a dataset, but with the Vote data, the '?' actually indicates an unknown disposition - meaning that the representative voted present, voted present to avoid a conflict of interest, or did not vote or make their position known. If we were to remove the entries with one or more '?' attributes, that would reduce the size of the dataset from 435 data points to 232, and this smaller sized dataset would likely compromise the effectiveness of our algorithms. To that end, we will explore two approaches to dealing with the '?' valued attributes in the Vote

dataset. The first will be to impute the data, using the previously mentioned fractioning approach, and the second approach involves treating the '?' as a ternary value as argued for in the work of (Maurus & Plant, 2014).

The only dataset with truly missing information is the Breast Cancer data, and in this case there are only 16 instances of missing attribute values. Given that the Breast Cancer data set is comprised of 699 data points and the majority of the missing attribute values belong to the majority class, a strong argument can be made that removing these would probably have little impact on the effectiveness of our learning algorithms. As before, we will examine two approaches to deal with this missing data, the first will involve the previously mentioned data imputation approach, and the second will involve removing the data points with missing attribute values. Further details about our plan to test the proposed missing attribute value strategies is provided in Section 4 of this work.

3.2 Data Discretizer

Three of our datasets contain continuous (real-valued) data. This presents a problem in that we would like to use the same distance metric for all features in all datasets. Our distance is the value difference distance metric which takes only categorical data. In order to convert the real-valued data to discretized data, we have several options. We could simply "snap" the real valued number to the nearest integer. However, if there are features with a range between 0 and 1, snapping to the nearest integer would eliminate a large amount of variability within that feature. Another option would be to divide up the full range of the feature into equal sized bins. We considered this option, but we decided against it because if there were outliers, this would tend to skew the bins toward the outliers, leading to many bins with few or no examples, and a few bins with many examples. We chose to use the equal frequency discretization method (Clarke & Barton, 2000). This method divides the range of the feature into bins based on the number of examples from the training set in each bin. In other words $r\%$ of the dataset is assigned to each bin, and there are $1/r$ bins where r is a real-valued number between 0 and 1. This method is not perfect however. Two examples that are close together may wind up being assigned to different bins. Additionally, two examples that are far apart may wind up being assigned to the same bin. One way we can minimize this drawback is to tune r very carefully. Descriptions of our tuning process can be found in Section 4.

3.3 ID3

The ID3 algorithm (or Iterative Dichotomizer 3)(Russell & Norvig, 2003) constructs a classification tree that can be binary, non-binary or some combination of the two. The classification tree takes in an example, tests that example on a number of feature values and returns a class label. We construct the tree by finding ways to split the data set. To find each split, we maximize a measure called Information Gain, which is the expected reduction in entropy. When we create a split, we create a new node for each value in the split (e.g. Values of True/False for a binary splits, 1/2/3 for integer splits, etc.). We keep creating splits until every example associated with the node has the same class label. The node then becomes a leaf node and is assigned the class label of the examples. When all of

the terminating nodes of the tree have been assigned class labels, we are done growing the tree.

Overfitting is a common problem with decision trees. Overfitting happens when the tree memorizes each individual example rather than creating a general model for the problem. There are a number of heuristics we could use to prevent overfitting. We will use the reduced error pruning method for this project. Once the tree is fully expanded, we will conduct reduced error pruning, which will use a pruning set to determine whether a particular split produces more or less classification error. If the test fails (i.e. the split produces more classification error on the pruning set), the split and the child nodes are removed, and the class labels for the examples are counted. The class label with the greatest number of examples becomes the class label for the new leaf node. Splits only occur on internal nodes and only leaf nodes will have an associated class label.

Another possible heuristic for preventing overfitting is early stopping. This method stops generating new nodes when there is no "good" attribute on which to split. This heuristic has lower computation times as it is not producing splits and then pruning them away. However, there are situations where combinations of splits provide more Information Gain than Information Gain for each split individually would suggest. This is why we chose the reduced error pruning method.

3.4 k-Nearest Neighbor

The k-nearest neighbor (k-NN) algorithm is a non-parametric lazy learning method that provides a conceptually simple approach to classification. When the algorithm encounters an unclassified point x_q, it assigns that point the class that occurs most frequently among the k closest (as defined by some distance metric) neighbors of x_q. This method was proposed for classification problems in which the underlying joint distribution of the observation x and the category θ are unknown, and it operates on the assumption that observations that are close to each other will have the same class label, or will have similar posterior probability distributions on their respective classes (Cover & Hart, 1967).

Although k-NN is perhaps the conceptually simplest algorithm we will implement in this work, there are some pitfalls associated with this algorithm that we hope to avoid or mitigate via our design decisions. Given that the k-NN algorithm relies on finding the k points that are closest to our query point x_q, it is not surprising that the choice of distance metric is important to the performance of the algorithm. In our implementation of k-NN, we will be using the Value Difference Metric (VDM), which was designed as a distance function for nominal attributes, and was made to measure the difference between predictive features by taking the similarity (or dissimilarity) of values into account (Stanfill & Waltz, 1986). Additionally, an appropriate choice of k is very important to the performance of the algorithm, and we will do tuning experiments to pick a value of k that is not under or overfitting the data (Russell & Norvig, 2003). Our implementation of the k-NN algorithm will be based on the one provided by Fukunage and Narendra, which aims to mitigate the required number of expensive distance computations through a branch and bound approach that eliminates some points from consideration without explicitly computing the distances (Fukunage & Narendra, 1975). As previously mentioned, our data sets vary in size and dimensionality, and this approach is attractive because it even with these relatively small

and low-dimensional data sets we want to be sensitive to the curse of dimensionality. The branch and bound method we will implement uses a k-d tree to find nearest neighbors, and although some of the data sets do not have the 2^n relationship between examples and dimensions recommended by Russell and Norvig, the branch and bound approach should result in computational savings that are an improvement over a linear scan of the entire data set (Russell & Norvig, 2003).

3.5 Naïve Bayes

As a classifier, naïve Bayes stands out for its simplicity in implementation, speed, its efficiency for many text-based classifications, and being scalable to large data sets (Koller & Friedman, 2009). The most important part of naïve Bayes is an assumption of independence between features of a class. For example, in trying to classify fruit, the algorithm assumed the hue of a piece will have no relation to its firmness and ripeness, despite this not being true in the real world. This allows for the use of Bayes Theorem to find probabilities between features and classes, given multiple features and without resorting to complicated mathematics. Using the assumption of independence and Bayes Theorem, we will effectively uncouple pieces of probabilistic evidence. For a provided data set, all pieces of Bayes Theorem will be calculated for each feature, such as the total evidence for a feature, prior probability of outcome and so on. This gives us probabilities of an item belonging to a class given certain features. When presented with an item from the testing set, the algorithm simply chooses the highest probability for a class given the item's features.

In addition to the drawback that features might not be independent, as in the above example, the other significant issue of naïve Bayes is known as Zero Frequency. Due to the fact that the algorithm requires multiplication between probabilities of features, if there is a null or zero value, the algorithm will fail. To solve this issue we will use a simplistic version of Laplace smoothing, adding one to each count to eliminate null values, as discussed in the textbook (Russell & Norvig, 2003).

3.6 TAN

While näive Bayes often does well in classification problems, there are many cases where the conditional independence assumption can adversely impact performance (Sheppard, 2016). To address this issue, tree augmented naïve Bayes (TAN) was introduced to help relax this strong assumption while still retaining much of the computational simplicity and robustness of the original naïve Bayes algorithm and exhibiting greater classification accuracy (Friedman, Geiger, & Goldszmidt, 1997). TAN is capable of approximating the interactions between attributes through the imposition of a tree representation on the naïve Bayes structure, where "highly dependent" features share an edge, provided that no node has more than two parents. What features are highly dependent is determined by the conditional mutual information measure, which helps to determine what attributes are correlated in the training data. In essence, the TAN algorithm builds a complete undirected graph with all feature nodes connected, the edges are weighted using conditional mutual information, and a maximum spanning tree is found. The undirected tree then becomes a directed tree by arbitrarily designating a root node, and directing edges downward. With respect to classification, the probability tables for each attribute are calculated in much the

same manner as with naïve Bayes, however, now two conditions are applied (to account for the parent attribute).

With respect to design decisions for our implementation of TAN, we are utilizing the approach provided by Friedman *et al.* (Friedman et al., 1997). To build and use the tree we will utilize the `BayesTreeNode` class depicted in Figure 2. Additionally, depending on performance of the algorithm on the selected data sets, we may choose to introduce a smoothing factor (as recommended by Friedman *et al.*) to further increase accuracy.

4. Experimental Design

Cross validation is way of determining whether a model will generalize beyond a training data set. This is one way to determine if overfitting is happening in classification. There are several types of cross validation. One type of cross validation is 10-fold cross validation. This method divides the dataset into 10 equal sized, miniature datasets. The algorithm is run 10 times. Each time the algorithm is run, the algorithm is trained on 9 of the 10 miniature datasets, and tested on the final miniature dataset. The test miniature dataset is rotated each run, so that each test is run on a different part of the dataset.

Another cross validation method is 5×2 cross validation. This method splits the dataset into 5 miniature datasets.The procedure is the same as for 10-fold cross validation, except that there are 5 miniature datasets instead of 10. The primary difference is that the algorithm is run on each train/test split twice. The 5×2 cross validation method has been shown to more directly assess variation due to the choice of training set than the 10-fold cross validation method (Dietterich, 1998). It has also been shown to be slower than other cross validation methods, but we prioritized better variation assessment in this work.

In order to produce our train/test splits, we will use a stratified method that stochastically assigns a specific proportion of the examples with a given class label to the test set, with the remainder of the examples going to the training set. In this way, the test set will more closely mimic the distribution of the training set.

Some of our algorithms require a validation or pruning set. In order to maintain the viability of our tests, examples from the pruning/validation sets will not be drawn from the test set. Only examples from the training set will be used in the validation set. Since the validation set is used as part of the training process, we still consider these examples to be a part of the training set even though they are slated for a different process. For algorithms that require a pruning/validation set, we will use 10% of the full dataset as the size of our validation set. The same method used to select the test set will be used to select the validation set.

We will conduct tuning experiments on our algorithms to find a good value for our parameters. We have identified two parameters that need tuning: r in our discretization function and k in the k-NN algorithm. We will tune k by running k-NN on the Vote dataset with a number of values for k. The k with the lowest error for a test set will be used as the k for all other datasets.

Tuning r will be more challenging than tuning k because it needs to be tuned separately for each dataset. However, we have two possible methods for selecting r. The first is to simply use $r = 2 * c$ where c is the number of class labels in the dataset. An alternative method would be to increment r and see if the classification accuracy improves. If it does

not improve, we will decrement r and use that value for the r for the dataset. We will test both of these methods on the Soybean and Glass datasets and select whichever method gives us the highest average classification accuracy (UCI, 2013d, 2013c). We will then use that method to select r for all the datasets.

We have two datasets that could use imputation. One of our experiments will be to determine if imputation changes our results. For this experiment, we will create two versions of each of the two datasets. For the Voting dataset, our two versions will consist of a dataset with imputed values, and a dataset with the missing values interpreted as third value. For the Breast Cancer dataset, the two versions will consist of a dataset with imputed values and a dataset where the examples with missing values have been removed. We will test all algorithms on all four datasets on classification accuracy, precision, recall, and F-measure. We will select the imputation strategy that gives the best combination of values for accuracy, precision, recall, and F-measure for later experiments

Our final experiment will compare all four algorithms on the five datasets based on accuracy, precision, recall, and F-measure. For conducting statistical tests, we will use student's t-tests to compare the outcome of the algorithms on each of the four measures. If warranted, we will run each cross-validation set of algorithm runs multiple times in order to fully explore the variation in the dataset.

References

Clarke, E. J., & Barton, B. A. (2000). Entropy and mdl discretization of continuous variables for Bayesian belief networks. *International Journal of Intelligent Systems*, *15*(1), 61–92.

Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, *13*(1), 21–27.

Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, *10*(7), 1895–1923.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns*. O'Reilly Media.

Friedman, N., Geiger, D., & Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning*, *29*(2-3), 131–163.

Fukunage, K., & Narendra, P. (1975). A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, *24*, 750–753.

Koller, D., & Friedman, N. (2009). *Probabilistic Graphical Models* (1st edition). MIT Press.

Lichman, M. (2013). UCI machine learning repository. `http://archive.ics.uci.edu/ml`. University of California, Irvine, School of Information and Computer Sciences.

Maurus, S., & Plant, C. (2014). Ternary matrix factorization. In *Proceedings of the 2014 IEEE International Conference on Data Mining*, ICDM '14, pp. 400–409.

Russell, S. J., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (3rd edition). Pearson Education.

Sheppard, J. W. (2016). *A Graduate Course in Machine Learning*. lecture notes distributed in CSCI 547 at Montana State University.

Stanfill, C., & Waltz, D. (1986). Toward memory-based reasoning. *Communications of the ACM - Special issue on parallelism*, *29*(12), 1213–1228.

UCI (2013a). Breast cancer Wisconsin (original) data set. `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29`. [Online; accessed 30 October 2016].

UCI (2013b). Congressional voting records data set. `https://archive.ics.uci.edu/ml/datasets/Congressional+Voting+Records`. [Online; accessed 30 October 2016].

UCI (2013c). Glass identification data set. `https://archive.ics.uci.edu/ml/datasets/Glass+Identification/`. [Online; accessed 30 October 2016].

UCI (2013d). Soybean (small) data set. `https://archive.ics.uci.edu/ml/datasets/Soybean+%28Small%29/`. [Online; accessed 30 October 2016].