Comp 3710, Artificial Intelligence Concepts

Final Project

### Group Members:

Cole Fuerth, Isaac Kilbourne, Mathew Pellarin, Ryan Dallaire

# Information Overview

## Data set Information

For this project, the chosen data set was a randomly generated list of hypothetical mushrooms that could exist in nature. The data set has 21 classifiable attributes and 8124 instances. This data set had only one classification task, which was to identify a mushroom's edibility. A mushroom can either be Edible or Poisonous. A link to the data set used in the project has been provided at the end of this report in the appendix section.

## Python Libraries

This project was done in Python, and the following main libraries were used:

* **Pandas** was used in order to read the CSV files the data was presented in, as well as to re-organize and re-label the data for more efficient processing.
* Both **Tensorflow-CPU** and **SKLearn** were used for implementing the actual learning methods mentioned after this section. Both libraries are equipped with built in methods and algorithms for machine learning. The **Tensorflow-GPU** library was also experimented with, but for unknown reasons it was far less efficient than the other 2 libraries, so it was cut from the project.
* **NumPy** has many methods for dealing with arrays, and is used frequently for storing and manipulating the data.

# Testing methods

For the classification task, we implemented and tested as many methods discussed in class as time allowed. For testing methods, we implemented both Hold-Out and K-fold cross validation. We wanted to test the significance that the 2 different methods would have on the outcome of the learning agent’s success. Theoretically K-fold should be far more effective in actual learning for the agent, and should provide better results than Hold-out, while hold-out should be more computationally efficient. Given the number of instances in this data set, higher values of K would increase training time and may not even yield notable results.

# Learning Methods

For this project, we managed to fully implement both decision trees and neural networks, and partially implement and test SVM learning.

## Decision Trees

Decision trees are the most basic of the implemented learning methods. The Agent will take in labeled data, and learn to make a prediction (a ‘decision’) given a set of values for some attributes by ‘splitting’ the data into subcategories based on the values of the most important attributes.

Tree construction is recursive. We first check if any of the following conditions are met:

* Are there more attributes to classify? If not, create a leaf node representing the decision .
* Are there more examples? If not, the default value argument is used (more on that below).
* Do all classifications in the examples match? If so, create a leaf node representing a decision of .

If none of the above conditions hold, we select the attribute with the highest Information Gain (entropy before splitting - entropy after splitting). Then, for each possible value of that attribute, we learn a subtree from the subset of examples which have . We also pass the remaining attributes to classify, and a new default value which is calculated from . This new subtree is then connected to the decision tree by a branch with label .

Once the tree is learned, it is very simple for the tree to make a decision. Starting from the tree’s root, while the current node isn’t a leaf node, move to the branch labeled where is the value of the attribute this node is classifying. Once a leaf node is reached, its value is returned as the tree’s decision.

There are only two hyperparameters in our implementation:

* A tree height limit, which restricts how many decisions the tree is allowed to make before making a prediction. As will be shown, it is not necessary for the tree to use all attributes to make a very good prediction.
* An absolute default value, which is used in the case that no examples are available. In this case, the tree simply decides this default on every input. This is effectively random guessing.

## Neural Networks

Neural Networks are more complex when implementing. The agent will take in labeled data only in the form of floating point numbers. These numbers go through a complex calculation to determine the probable output.

The first layer is the input layer. This layer is the total amount of data taken in by the neural network, or the amount of attributes the data may have. These values are then multiplied by a weight value and a bias is added within the neural network. Each node is then passed through an activation function.

There are many activation functions and algorithms each serving different purposes. The ones used in our implementation are rectified linear and sigmoid activation. Rectified linear activation can have any output ranging from zero to positive infinity. This means the data input will be the least restrictive but can only be a positive number. Each hidden layer uses this activation algorithm as the input can be more than two different values. The output layer uses the sigmoid activation function as this activation is binary. The node will then output a value between zero and one and if the output is greater than 0.5, prediction is true or in our dataset, the mushroom is edible, else the mushroom is poisonous.

The process each value goes through is called forward propagation. The input data has created an output probability but has not learned from its mistakes. Forward propagation uses training data and the expected output to adjust the network. As each prediction is made, the neural network adjusts each node weight to correct the output prediction accordingly.

The main hyperparameters used in this method are “epoch” and “batch size”. The epoch correlates to how many times a particular set of inputs is tested to train the network. The more epochs, the more times it will iteratively train the same set of data. The batch size will determine how many values will be passed through to determine what weights will be adjusted. In essence the larger the batch size, the more accurate the weight adjustment should be for each value.

**SVM**

A Support Vector Machine classifies data by viewing each data point as an n-dimensional vector, and then mapping that vector into a higher dimensional hyperplane. Data represented in lower dimensions cannot be easily told apart from other nearby pieces of data, which is why it is mapped to higher dimensions to more easily separate it. The mapping function used is called the kernel, and the type of function used for the kernel is the main hyper parameter for an SVM. Unfortunately, SVM was the last method we implemented, and we did not have time to test any other kernel functions besides linear. We did however implement 2 different versions of SVM using different python libraries, as we were trying to figure out which one would be best for the final run of the project.

# Literature Review

## Loss function and Gradient Descent Algorithms

Many of the hyper-parameters discussed above need little explanation. In this section we will give a brief explanation of the differences between the optimizers used for the Neural Network.

For the classification task, the network has a score function, and a loss function. The loss function of the network measures the quality of the parameters of its score function based on how accurate the predicted values are from the actual true values in the network’s training data. Optimization is the process of finding the best set of parameters, IE the set which minimizes the value of our loss function. There are many methods for this optimization but the ones used here are variants of Gradient Descent.

The loss function is, as the name suggests, a mathematical function that optimizes the parameters of the score. This means that we can in fact compute what the best direction to take at each step to minimize its values by following the slope of this function downwards. For one-dimensional functions the slope is the rate of change at a singular point. For higher dimensional functions, the gradient is a generalized version of the slope. Gradient descent simply refers to the process of calculating the gradient and updating the parameters of the score based on this calculation [1].

Generally, the gradient is calculated on batches of data in your training set, as this is more efficient and provides a good approximation of the gradient for the entire set. **Stochastic Gradient Descent (SGD)** is the extreme case in which each batch contains only a single element of the training data. This is obviously not very efficient for computing, but provides good results.

The other 2 optimization methods are both very complicated, so are only given a brief and shallow explanation here. For a more rigorous mathematical explanation of each method, see the references section at the end of the report.

While SGD uses the same learning rates for every parameter, **Adam** and **RMSprop** use individual learning rates for each parameter. At each parameter update, they both modify the learning rate for each parameter based on an exponentially decaying average of past square gradients computed for each other parameter. All this means is that parameters ‘close’ to each other will have more similar learning rates and should overall produce a more accurate result faster [2].

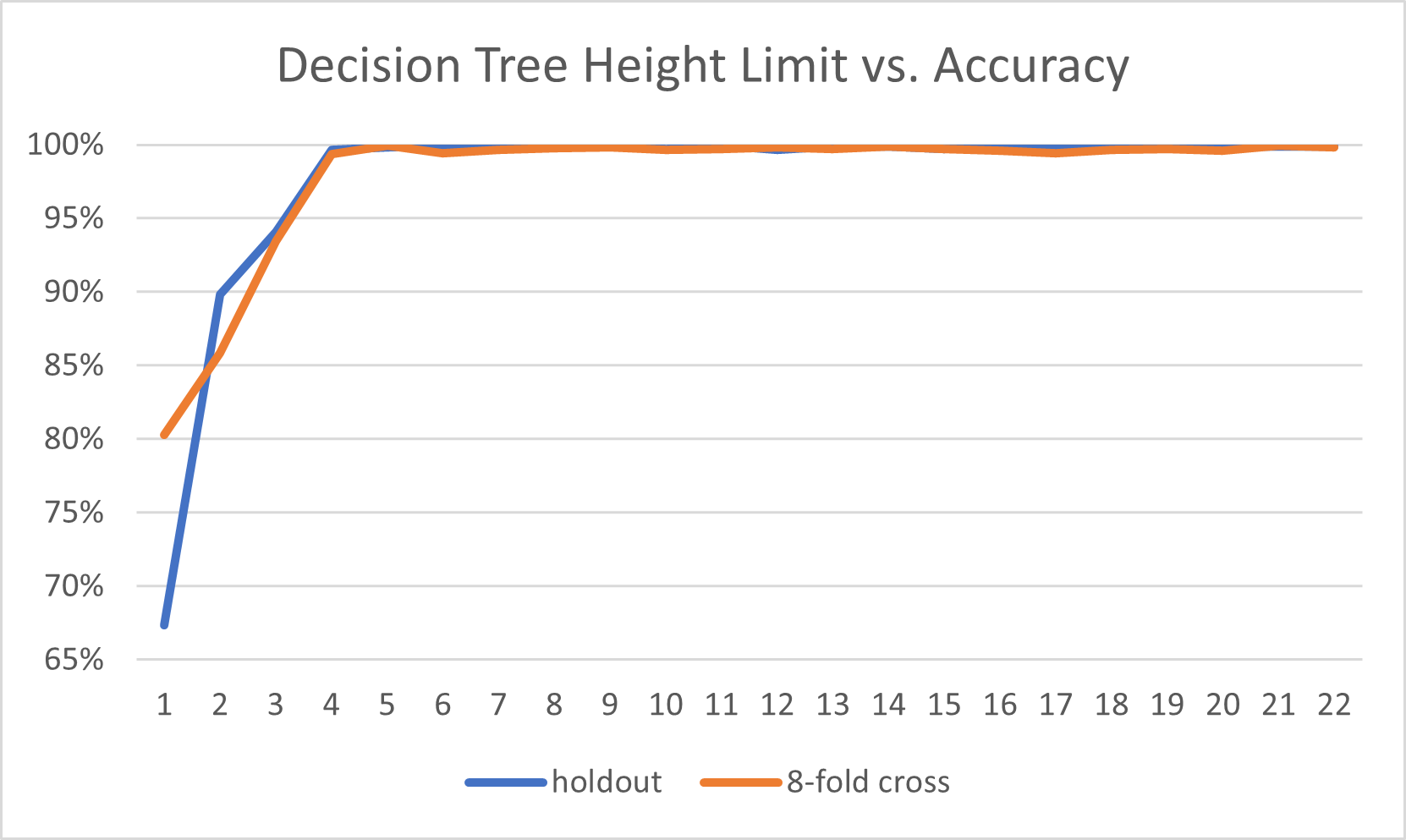
Adaptive Moment Estimation (Adam), on top of using a decaying average as RMSprop does, also has an exponentially decaying average of the *actual* gradients. These 2 values combined accelerate how fast we descend the gradient, ultimately leading to much quicker learning rates.

# Testing Results

Obviously results will go here: show learning results of different methods with different values of their hyper parameters, show which ones performed best/worst, explanation/guess as to why this may be the case.

## Decision Trees

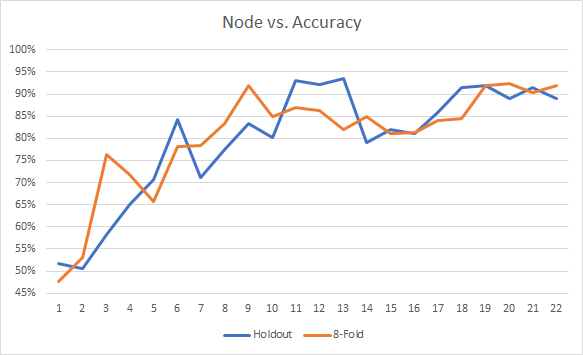
We found that using Information Gain to decide on the split attribute was very effective. The graph below shows the tree’s decision accuracy as a function of the tree’s allowed height.



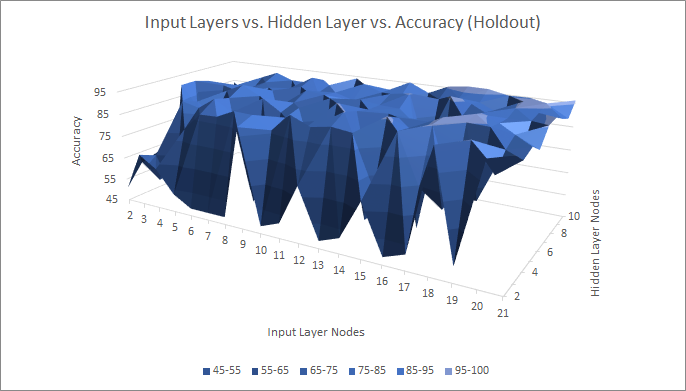
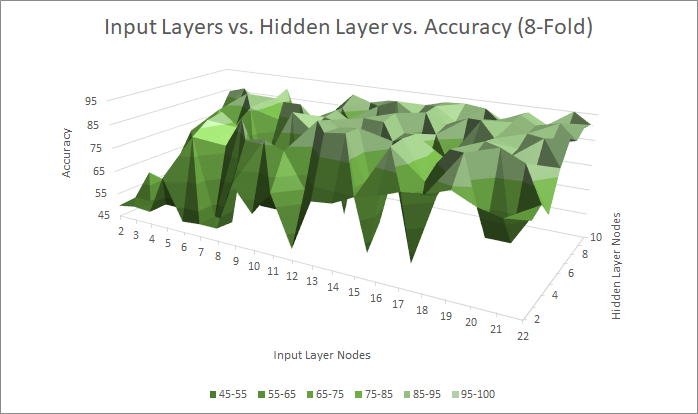
The accuracy quickly converges around 100% as tree height increases. With only one split permitted (a height of 1), the tree has an average accuracy of 74% - considerably better than random guessing, which would have an accuracy around 50%. With just one more split, the accuracy jumps to 90%, and by 4 splits it makes near-perfect predictions. The graph also shows that for holdout vs 8-fold cross validation, holdout performed much worse with only 1 attribute, slightly better with 2, and approximately the same for more than 2 attributes. Overall we were surprised by just how accurate the decision tree can become with such a short depth.

Not shown above is what happens when the tree is forced to use its absolute default. Unsurprisingly the tree gets about a 50% accuracy in this case since it is essentially flipping a coin for each of its decisions.

## Neural Network

During testing, we found using a Neural Network can also quickly achieve over 80% accuracy very quickly. Using a single layer network, an 80% accuracy can be accomplished by using only 10 nodes on a hidden layer with even weights. 

Testing multilayer networks prove to achieve a high accuracy with an average of 95%. This came with a cost of system resources but proved to be much more reliable when testing. With over 17 input layer nodes and 7 second hidden layer nodes, an accuracy over 90% can be achieved reliably. Adding more layers and nodes had about the same accuracy or slightly better accuracy.



In our implementation, we added the ability to tune hyperparameters. The parameters are weight, epoch count and training batch count. Adjusting these parameters modifies the accuracy. During our testing, if epoch was set too low or high, the accuracy would decrease dramatically. For the batch size hyperparameter, the same finding can be said as epoch. When the batch size was too low or high, accuracy would decrease as well. The minimum amount that was found to get reliable data was when epoch was set to 10 and batch size was set to 20. When adjusting the weight of each layer, using a random weight to each node had proven to be the most accurate where glorat had proven to be the worst.

**SVM**

The Support Vector Machine model has two python implementations in this project: one with TensorFlow, and the other with SKLearn. The reason for this was, TensorFlow is the was used for the NeuralNetwork implementation, and we wanted something to compare to that runs on the same backend, and SKLearn has a pre-build SVM model, that can just be loaded and trained without much tuning.

The TensorFlow implementation is considered a ‘Support Vector Machine’ because of two key features: (i) its first *hidden layer* is a RandomFourierFeatures layer, and (ii) it uses a `hinge` loss function. The `RandomFourierFeatures` layer used a ‘gaussian’ kernel initializer. This model also had a number of other hyperparameters, as TensorFlow allows for atomic control of models. We used two additional hidden layers after the Fourier layer, each with `relu` activation, and at 16 and 4 units, respectively. The output layer uses a `sigmoid` activation function, with only one unit.

The TensorFlow model was compiled using a hinge loss function, a binary accuracy function (as there are only two possible results), and uses the ‘Adam’ optimizer, at a learning rate of 0.001. We trained the model with 20 epochs, a batch size of 128, and a validation split of 50%. These hyperparameters produced a consistent accuracy of >99%.

The SKLearn model did not have many hyperparameters associated with it, as it was a pre-assembled model included in the library. The model used a `linear` kernel function, and an ‘ovo’ shape. This model runs faster than the TensorFlow model, but at the cost of some accuracy. The SKLearn accuracy results were inconsistent, ranging all the way from 98%, down to 60% in some cases.

# Conclusion

After testing each method, it is not completely clear which learning method could be considered “best”. Neural Networks and SVM performed very well with nearly every setting of the hyperparameters, however testing and tuning these took a lot of time and computing power. The Decision Tree had a terrible result with a low depth limit, but performed just as well once the depth was increased. It also had the bonus of being easy to test, since it only has one parameter to tune. Overall, once the decision tree had a large enough depth, it easily outperformed our neural networks and SVMs.

For the hyper parameters, ‘tree depth’ was the only parameter of decision trees but had a large impact on its success up to a certain point. If our data set had more attributes, the decision tree most likely would have been less efficient, requiring a much larger depth.

Batch size seemed to be the most impactful parameter for Neural Networks. We had the batch size range from 5 to 100, and found that if it was too high or low, it would reduce performance by a somewhat noticeable amount (5% - 10% reduction). Regardless of this, the neural network performed very well in all cases.

# Future work

An obvious first step in the future would be to continue testing the SVM method, as we ran out of time to actually tune its parameters. Besides SVM, the only method discussed in class we were not able to implement was back propagation. We figured that since it was mainly for regression, we would focus on the classification algorithms first instead. In the future, we would like to add additional learning methods to the project, including ones not discussed in class such as the “random forest” method, which incorporates multiple decision trees at once. There are also many other testing methods available than just hold out and k-fold [3]. It may be interesting to implement several others to see how they perform against the ones currently available. Lastly, this project of course only focused on a singular data set. The project required a set with at least 5000 instances and 20 attributes, but it would be interesting to test it on data sets with vastly different proportions. For example, a data set we came across had only 3300 instances, but had nearly 1600 attributes [4]. This is of course a much different set, and seeing which learning/testing methods perform best on different types of data could provide important insight.

# References

[1] Convolutional Neural Networks for Visual Recognition

<https://cs231n.github.io/optimization-1/#opt3> Accessed on: 2022-04-09.

[2] An overview of gradient descent optimization algorithms

<https://ruder.io/optimizing-gradient-descent/> Accessed on: 2022-04-10.

[3] Types of cross validation

<https://www.analyticssteps.com/blogs/7-types-cross-validation> Accessed on: 2022-04-16.

[4] Internet Advertisements Data Set

<https://archive.ics.uci.edu/ml/datasets/Internet+Advertisements> Accessed on: 2022-04-21.

# Appendix

Github link: <https://github.com/kilbouri/comp3710-group-project-2>

Link to data set used in this project: <https://archive.ics.uci.edu/ml/datasets/mushroom>