F21DL Jamie McCulloch H00189648
Link to screenshots:
https://drive.google.com/open?id=1e_c31QiAvsGe4_hn_StUfSqNYQsXWfKR

Introduction

Using Microsoft Excel, I split the pixel values into different columns using the 'data to columns' option. This resulted in the fer2017-training dataset having 2305 columns and 28709 instances. To run this in weka with a larger heap size, I ran weka with the command: *java -Xmx4000m -jar weka.jar*
For the following experiments, I reduced the dataset size by 75%, to 7178 instances, in terms of fer2017-training.

Different types of trees

J48 works by taking a subset of the dataset to be the training set, and works out the decision tree of this. Then it applies this resulting decision tree to the dataset as a whole. It is an extension of the C4.5 algorithm, which has been regarded as the "a landmark decision tree program that is probably the machine learning workhorse most widely used in practice to date" [1]. This goes to prove that J48 is one of the leading machine learning algorithms, and should be one of the best in terms of classification. Through experimenting with different parameters, I found that changing the binary splits did not change the results, nor did pruning the tree. However changing the minimum weight of the nodes nodes to 15 and then 50, from when it was 2 did improve the accuracy.

User classifier allows users to construct their own decision tree, and has to be installed separately, I installed it through SourceForge, and linked with weka through their package manager. While this is useful, it becomes very difficult to work with a lot of attributes (as the user has to use the graph to find correlations), and also with each fold of cross validations, a new decision tree is built. This was time consuming to carry out and I ended up not completing this as it was too inconvenient with such a large dataset and I prioritised doing the other classifiers first. I did use the randomSubset filter and removed 75% of the attributes from the fer2017-testingnom.arff dataset, leaving me with 577 attributes.

RandomTree algorithm tests by constructs a tree that considers a amount of random K features at each node. It provides no pruning to the dataset. It uses a subset of attributes, by randomly splitting, which may result in noisy trees being constructed. By increasing the amount of randomly chosen attributes higher, the accuracy got lower. By increasing the minimum number of nodes permissible in the tree, the accuracy improved and the tree got smaller (as more leaves were attributed to one parent). The accuracy increased from 24.29% to 25.23% when the minimum was increased to 5. When increased to 15, the accuracy increased further to 25.75%.

Loading the fer2017-training dataset into weka, I then used the unsupervised attribute filter 'NumericToNominal' to change the class attribute emotion to nominal, so it can be classified correctly.

F21DL Jamie McCulloch H00189648

Link to screenshots:
https://drive.google.com/open?id=1e_c31QiAvsGe4_hn_StUfSqNYQsXWfKR

1. Variation in performance with size of the training and testing sets

I applied the 'RemovePercentage' filter to remove 75% of instances randomly to allow for quicker testing. I saved this file as *fer2017-training-nomreduced.arff.*

For Q6, by using the fer2017-testingnom.arff set, which had 7178 instances, I found 30% of this by removing 70% using the removePercentage filter, which was 2351 instances and saved this arff file through weka.
Then using the following command I combined this with the training set:

java weka.core.Instances append fer2017-testing2351.arff fer2017-training-nomreduced.arff > combinedsetq6.arff

Opening this new arff file into weka resulted in 9330 instances now in the training set. I then ran the classifier for this with cross validation set to 10, for each of the classifiers.

For Q7, using the same method as in Q6, I found 70% of the fer-testing dataset by removing 30%, and then appended with the fer-trainingreducednom.arff file which resulted in the new training set having 12202 instances. This was then ran with a cross validation of 10 for each classifier.

Classifier	**Q3 Accuracy**	**Q5 Accuracy**	**Q6 Accuracy**	**Q7 Accuracy**
J48	**25.51**	**26.3**	**29.39**	**30.67**
RandomTree	**24.29**	**24.1**	**27.58**	**29.23**
MultilayerPerceptron	**22.38**	**N/A**	**28.65**	**26.89**

Table 1 - Accuracies of each classifier with original fer2017-training.arff file

As evident from Table 1 above, the accuracy tends to increase with each given question, with the best accuracy coming from combining 70% of the testing set with the training set and running the classifiers on the resulting file. This may mean that, with a higher amount of instances, the accuracy increases, however with the time taken to run these classifiers with such large datasets makes it a disadvantage.

Link to screenshots:
https://drive.google.com/open?id=1e_c31QiAvsGe4_hn_StUfSqNYQsXWfKR

Q3, 6, and 7 all use the training set to evaluate the classifier, and, especially 6 and 7, are more accurate than Q5 which uses the testing set. This goes to prove that cross validation may evaluate the data better, compared to a test set.

I then ran all of these on the fer2017-happy-training.arff file, and fer2017-happy-testing.arff as the testing set. As expected, this was a much higher accuracy (due to only 2 classifications rather than 7). When I ran this with the classifiers, I ran with 75% of the instances removed (leaving 7177 instances) and saved this file as fer2017-happy-trainingnom.arff.

Classifier	**Q3 Accuracy**	**Q5 Accuracy**
J48	**69.5%**	**69.1%**
RandomTree	**67.32%**	**66.77%**
MultilayerPerceptron	**71.31%**	**N/A**

Table 2: Accuracies for fer2017happy

With regards to the neural net approach to this, I used the MultilayerPerceptron classifier with the following parameters:

MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a

I ran the same tests as before, however I reduced the attributes and instances as it was taking too long for the classifier to run with over 1000 instances/attributes.

Datasets were modified from the fer2017-trainingnomreduced.arff file;
- Q3: 577 attributes, 143 instances
- Q6: 251 attributes, 583 instances
- Q7: 251 attributes, 610 instances

Overfitting is the term given to when analysis is too specific to a certain segment of data, and so causes issues in terms of classifying things in the future, as things may be incorrectly classified. By varying the testing/training sets and running the classifiers on the different sizes of datasets, this gave a good indication as to what classifiers avoid overfitting and what don't.
As the RandomTree classifier may result in noisy trees, this increases the likelihood of overfitting. To counter this, we must produce rules that are not perfect within in the training set, so it is not always right [1].
Using the Multilayer Perceptron, and neural networks, with backpropagation can result in overfitting, especially if the network is much larger than what is needed. There are ways to

Link to screenshots:
https://drive.google.com/open?id=1e_c31QiAvsGe4_hn_StUfSqNYQsXWfKR

counter this, like early stopping which is used to automatically find out how many passes through the dataset should be used [2]. Another would be weight decay, which tries to limit the amount of influence of irrelevant connections by penalising large weights that do not contribute large reduction in errors [1].

2. Variation in performance with change in the learning paradigm (Decision trees vs Neural Networks)

Based on the results from above, in tables 1 and 2, it is hard to tell what is the better option to go for. It seems that neural networks need to be smaller, to work as effectively, otherwise it is too time consuming. However from table 1, it is not more accurate than J48 classifier, at all. This is perhaps not surprising as J48 is one of the most reliable and used decision tree classifiers but is still a shock. That being said, I have only run one neural net classifier, and three decision tree classifiers so it may be unfair to just this so heavily.

3. Variation in performance with varying learning parameters in Decision Trees

As mentioned earlier I modified the learning parameters of a the J48 classifier, by enabling/disabling binary splits, enabling tree pruning, and changing the minimum number of objects permissible. By enabling binary splits and disabling pruning, the accuracy did not change, and did not take any longer/shorter to run the classifier. However changing the minimum number of objects did change the accuracy, and increased it.

MinNumObj	2	15	50
Accuracy (%)	26.3	27.75	28.5

Table 3: changes in J48

This increase in accuracy is quite negligent however it does increase the accuracy, and so is worth investigating. The classifier also ran in the same time as each other, meaning there was no disadvantage to increasing this.

While using the RandomTree classifier, I changed the *k* value of the classifier, from the default of 0 to 5, which had a detrimental effect on the accuracy, causing it to decrease from 24.29% to

Link to screenshots:
https://drive.google.com/open?id=1e_c31QiAvsGe4_hn_StUfSqNYQsXWfKR

23.4%. This could be due to investigating more nodes causes the classifier to incorrectly classify more, due to it being more thorough. It also took longer to run (as it was checking more random nodes).
However by changing the minimum number of total weight of each node from 1 to 5 caused the accuracy to increase from 24.29% to 25.23%.

4. Variation in performance with varying learning parameters in Neural Networks

Given that I am only running the one classifier - MultilayerPerceptron, I changed the momentum of the network, increasing it from 0.2 to 10, and running it on the fer2017happy.arff as shown in table 2 to see if the accuracy increases, or if the time taken run decreases. With changing the momentum from 0.2 to 10, I noticed an increase in accuracy from 71.31% up to 73.54% when run. It took about the same amount of time to run (about half an hour) but does increase the accuracy of instances.

5. Variation in performance according to different metrics

As seen within the screenshots folder of my project, I screenshotted every confusion matrix/detailed accuracy section of each classifier. I found that in general, the J48 classifier had a higher True Positive Rate (**TP Rate**) meaning that it guesses the correct class x% of times. This in turn means the higher the TP Rate, the higher the accuracy. So, the higher the False Positive Rate (**FP Rate**) is, which is the amount of times the classifier predicts the class when it is not correct, the lower the accuracy is. The average TP rate of J48 on Q3 is 0.255 which is still low, but is better than Q3 using the MultilayerPerceptron classifier, which only got an average of 0.224. The RandomTree was inbetween, doing better than the MultilayerPerceptron, with an average of 0.243 also when run on Q3. As you can see, this is consistent with the accuracy of these classifiers.
Looking at the **Precision** of these classifiers, which is how many times a class is guessed is it correct, it can be seen that these closely mirror the TP Rate, in terms of the classifiers.

[1] Witten, I.H., Frank, E., Hall, M.A. and Pal, C.J., 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.
[2] Early Stopping - Deeplearning4j: Open-source, Distributed Deep Learning for the JVM, accessed on 25/11/17 https://deeplearning4j.org/earlystopping