**Comparing the NNs and Decision Tree models for ISOLET Data**

Manasa Vaidyabhushana, Prasanthi Putta

*Under guidance of* Dr. Christos Nikolopoulos

8th of May 2020

1. **Abstract**

Automatic recognition of spoken letters is one of the most challenging tasks in the field of computer speech recognition. This is because of the acoustic similarity of the letters.

The goal of this research paper is to build a best decision tree model for classification of Isolet dataset and to build a Multilayered, feed-forward, back propagation neural net for the large Isolet dataset. This experimentation determines the best architecture for the net and best values for the various tuning parameters that result in network of the smallest RMS error. At the end, we compared the NNs and Decision Tree models in terms of accuracy to come up with the best model. The dataset we used for this purpose is Isolated Letter Speech Recognition, which is considered a good domain for a noisy, perceptual task. It is also a very good domain for testing the scaling abilities of algorithms. Various algorithms were applied and compared.

1. **Introduction/background** 
   1. Machine learning is a complex discipline. But implementing machine learning models is far less daunting and difficult than it used to be, thanks to machine learning frameworks—such as [Google’s TensorFlow](https://www.tensorflow.org/)—that ease the process of acquiring data, training models, serving predictions, and refining future results. Created by the Google Brain team, TensorFlow is an open source library for numerical computation and large-scale machine learning. TensorFlow bundles together a slew of machine learning and deep learning (aka neural networking) models and algorithms and makes them useful by way of a common metaphor. It uses Python to provide a convenient front-end API for building applications with the framework, while executing those applications in high-performance C++.
   2. The dataset used in our research paper is Isolet. This numerical dataset aims to predict which letter-name was spoken with 618 continuous, real valued attributes scaled into the range -1.0 to 1.0 and 6238 instances. 150 subjects spoke the name of each letter of the alphabet twice. Hence, we have 52 training examples from each speaker. The speakers are grouped into sets of 30 speakers each, and are referred to as isolet1, isolet2, isolet3, isolet4 and isolet5. The data appears in isolet1+2+3+4. In sequential order. The test set, Isolet, is a separate file. Network architecture: 56 hidden units, 26 output units (one-per-class
   3. Weka is a collection of machine learning algorithms for solving real-world data mining problems. It is written in Java and runs on almost any platform. The algorithms can either be applied directly to a dataset or called from your own Java code.

In this paper, we used two algorithms for our classification problem.

* + 1. **Decision Tree**

Decision trees can support classification and regression problems.

Decision trees are more recently referred to as Classification and Regression Trees (CART). They work by creating a tree to evaluate an instance of data, start at the root of the tree and moving town to the leaves (roots) until a prediction can be made. The process of creating a decision tree works by greedily selecting the best split point in order to make predictions and repeating the process until the tree is a fixed depth.

After the tree is constructed, it is pruned in order to improve the model’s ability to generalize to new data.

* + 1. **K-Nearest Neighbors**

The k-nearest neighbors algorithm supports both classification and regression.

It is a simple algorithm, but one that does not assume very much about the problem other than that the distance between data instances is meaningful in making predictions.

When making predictions on classification problems, KNN will take the mode (most common class) of the k most similar instances in the training dataset.

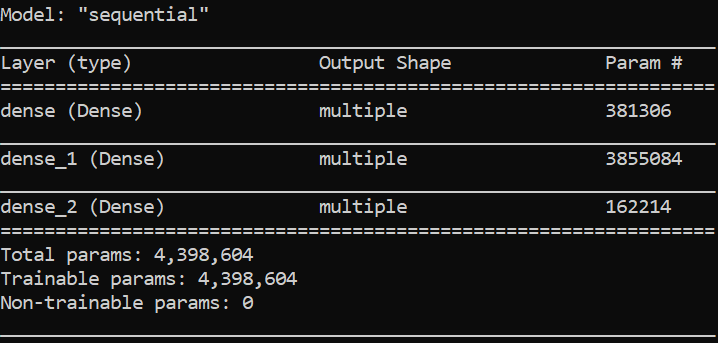
* 1. Multi-Layer Perceptron was built for ISOLET data set using TensorFlow whereas Keras API does back propagation automatically. In the Multilayer Perceptron, there can be more than one linear layer (combinations of neurons).

1. **Main Results**
   1. **Section 1 (Multilayer Perceptron results)**
      1. For this Data Set, we have built three-layer network using TensorFlow Keras API:
2. First layer is the *input layer* with 617 neurons
3. Last is *output layer* with 26 neurons, one for each alphabet
4. Middle layer is called *hidden layer,* which  *was* madewith *6238* neurons*.*

In a supervised classification system, each input vector is associated with a label, defining its class or class label is given with the data. The output of the network gives a class score, or prediction, for each input. To measure the performance of the classifier, the loss function is defined. The loss will be high if the predicted class does not correspond to the true class, it will be low otherwise.

* + 1. Model Architecture for our dataset ISOLET: **Sequential Model**

1. “**Relu**” activation function for input and hidden layers - Rectified Linear Unit (relu) activation function is linear (identity) for all positive values, and zero for all negative values. It’s cheap to compute as there is no complicated math. The model can therefore take less time to train or run
2. “**Softmax**” activation function for output layer - SoftMax turn logits (numeric output of the last linear layer of a multi-class classification neural network) into probabilities by taking the exponents of each output and then normalize each number by the sum of those exponents so the entire output vector adds up to one — all probabilities should add up to one. It’s a core element used in deep learning classification tasks.
3. **Cross entropy loss** - is usually the loss function for a multi-class classification problem. So, Categorical Cross entropy was used as our labels were categorical.
4. **“Adam Optimization algorithm”**  had been used as it is an adaptive learning rate optimization algorithm that’s been designed specifically for training deep neural networks. Adam is a combination of **RMSprop** and **Stochastic Gradient Descent** with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.
5. **“Parameter”** column in the below figure indicates the weights that passed through each layer.



Model Architecture for ISOLET Data Set

* + 1. **Experiments For Choosing The Best Model:**

It is critically important to consider myriad of decisions while building a neural network. Hyper parameters play a crucial role in determining a machine learning model’s performance. In a neural network, hyper parameters include the **number of epochs**, **batch size**, **number of layers**, **number of nodes** in each layer, and so on. Adapting the **learning rate** and pre-defined learning rateschedules for Adam optimization procedure can increase performance and reduce training time. So, our job is to set the initial learning rate then the optimization techniques will automatically set the learning rates which are best suitable for the data set.

1. Using trial and error method, we found 0.005 as best initial learning rate and we got **94% test accuracy and 97% train accuracy** after a lot of experiments.
2. Just to give a try with the model, learning rate was set to 0.002 and we achieved to get maximum **test accuracy of 95% and 99% train accuracy**.

1. And for epochs, we set it initially to 10 and increased till we got the maximum accuracy.
2. We found setting the batches size is very complex task but TensorFlow made it very easy. Initially, we have set batch size to 1000 but with help of a TensorFlow warning, we could able to identify batch size to be 162188, which is 6238\*52 where 6238 are the training instances.
3. Below are the final hyper parameters for neural net:
4. **Initial learning rate = 0.002**
5. **Adam Optimizer automatically sets the accurate momentum for neural net.**
6. **Epochs =100**
7. **Batch Size = 162188**
8. **Number of layers=3**
9. **Nodes for each layer**
   1. **Input layer = 617**
   2. **Hidden layer = 6238**
   3. **Output layer =26**
10. Following are the experiments using different combinations of activation functions and Optimizations Algorithms.

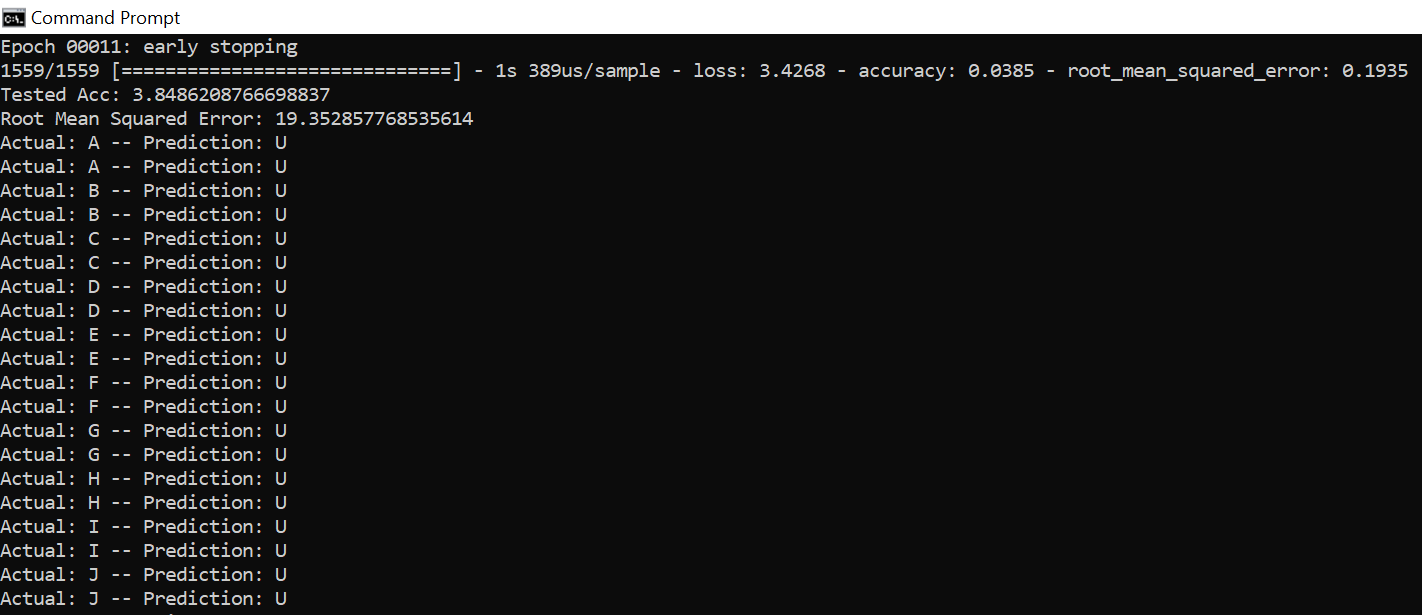
**Experiment 1:**

Initially we considered learning rate to be 0.005

Using “Sigmoid” activation function for Input and Hidden layer and “Softmax” activation function for Output layer, along with “Adam” Optimizer

Test Accuracy = 3.84%.

Root Mean Squared Error = 19.35%

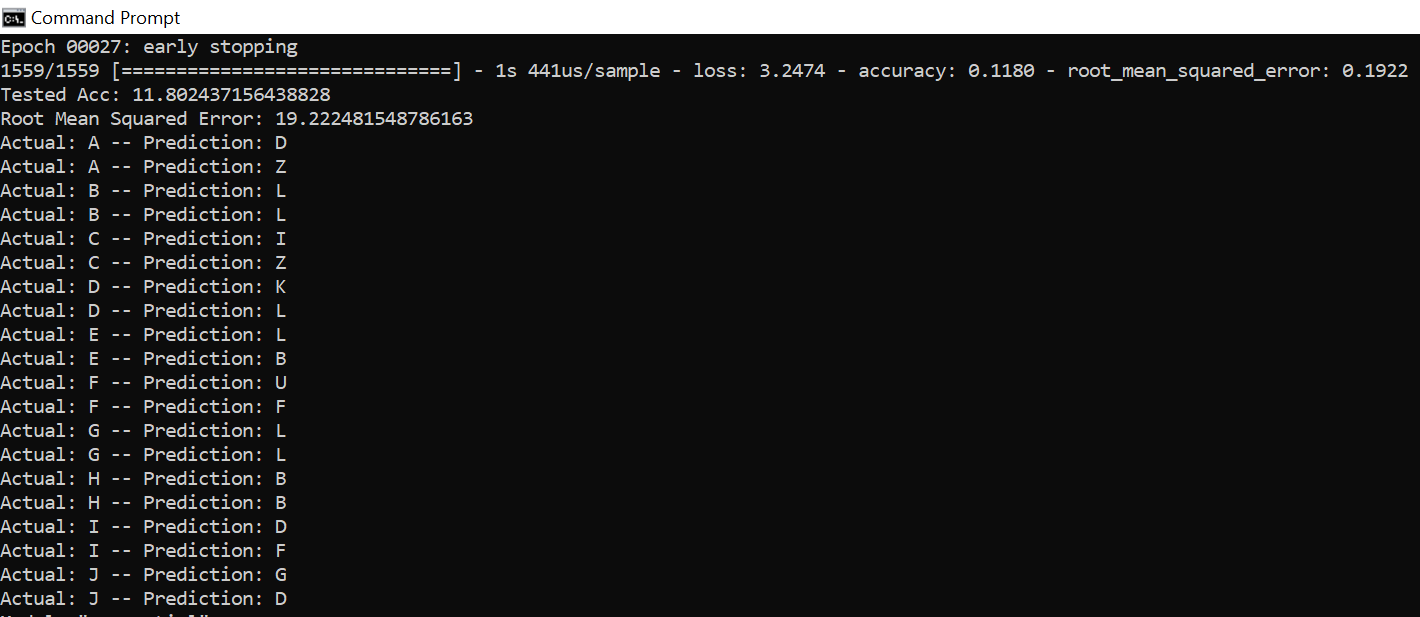


**Experiment 2:**

Using “Sigmoid” activation function for Input and Hidden layer and “Softmax” activation function for Output layer, along with “Gradient descent” Optimizer.

Test Accuracy = 11.80%.

Root Mean Squared Error = 19.22%

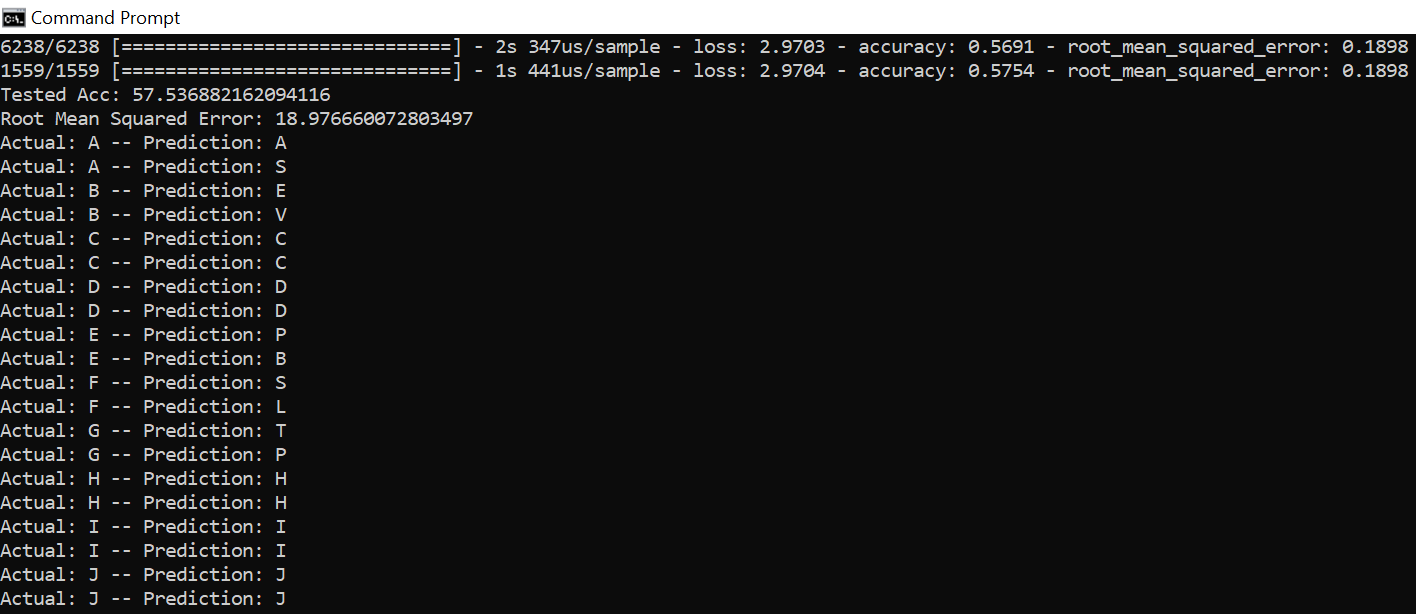


**Experiment 3:**

Using “relu” activation function for Input and Hidden layer and “Softmax” activation function for Output layer, along with “Gradient descent” Optimizer.

Test Accuracy = 57.53%

Root Mean Squared Error: 18.97%

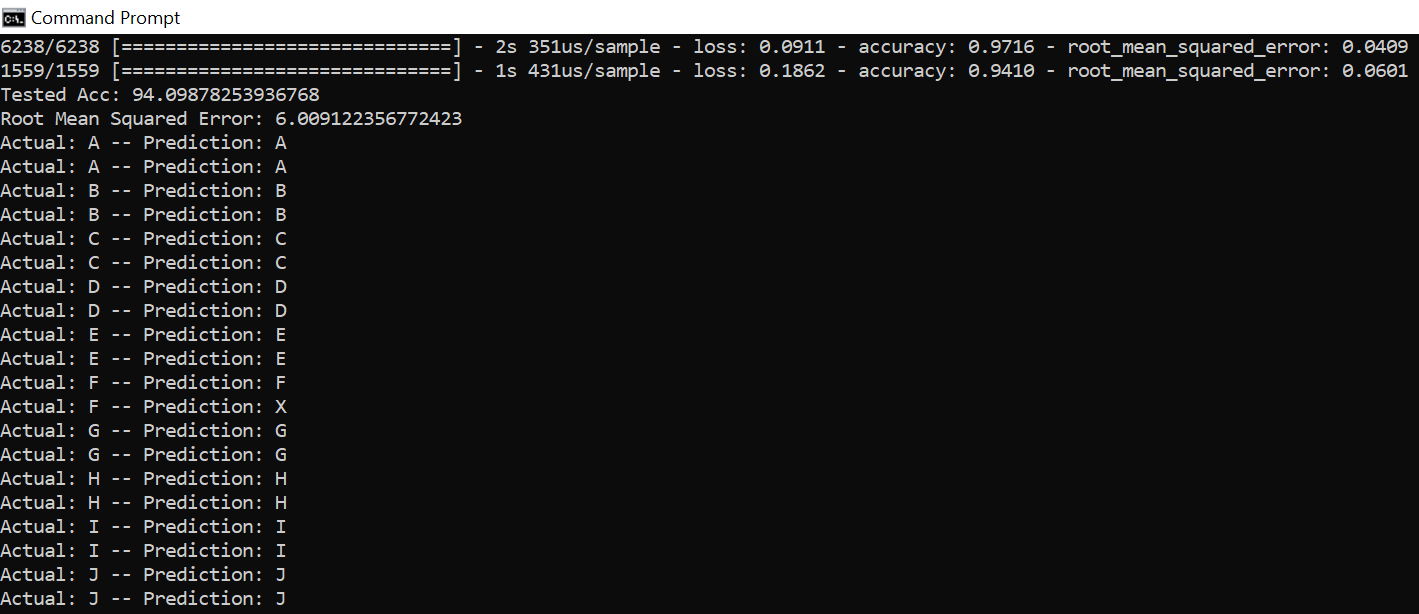


**Experiment 4:**

Using “relu” activation function for Input and Hidden layer and “Softmax” activation function for Output layer, along with “Adam” Optimizer, we got 94.9% train accuracy and 92.4% test accuracy.

Test Accuracy = 94.09%

Root Mean Squared Error = 6.009%

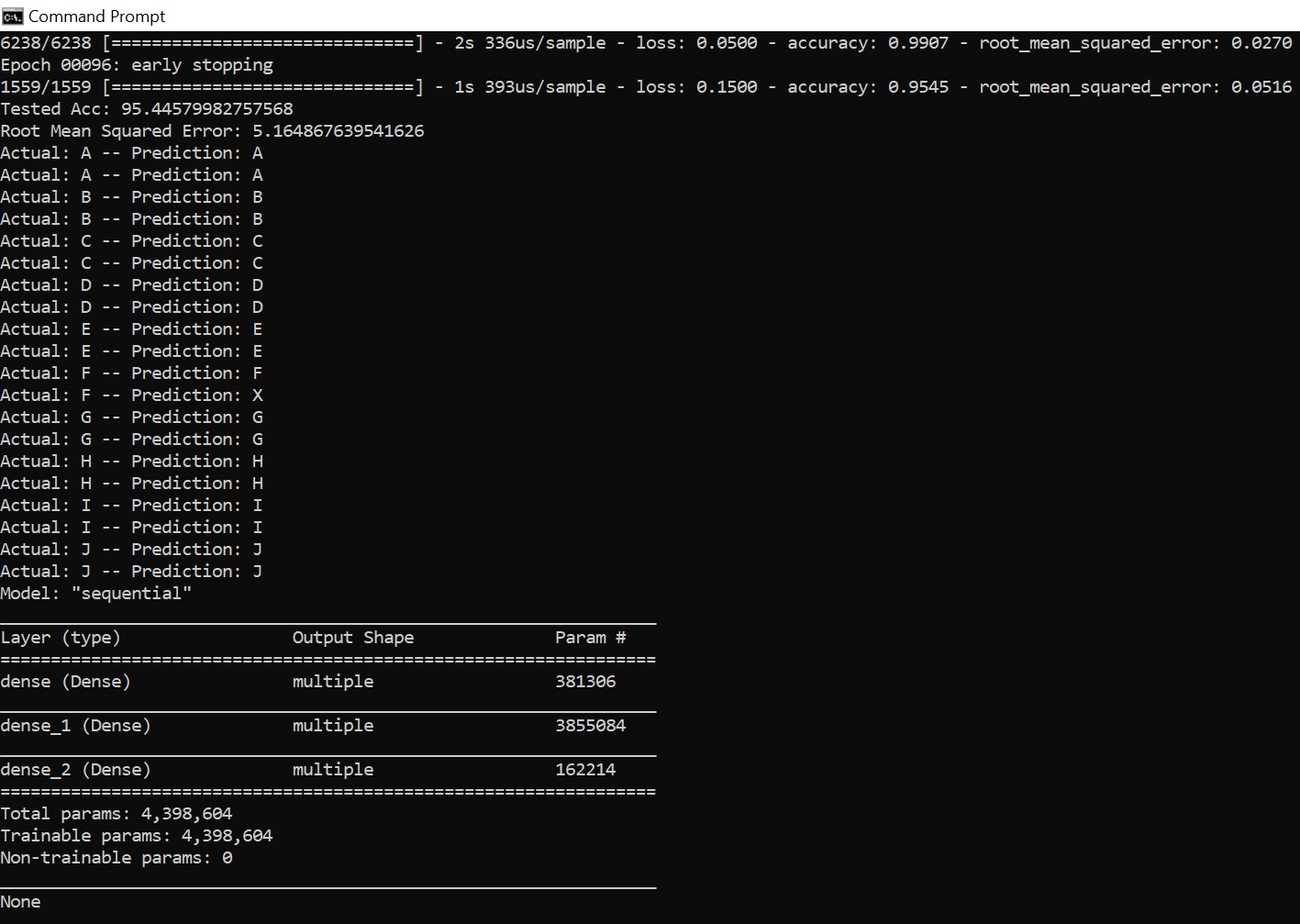


**Experiment 5:**

As we got good accuracy using “relu” activation function for Input and Hidden layer and “Softmax” activation function for Output layer, along with “Adam” Optimizer. Also, we have changed the **learning rate** to 0.002.

Test Accuracy = 95.44%

Root Mean Squared Error: 5.16%



**Final Result Set of best model:**

Train Accuracy = 99.1%

Root Mean Squared Error for Training Data Set=2.7%

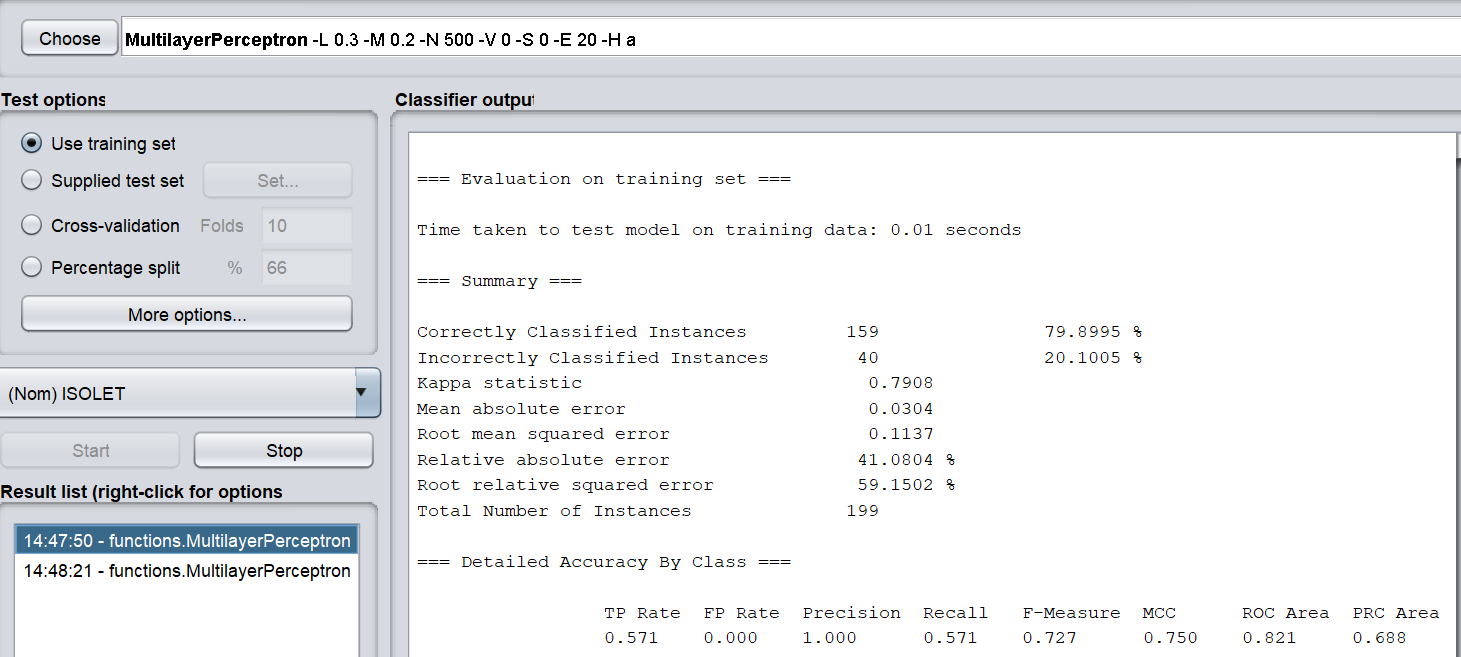
Test Accuracy = 95.4%

Root Mean Squared Error for Testing Data Set = 5.16%

Considering 20 test data inputs; we got one miss prediction, where F is predicted as X.

|  |  |
| --- | --- |
| Actual Output | Desire Output |
| A | A |
| A | A |
| B | B |
| B | B |
| C | C |
| C | C |
| D | D |
| D | D |
| E | E |
| E | E |
| F | F |
| F | X |
| G | G |
| G | G |
| H | H |
| H | H |
| I | I |
| I | I |
| J | J |
| J | J |

* + 1. We also tried to build a neural network using Multilayer Perceptron in WEKA, but it was taking too long with our dataset as its too big. Just to give it a try we tried running the classifier with a training dataset of 200 instances. Following results were observed.

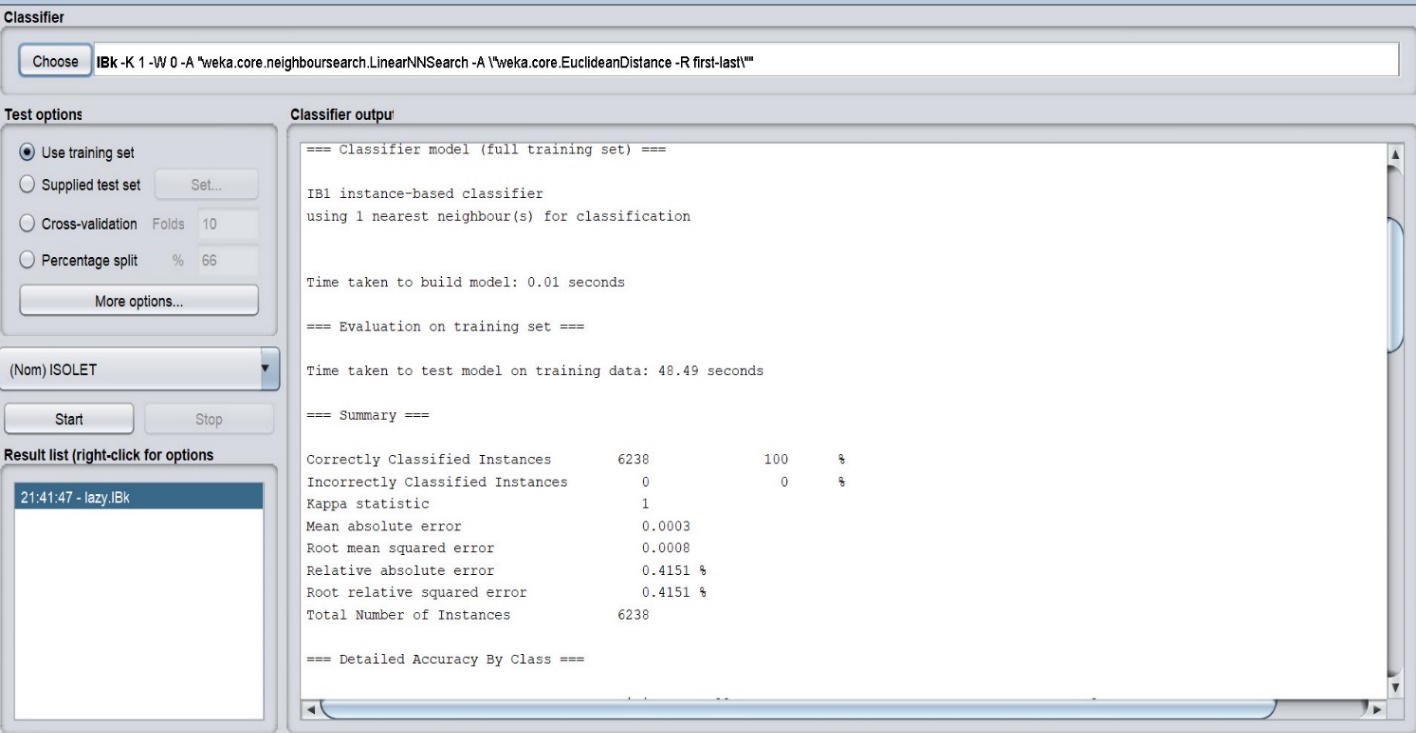


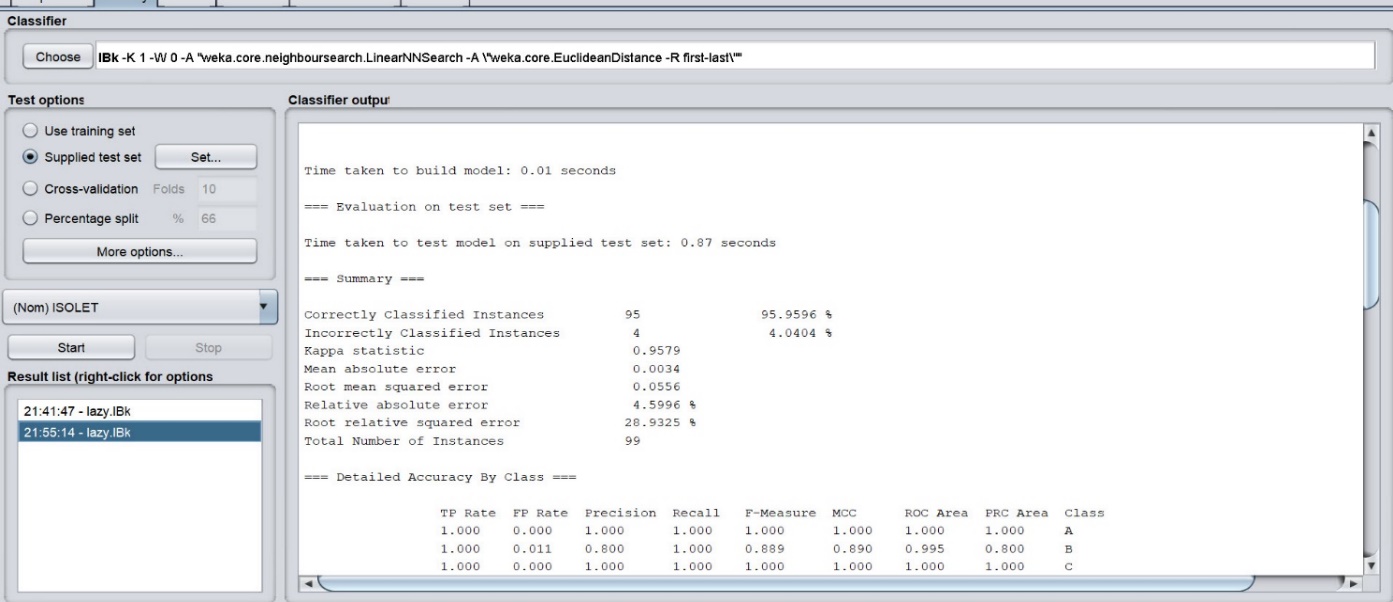
**Obviously, the accuracy rate was relatively lower, as we took only 200 instances.**

* 1. **Section 2 (Decision Tree)**

The above dataset was run through several classifiers on two different test options, i.e., Training set and Supplied test set. Different accuracy rates with different RMS error were spotted.

* + 1. **First algorithm - k-nearest neighbor**



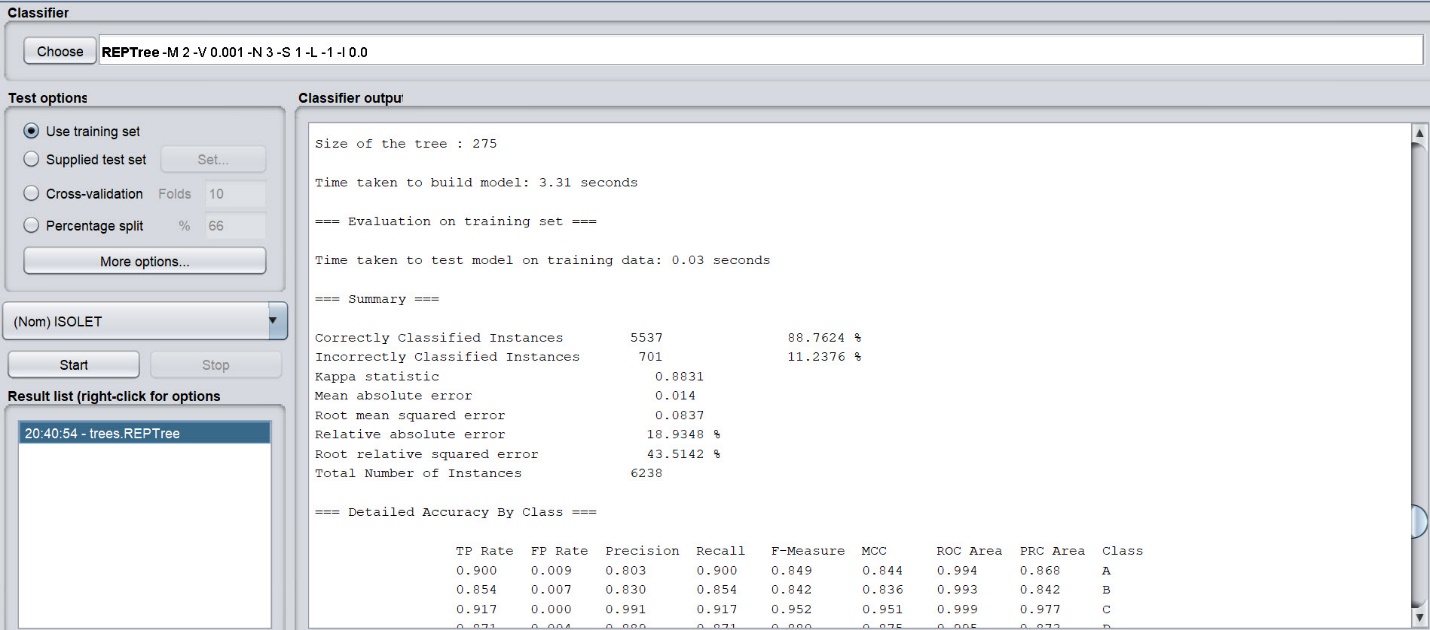


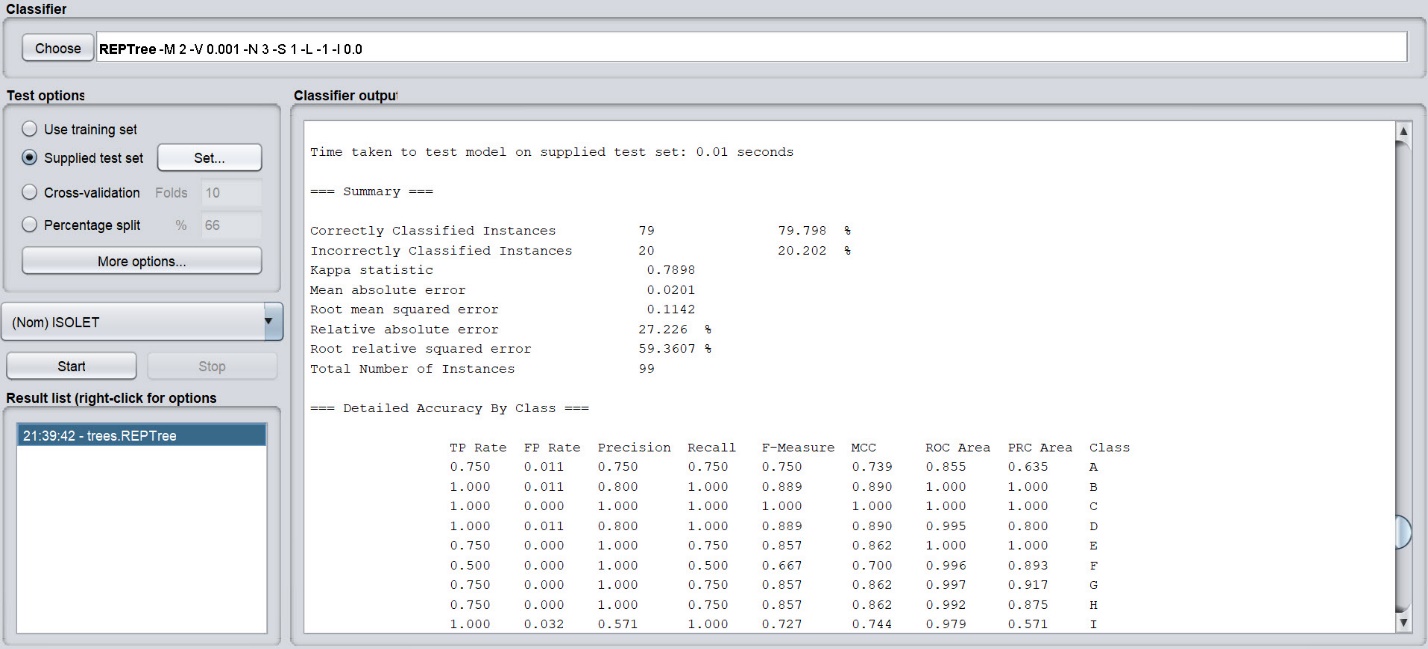
Parameters used:

1. KNN value: 1
2. Distance Weighting: No distance weighting
3. nearestNeighbourSearchAlgorithm: Linear NN Search
4. distanceFunction : Euclidean distance

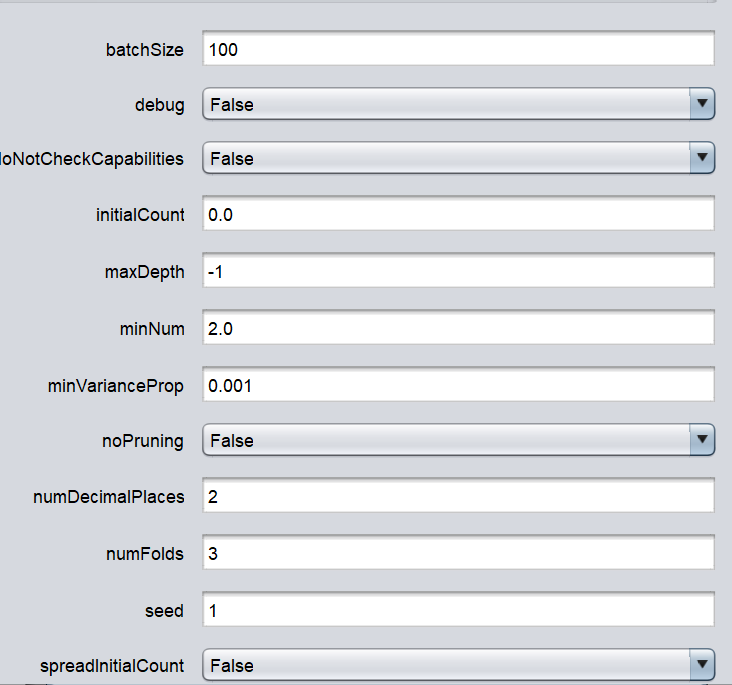
**An accuracy of 100% was found using training Set and a 95.9% using a Supplied test set.**

* + 1. **Second algorithm: Decision Trees**
       1. **REP Tree - Fast decision tree learner.**



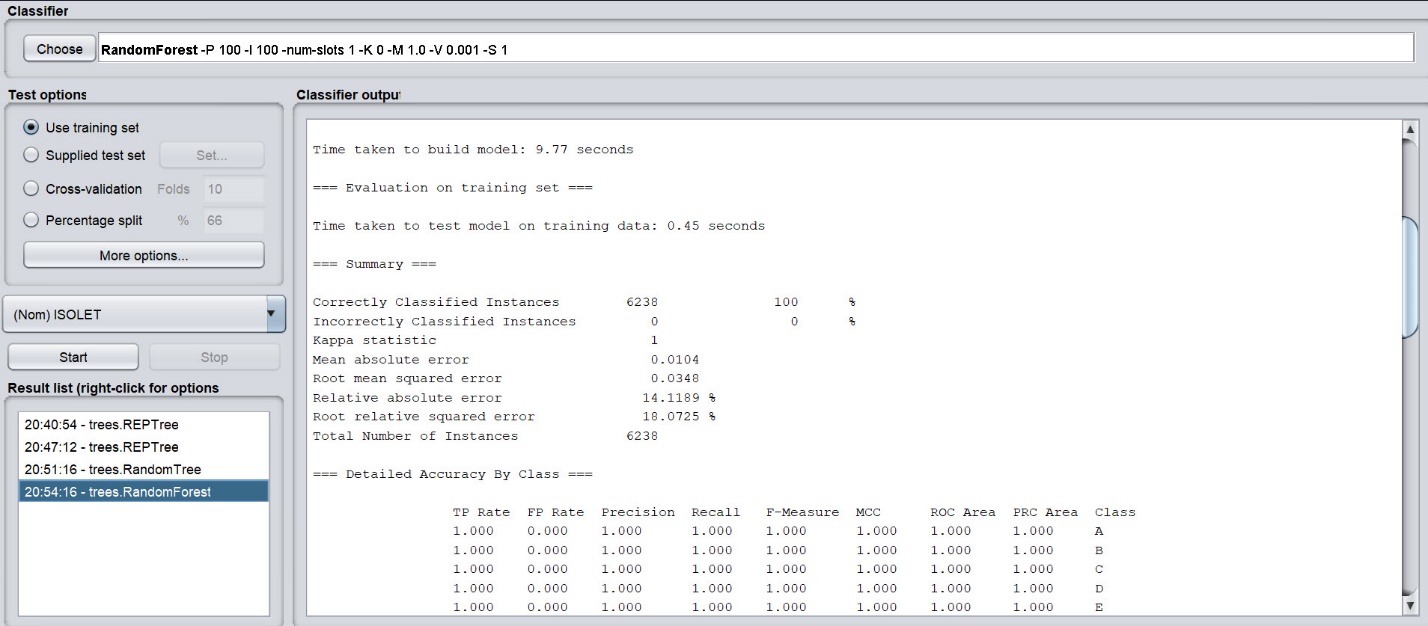


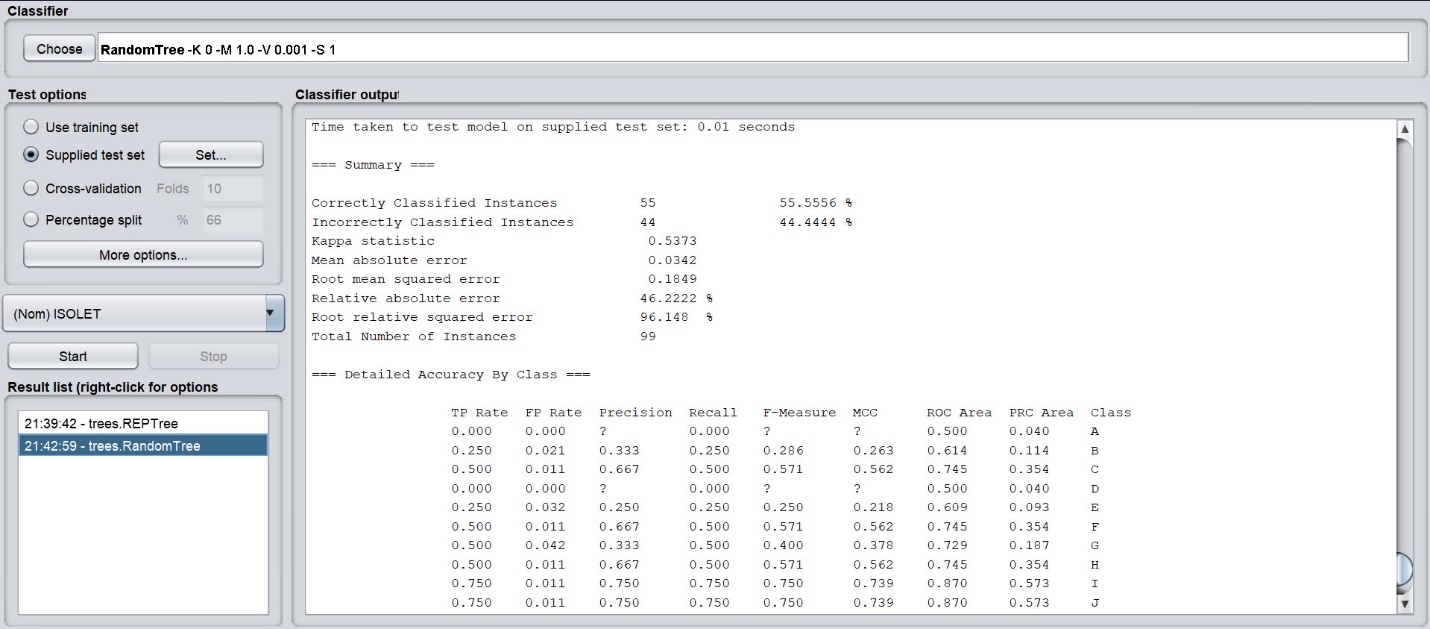
Parameters used:



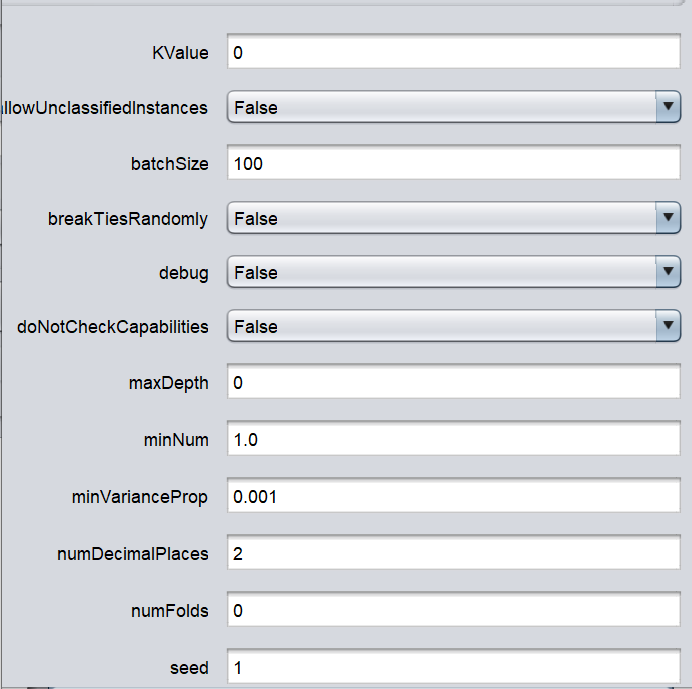
**An accuracy of 88.7% was found using training Set and a 79.8% using a Supplied test set.**

* + - 1. **Random Tree - Class for constructing a tree that considers K randomly chosen attributes at each node.**



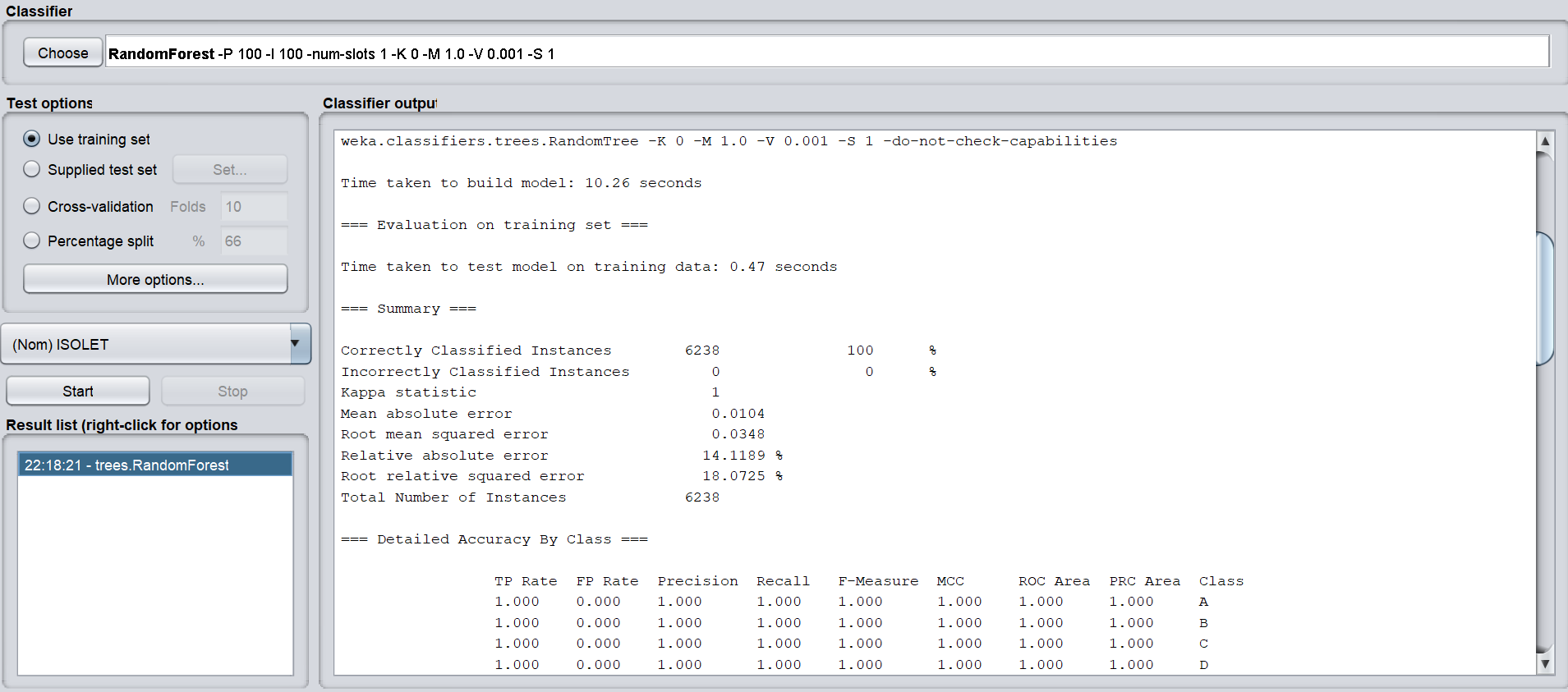


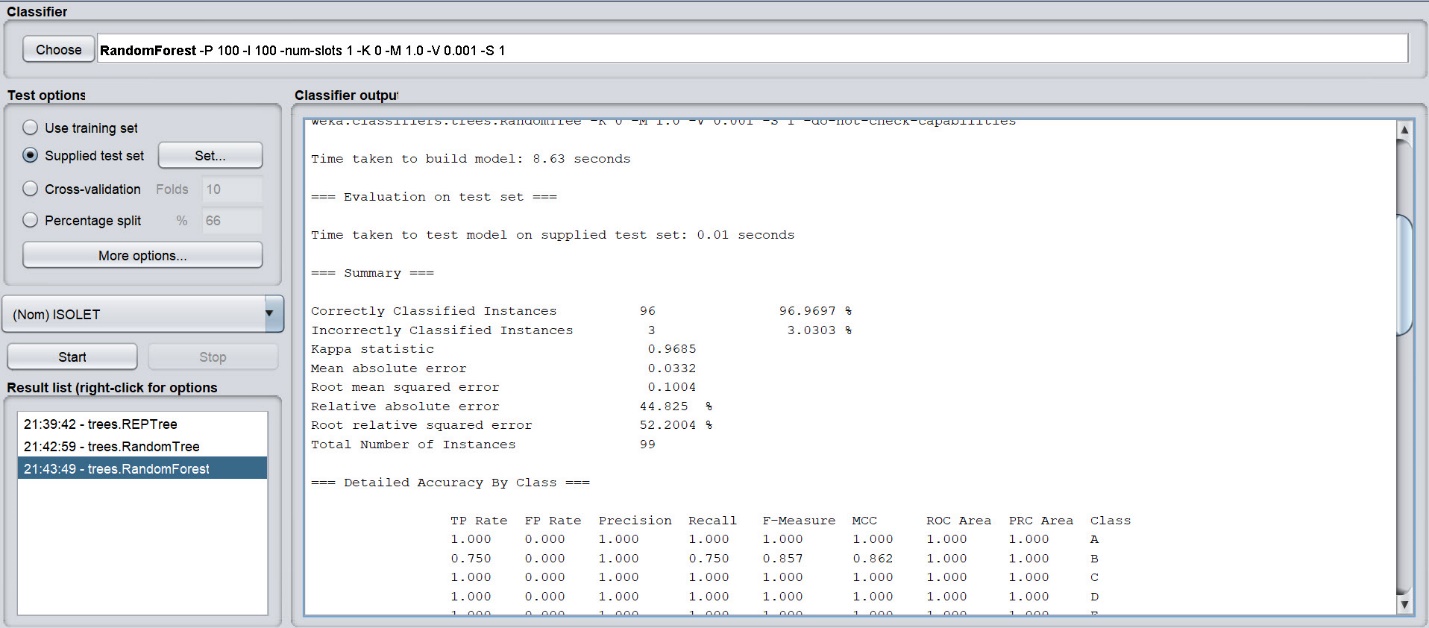
Parameters:



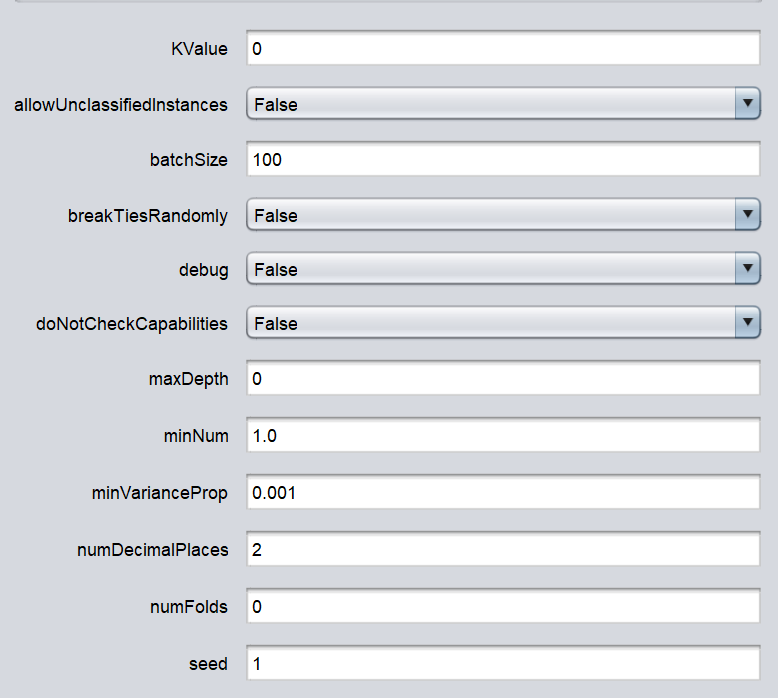
**An accuracy of 100% was found using training Set and a 55.5% using Supplied Test Set**

* + - 1. **Random Forest Tree - Class for constructing a forest of random trees.**



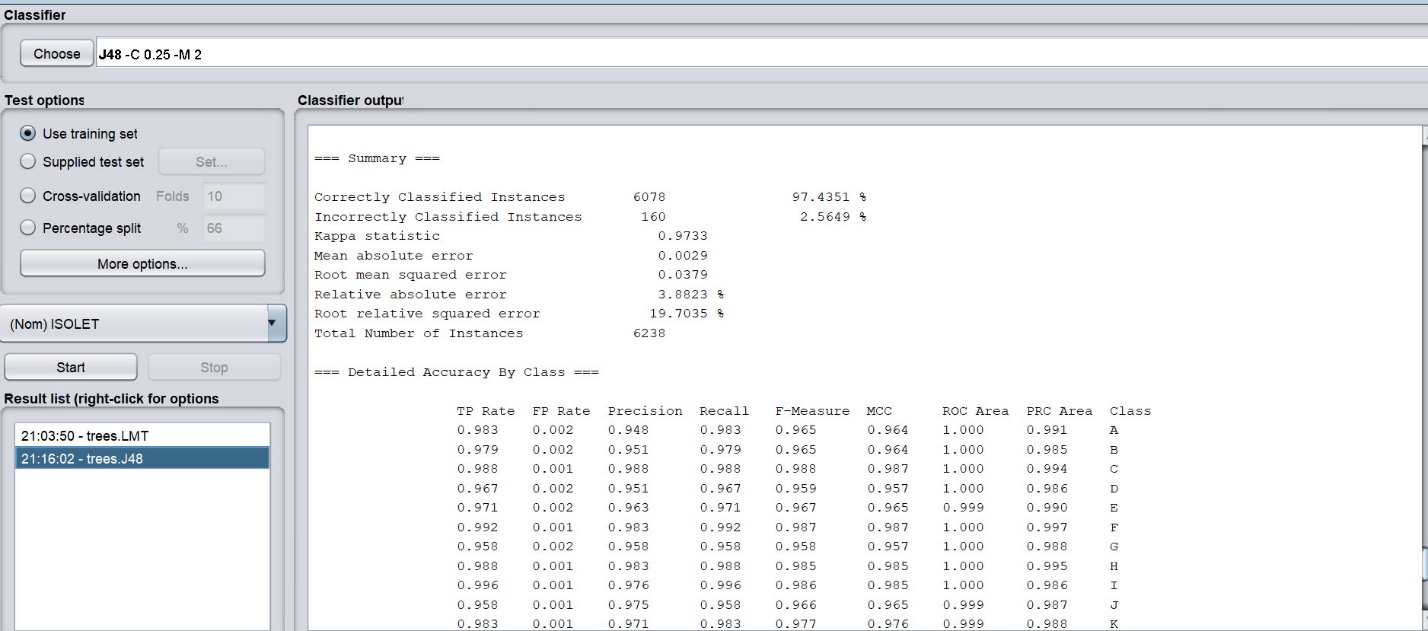


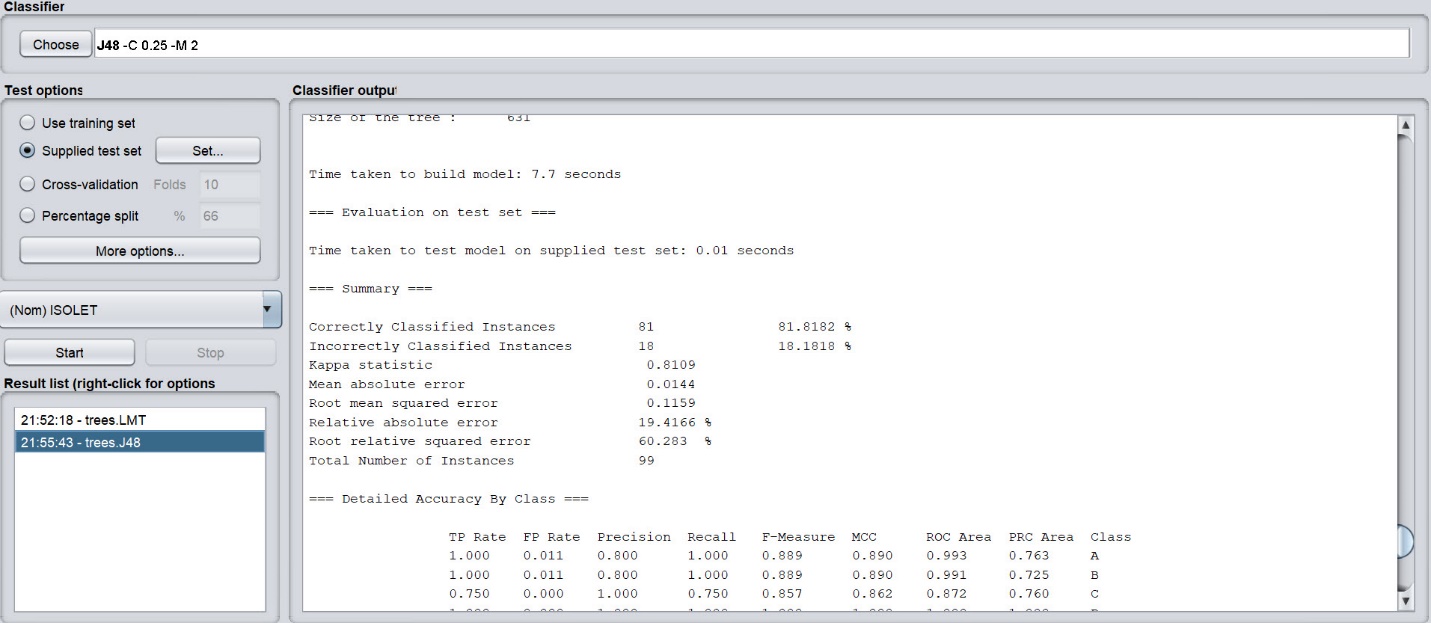
Parameters:



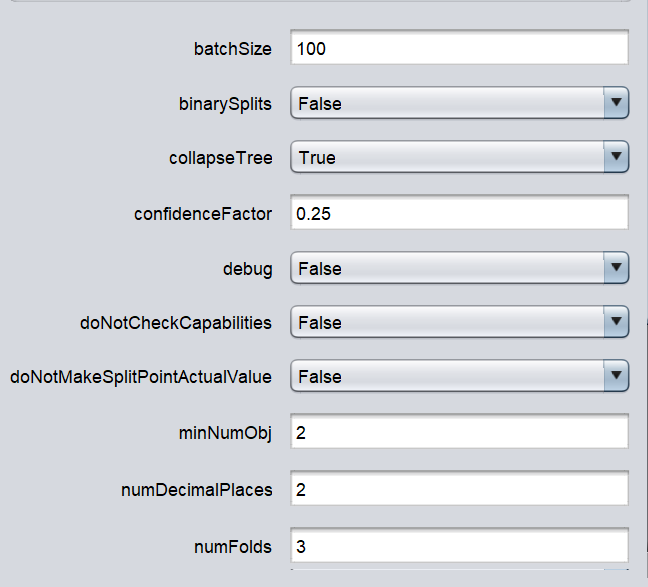
**An accuracy of 100% was found using training Set and a 55.5% using Supplied Test Set**

* + - 1. **J48 tree - Class for generating a pruned or unpruned C4.**



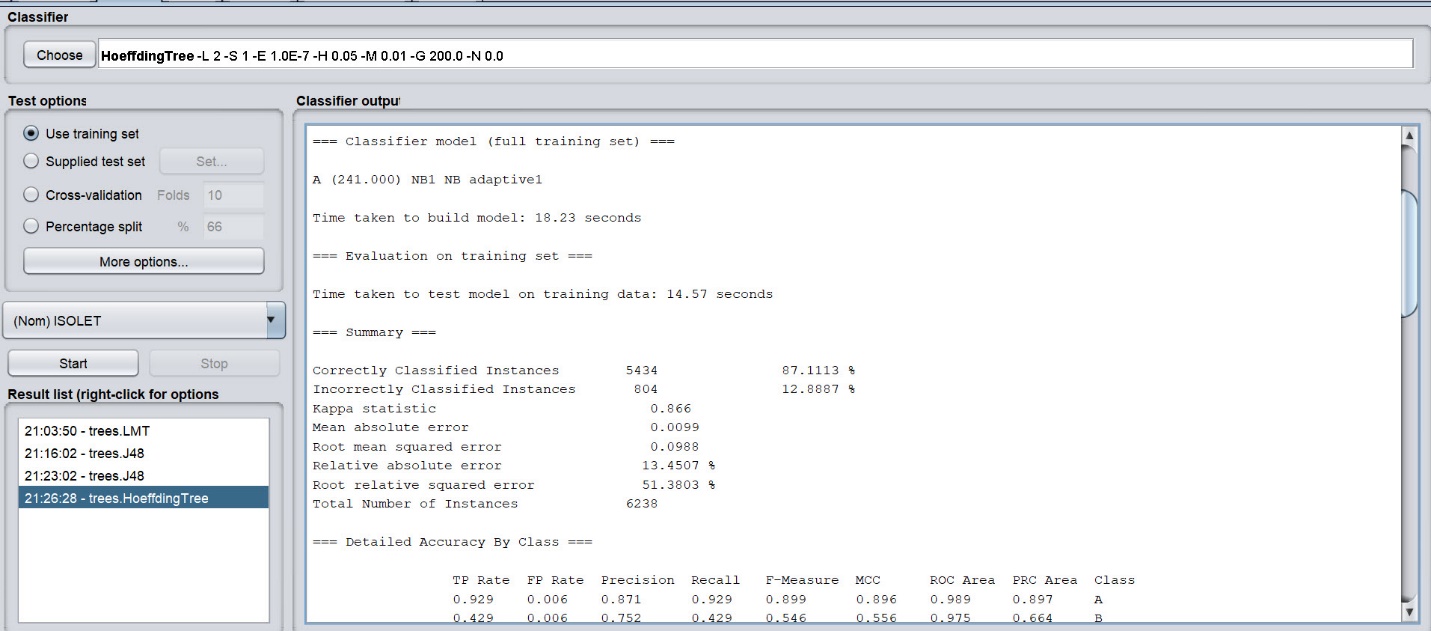


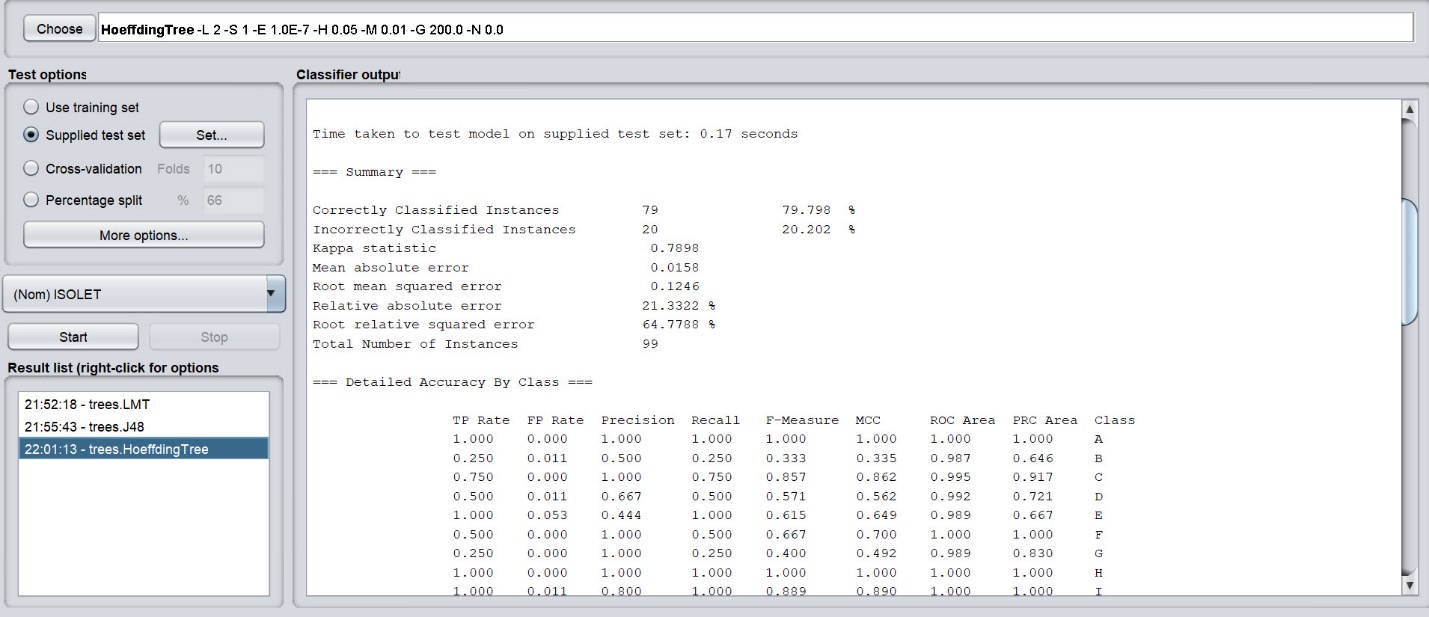
Parameters:



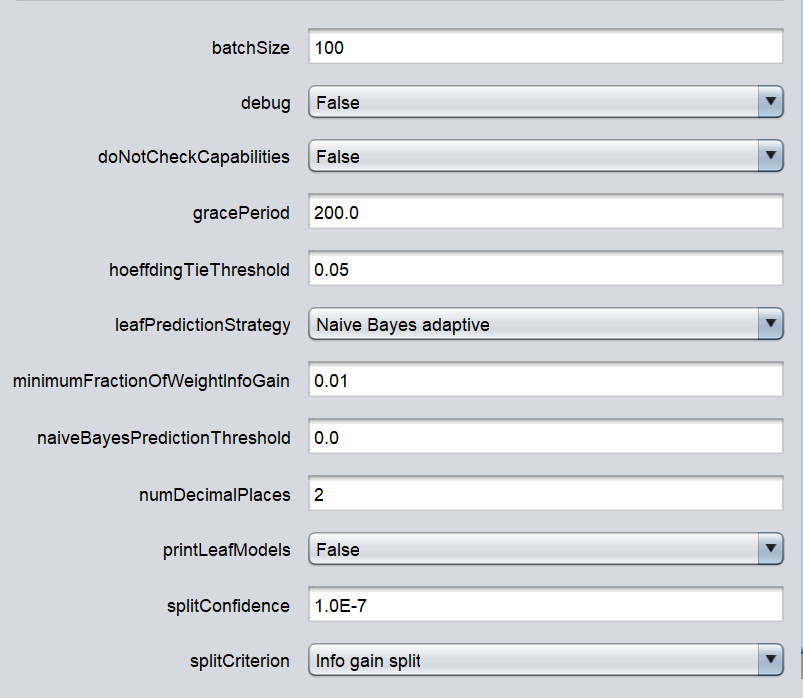
**An accuracy of 97.4% was found using training Set and a 81.8% using Supplied testing Set.**

* + - 1. **Hoeffding Tree - A Hoeffding tree (VFDT) is an incremental, anytime decision tree induction algorithm that is capable of learning from massive data streams, if the distribution generating examples does not change over time.**



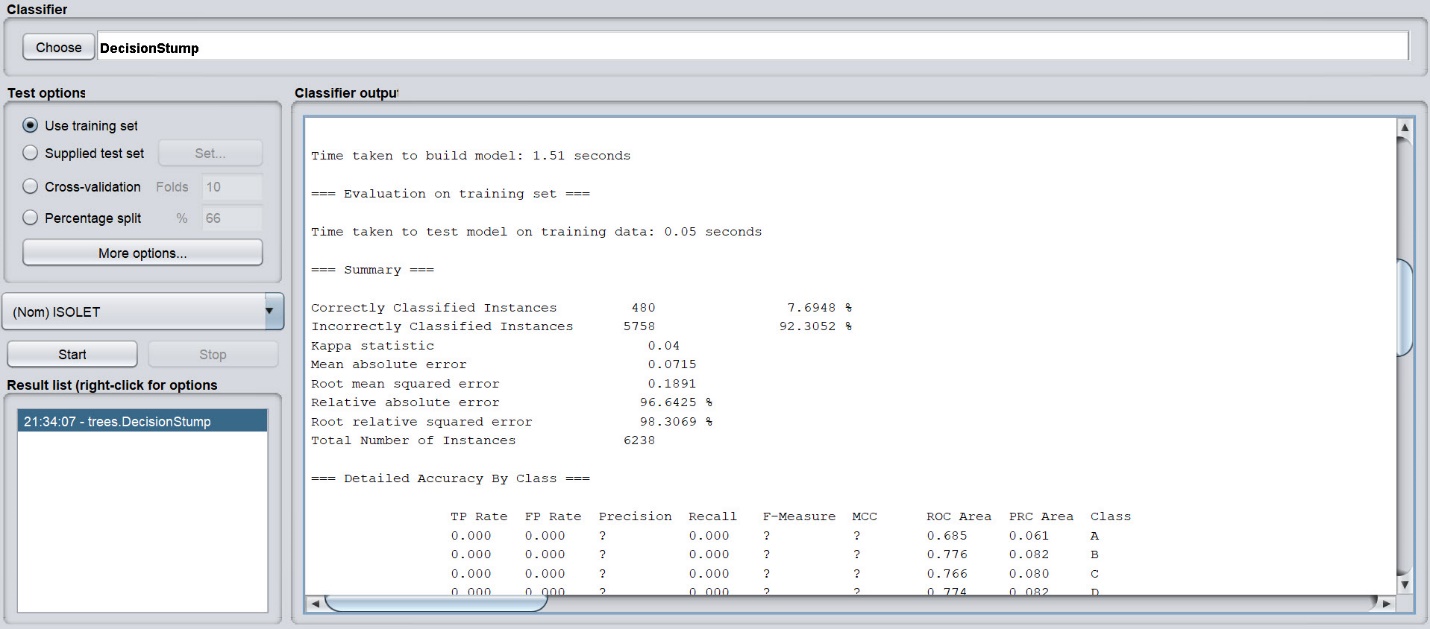


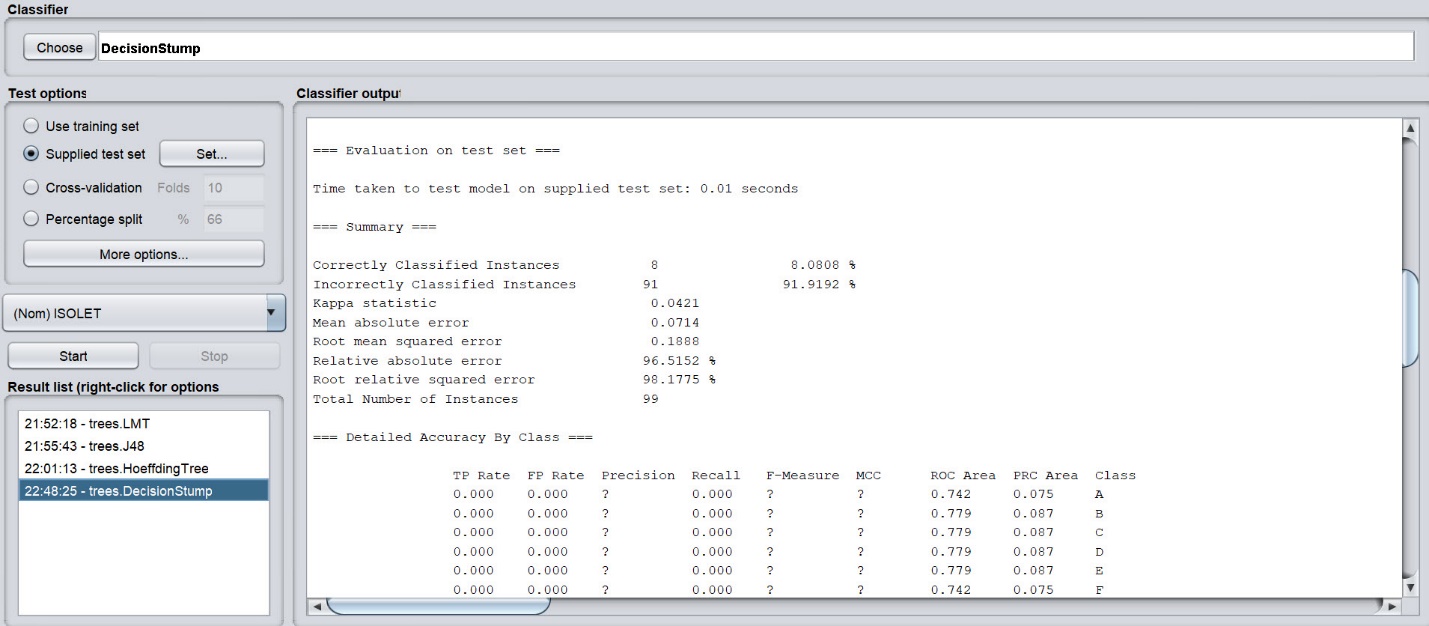
Parameters:



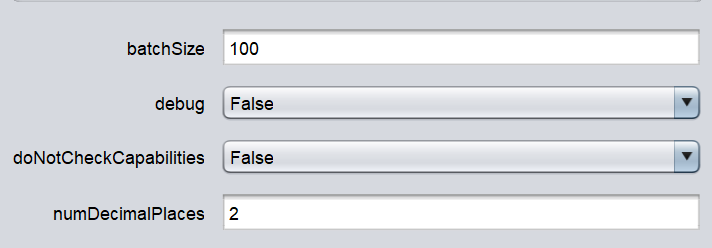
**An accuracy of 97.4% was found using training Set and a 81.8% using Supplied testing Set.**

* + - 1. **Decision Stump - Class for building and using a decision stump.**





Parameters:



**A very low accuracy of 7.6% was found using Training Set and a 8.08% using Supplied Testing Set.**

1. **Conclusions (summary of what you just presented and summary of results)**

* **We compared the NNs and Decision Tree models in terms of accuracy. From our research, we found that , the accuracy for both Neural Network Model and Decision Tree Model are very high ranged from 95% to 100% for different training algorithms. Although, Neural Net which was built using TensorFlow yield very high accuracy of about 95%, we found that using Decision tree models is more accurate than using Neural Network models in Weka. In terms of speed, Decision tree in weka was faster in getting the results than Tensor flow which differed for every set of optimizations. For our dataset, even though Random Forest presented better results of 100% using training set than J48 Decision tree, in our opinion J48 is the best model, because it showed a high accuracy for both types of datasets - 97.4% for training set and 81.8% for supplied test set.**

1. **References**
2. Ronald Cole. [Mark Fanty. (1990 June) Spoken letter recognition.](javascript:void(0);) <https://dl.acm.org/doi/abs/10.3115/116580.116725>
3. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, (n.d) TensorFlow: A System for Large-Scale Machine Learning.<https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>a
4. Weka. (2020 April). <https://sourceforge.net/projects/weka/>
5. Jason Brownlee. (2016 July 25) How to Use Classification Machine Learning Algorithms in Weka <https://machinelearningmastery.com/use-classification-machine-learning-algorithms-weka/>
6. [Pandas DataFrame and Keras](https://stackoverflow.com/questions/43876770/pandas-dataframe-and-keras) (n.d) <https://stackoverflow.com/questions/43876770/pandas-dataframe-and-keras>
7. Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V.and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>
8. tf.keras.Sequential (2020 April). <https://www.tensorflow.org/api_docs/python/tf/keras/Sequential>
9. tf.keras.callbacks.EarlyStopping. (2020 April)
10. <https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping>
11. tf.keras.optimizers.schedules.ExponentialDecay (2020 April)
12. <https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/ExponentialDecay>
13. tf.keras.optimizers.Adam (2020 April).
14. <https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam>
15. Vitaly Bushaev (2018 October). Adam — latest trends in deep learning optimization. <https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c>
16. Train and evaluate with Keras (2020 April).
17. <https://www.tensorflow.org/guide/keras/train_and_evaluate>
18. Sayak Paul (2018 October). Running Hyperparameter Sweeps to Pick the Best Model.
19. <https://www.wandb.com/articles/running-hyperparameter-sweeps-to-pick-the-best-model-using-w-b>
20. **Appendix A: TENSORFLOW CODE FOR ISOLET DATASET**

import tensorflow as tf

import sklearn.preprocessing as prep

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

#Assigning columns names to skip the headings from ISOLET Training DataSet and ISOLET Testing DataSet

names = ['f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7', 'f8', 'f9', 'f10', 'f11', 'f12', 'f13', 'f14', 'f15', 'f16', 'f17', 'f18', 'f19', 'f20', 'f21', 'f22', 'f23', 'f24', 'f25', 'f26', 'f27', 'f28', 'f29', 'f30', 'f31', 'f32', 'f33', 'f34', 'f35', 'f36', 'f37', 'f38', 'f39', 'f40', 'f41', 'f42', 'f43', 'f44', 'f45', 'f46', 'f47', 'f48', 'f49', 'f50', 'f51', 'f52', 'f53', 'f54', 'f55', 'f56', 'f57', 'f58', 'f59', 'f60', 'f61', 'f62', 'f63', 'f64', 'f65', 'f66', 'f67', 'f68', 'f69', 'f70', 'f71', 'f72', 'f73', 'f74', 'f75', 'f76', 'f77', 'f78', 'f79', 'f80', 'f81', 'f82', 'f83', 'f84', 'f85', 'f86', 'f87', 'f88', 'f89', 'f90', 'f91', 'f92', 'f93', 'f94', 'f95', 'f96', 'f97', 'f98', 'f99', 'f100', 'f101', 'f102', 'f103', 'f104', 'f105', 'f106', 'f107', 'f108', 'f109', 'f110', 'f111', 'f112', 'f113', 'f114', 'f115', 'f116', 'f117', 'f118', 'f119', 'f120', 'f121', 'f122', 'f123', 'f124', 'f125', 'f126', 'f127', 'f128', 'f129', 'f130', 'f131', 'f132', 'f133', 'f134', 'f135', 'f136', 'f137', 'f138', 'f139', 'f140', 'f141', 'f142', 'f143', 'f144', 'f145', 'f146', 'f147', 'f148', 'f149', 'f150', 'f151', 'f152', 'f153', 'f154', 'f155', 'f156', 'f157', 'f158', 'f159', 'f160', 'f161', 'f162', 'f163', 'f164', 'f165', 'f166', 'f167', 'f168', 'f169', 'f170', 'f171', 'f172', 'f173', 'f174', 'f175', 'f176', 'f177', 'f178', 'f179', 'f180', 'f181', 'f182', 'f183', 'f184', 'f185', 'f186', 'f187', 'f188', 'f189', 'f190', 'f191', 'f192', 'f193', 'f194', 'f195', 'f196', 'f197', 'f198', 'f199', 'f200', 'f201', 'f202', 'f203', 'f204', 'f205', 'f206', 'f207', 'f208', 'f209', 'f210', 'f211', 'f212', 'f213', 'f214', 'f215', 'f216', 'f217', 'f218', 'f219', 'f220', 'f221', 'f222', 'f223', 'f224', 'f225', 'f226', 'f227', 'f228', 'f229', 'f230', 'f231', 'f232', 'f233', 'f234', 'f235', 'f236', 'f237', 'f238', 'f239', 'f240', 'f241', 'f242', 'f243', 'f244', 'f245', 'f246', 'f247', 'f248', 'f249', 'f250', 'f251', 'f252', 'f253', 'f254', 'f255', 'f256', 'f257', 'f258', 'f259', 'f260', 'f261', 'f262', 'f263', 'f264', 'f265', 'f266', 'f267', 'f268', 'f269', 'f270', 'f271', 'f272', 'f273', 'f274', 'f275', 'f276', 'f277', 'f278', 'f279', 'f280', 'f281', 'f282', 'f283', 'f284', 'f285', 'f286', 'f287', 'f288', 'f289', 'f290', 'f291', 'f292', 'f293', 'f294', 'f295', 'f296', 'f297', 'f298', 'f299', 'f300', 'f301', 'f302', 'f303', 'f304', 'f305', 'f306', 'f307', 'f308', 'f309', 'f310', 'f311', 'f312', 'f313', 'f314', 'f315', 'f316', 'f317', 'f318', 'f319', 'f320', 'f321', 'f322', 'f323', 'f324', 'f325', 'f326', 'f327', 'f328', 'f329', 'f330', 'f331', 'f332', 'f333', 'f334', 'f335', 'f336', 'f337', 'f338', 'f339', 'f340', 'f341', 'f342', 'f343', 'f344', 'f345', 'f346', 'f347', 'f348', 'f349', 'f350', 'f351', 'f352', 'f353', 'f354', 'f355', 'f356', 'f357', 'f358', 'f359', 'f360', 'f361', 'f362', 'f363', 'f364', 'f365', 'f366', 'f367', 'f368', 'f369', 'f370', 'f371', 'f372', 'f373', 'f374', 'f375', 'f376', 'f377', 'f378', 'f379', 'f380', 'f381', 'f382', 'f383', 'f384', 'f385', 'f386', 'f387', 'f388', 'f389', 'f390', 'f391', 'f392', 'f393', 'f394', 'f395', 'f396', 'f397', 'f398', 'f399', 'f400', 'f401', 'f402', 'f403', 'f404', 'f405', 'f406', 'f407', 'f408', 'f409', 'f410', 'f411', 'f412', 'f413', 'f414', 'f415', 'f416', 'f417', 'f418', 'f419', 'f420', 'f421', 'f422', 'f423', 'f424', 'f425', 'f426', 'f427', 'f428', 'f429', 'f430', 'f431', 'f432', 'f433', 'f434', 'f435', 'f436', 'f437', 'f438', 'f439', 'f440', 'f441', 'f442', 'f443', 'f444', 'f445', 'f446', 'f447', 'f448', 'f449', 'f450', 'f451', 'f452', 'f453', 'f454', 'f455', 'f456', 'f457', 'f458', 'f459', 'f460', 'f461', 'f462', 'f463', 'f464', 'f465', 'f466', 'f467', 'f468', 'f469', 'f470', 'f471', 'f472', 'f473', 'f474', 'f475', 'f476', 'f477', 'f478', 'f479', 'f480', 'f481', 'f482', 'f483', 'f484', 'f485', 'f486', 'f487', 'f488', 'f489', 'f490', 'f491', 'f492', 'f493', 'f494', 'f495', 'f496', 'f497', 'f498', 'f499', 'f500', 'f501', 'f502', 'f503', 'f504', 'f505', 'f506', 'f507', 'f508', 'f509', 'f510', 'f511', 'f512', 'f513', 'f514', 'f515', 'f516', 'f517', 'f518', 'f519', 'f520', 'f521', 'f522', 'f523', 'f524', 'f525', 'f526', 'f527', 'f528', 'f529', 'f530', 'f531', 'f532', 'f533', 'f534', 'f535', 'f536', 'f537', 'f538', 'f539', 'f540', 'f541', 'f542', 'f543', 'f544', 'f545', 'f546', 'f547', 'f548', 'f549', 'f550', 'f551', 'f552', 'f553', 'f554', 'f555', 'f556', 'f557', 'f558', 'f559', 'f560', 'f561', 'f562', 'f563', 'f564', 'f565', 'f566', 'f567', 'f568', 'f569', 'f570', 'f571', 'f572', 'f573', 'f574', 'f575', 'f576', 'f577', 'f578', 'f579', 'f580', 'f581', 'f582', 'f583', 'f584', 'f585', 'f586', 'f587', 'f588', 'f589', 'f590', 'f591', 'f592', 'f593', 'f594', 'f595', 'f596', 'f597', 'f598', 'f599', 'f600', 'f601', 'f602', 'f603', 'f604', 'f605', 'f606', 'f607', 'f608', 'f609', 'f610', 'f611', 'f612', 'f613', 'f614', 'f615', 'f616', 'f617', 'ISOLET']

#Assigning Class\_names for prediction

class\_names = ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z']

#Using Pandas library to read ISOLET Train Data and Test Data

train = pd.read\_csv('.\ISOLET\_TrainData.csv', names=names, skiprows=1)

test = pd.read\_csv('.\ISOLET\_TestData.csv', names=names, skiprows=1)

#Droping ISOLET Column, which has Class Values from "A - Z" and Assigning Attribute values(Input) for Train and Test DataSet

Isolet\_train = train.drop("ISOLET", axis=1)

Isolet\_test = test.drop("ISOLET", axis=1)

#Assigning Label column "ISOLET" for Train and Test DataSet

ISOLET\_trainC = pd.get\_dummies(train.ISOLET)

ISOLET\_testCtst = pd.get\_dummies(test.ISOLET)

#Keras needs a numpy array as input and not a pandas dataframe. So, Converting Pandas Dataframe to numpy array. Other option is import keras\_pandas package to fit a pandas dataframe to keras

X= Isolet\_train.to\_numpy()          #Attribute Values of Train DataSet

Xtst=Isolet\_test.to\_numpy()         #Attribute Values of Test DataSet

C=ISOLET\_trainC.to\_numpy()          #Label Values of Train DataSe

CTst=ISOLET\_testCtst.to\_numpy()     #Label Values of Test DataSet

#Original range of each feature is (-1,1). Transforming features by scaling each feature to (0,1) .

scaler = prep.MinMaxScaler(feature\_range=(0,1))

#Fits transformer to X and returns a transformed version of X.

scaledX = scaler.fit\_transform(X)

#Scales features of Xtst according to feature\_range

scaledXtst = scaler.transform(Xtst)

#Sequential groups a linear stack of layers into a tf.keras.Model

model = tf.keras.models.Sequential([

  tf.keras.layers.Dense(X.shape[1], activation='relu'),

#First layer,which is an Input layer with 617 neurons

  tf.keras.layers.Dense(6238, activation='relu'),

#Second layer,which is a Hidden layer with 6238 neurons

  tf.keras.layers.Dense(26, activation='softmax')

#Third layer,which is an Output layer with 26 neurons

])

#Stops training when a monitored metric has stopped improving.

ES\_callback = tf.keras.callbacks.EarlyStopping(monitor='loss', min\_delta=1e-2, patience=10, verbose=1)

#Assigning learning rate=0.002

initial\_learning\_rate = 0.002

#A LearningRateSchedule that uses an exponential decay schedule. Meaning learning rate is modified automatically based on the decay\_rate to improve the training accuracy.

lr\_schedule = tf.keras.optimizers.schedules.ExponentialDecay(initial\_learning\_rate,decay\_steps=100000,decay\_rate=0.9999,staircase=True)

#Optimizer that implements the Adam algorithm.

optimizer = tf.keras.optimizers.Adam(learning\_rate=lr\_schedule)

#Model architecture

model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy ', tf.keras.metrics.RootMeanSquaredError()])

#Training our Model

model.fit(scaledX, C, epochs=100, callbacks=[ES\_callback], batch\_size = 162188)

#Evaluate the model on the test data using `evaluate`

test\_loss, test\_acc, test\_rms = model.evaluate(scaledXtst, CTst)

print("Tested Acc:",test\_acc\*100)

print("Root Mean Squared Error:",test\_rms\*100)

#Generate predictions (probabilities -- the output of the last layer) on new data using `predict`

prediction = model.predict(scaledXtst)

for i in range(10):

    arr = np.array(CTst[i])

    result = np.where(arr == 1)

    actual\_test\_label = int(result[0])

    print("Actual: %s -- Prediction: %s"%(class\_names[actual\_test\_label],class\_names[np.argmax(prediction[i])]))

print(model.summary())