Abstract—WEKA is an important tool used by affective computing researchers for experimentation. WEKA is useful because it has an intuitive user interface and is therefore easily accessible to domain specialists that are not proficient with machine learning, but its speed and memory requirements limit its usefulness in real time systems. Other modifications to WEKA have not focused on streamlining WEKA with real time systems in mind. Our goal is to modify WEKA so that it requires less memory space and classifies data in less time than the standard version of WEKA. Our hope is that our modifications can support the use of WEKA in real time affective computing applications. Within this paper we explain our proposed methods and some preliminary results obtained for benchmarking purposes. We have identified two areas of WEKA’s process that can potentially be modified to yield improvements. By collecting pertinent information on WEKA’s performance across four different data sets on two different systems, we have create a series of benchmarks which we hope to improve upon through our modifications. We have also identified how to collect similar metrics for a third system, but have not yet done so. Future work includes making the modifications to WEKA and collecting new performance information and comparing those results to our benchmarks.

I. INTRODUCTION

Affective computing concerns itself with developing systems that can detect, express, and ultimately feel emotions, or affective states [1]. Applications vary, but most include interfaces which take the user’s mood into account. Machine learning techniques are typically employed to perform classification tasks and determine the affective state of the user. The ultimate goal of most affective computing research is a real-world application that performs classification in real time. WEKA (Waikato Environment for Knowledge Analysis) is a machine learning workbench written in Java and published as open source software which is often used by the affective computing community [2], [3]. The ease of using WEKA for experimentation is one of the prime reasons for its ubiquity in the affective computing community, but it is not as efficient as custom code when it comes to a real time system. A real time system requires quick classification in order to respond in an appropriate time frame. In addition, many affective computing applications may have memory requirements that need to be met. As a tool primarily used for research, classification in WEKA is currently neither fast enough nor necessarily small enough for real time system requirements.

To that extent, we wish to improve WEKA’s performance with the hopes of making WEKA suitable for real time systems. We plan on doing this through two ways. First, we will speed up classification in WEKA. We will not be modifying the implementations of classification algorithms themselves; rather, we will be looking for ways to improve WEKA’s alacrity from a systems level perspective. Second, we will reduce the space taken by WEKA during classification. Once again, this will be done from a systems level perspective as opposed to modifying the implementations of classification algorithms within WEKA. Specifically, we will identify areas where we can modify system calls made by WEKA. Finally, in order to make our extension of WEKA generalizable we will use four different datasets of interest to the affective computing community in our experimentation.

This paper reports results for the basic version of WEKA, without modification, and outlines further steps.

II. RELATED WORK

WEKA is designed to bring different machine learning techniques under a common interface so that they can be easily applied to real world datasets in a consistent manner. The main focus of WEKA was to provide a working environment for the domain specialists, rather than the machine learning experts [4], [5]. A description of WEKA is presented in [6]. These days, WEKA has widespread acceptance in both academia and business and has been downloaded more than 1.4 million times since being placed on SourceForge in April 2000 [5]. Other scholars in affective computing have used WEKA for implementing their machine learning algorithms [7], [8], [9].

Wong and Cho used WEKA for developing a new classifier for face emotion recognition. They benchmarked their proposed method against well-known classifiers such as support vector machines (SVM), C4.5 Trees (C45), and Gaussian radial basis function networks (RBF network), which were all performed from the WEKA package [10]. Francke, Ruiz-del-Solar and Verschae developed a multimodal classifier for hand gesture detection and recognition using the WEKA package [11]. Baumgartner and Serpen proposed a tool, LEET (Large Experiment and Evaluation Tool) to
accomplish some tasks that are difficult in standard WEKA interface. This includes tracking execution time, calculating diversity measures and comparing different classifiers in multiple experiments and summarizing characteristics of datasets [12]. Castellano et al. presented an approach for the recognition of acted emotional states based on the analysis of body gestures [13]. For classification they used WEKA. They tested different WEKA settings for 1NN, decision tree and Bayesian Network to find the best performance for each of them [13]. Gunes and Piccardi used WEKA for fusing face and body gestures for machine recognition of emotions [14].

Despite much effort, creating a real time system that can reasonably handle very large datasets is a problem. Common issues include an unreasonably long training or testing time or running out of memory. Scholars report the time of training and testing process in their systems to compare the effectiveness of their proposed machine learning algorithms in WEKA to other methods.

Wong and Cho tried to reduce training time by encoding the feature into hierarchical relationship information such that it allows the system to “reuse” knowledge and thus generalize with less training. They found that when put into a hierarchical order, low-level features may take a large amount of time and memory to learn, but high-level features take a smaller amount of time and memory to learn [10]. Ours is not the only project that has focused on improving the speed of WEKA. WEKA Parallel was an effort to streamline k-fold cross validation in WEKA [15]. K-fold cross validation is a prime candidate for parallelization as it is easily parallelizable and is used in a multitude of machine learning tasks. The authors manage to decrease the total running time dramatically by parallelizing on multiple remote machines. WEKA Parallel in turn has had improvements made to it. In his paper Xin Zuo describes work done to improve the performance of WEKA Parallel in distributed systems, such as fault tolerance and load balancing [16]. His efforts served to increase the reliability of WEKA Parallel as well as the speed.

III. METHOD

We plan to make two modifications to WEKA. The first is to decrease the amount of memory space that WEKA uses and the second is to decrease the amount of time that WEKA takes to complete a classification procedure. We will be testing our improvements on three systems. The first is an EEEPC running Ubuntu 10.04 with Linux Kernel 2.6.32-33 along with 2.0GB of RAM and two Intel Atom D525 dual cores with 1.80GHz each. The second is a Linux desktop running Red Hat 6.3 with Linux Kernel 2.6.32-279 along with 4.0GB of RAM and one dual core AMD Opteron 1210 processor with 1.0GHz. The third system is a laptop running Windows Vista Enterprise 32-bit with 4 GB of RAM and an Intel Core 2 Duo with 2GHz. The RAM and OS dictate the size of processes we are able to run. RAM is limited in most affective computing applications so we want to reduce the amount of RAM necessary to run WEKA. Processor speed and the number of cores have an effect on the speed of affective computing applications. Once again, an affective computing application will have access to less processing power because of physical size restrictions. As we want to increase speed as much as possible it is important that we test our modifications on several different combinations of processor speed and number of processors.

We have chosen these three systems because they represent a good variety in terms of RAM, OS, and processor speed. The EEEPC is the most similar to a system that would be used in an affective computing application as it has a smaller amount of RAM along with less processor speed. The Linux desktop is useful to compare to the EEEPC because it has a similar operating system but a different amount of RAM, processor speed, and number of processors. The raw processing power of each core is less than the EEEPC, as is the number of processors, but the Linux desktop has more RAM than the EEEPC. The Windows laptop system has a different processor than both the other systems, but has more raw processing speed. However, the number of processors is actually less than the EEEPC, while being the same as the Linux desktop. Finally, there is more RAM available on the Windows machine than the EEEPC, which will allow for a lesser restriction on memory which may lead to less page faults. The different operating system could have an effect on either overall speed or memory available which is the primary allure to using the Windows laptop.

IV. RESULTS

A. Run-Time

Run time results for multiple runs of each dataset and classifier on the EeEPc and Desktop can be seen on logarithmic scales in figures 1 and 2. A logarithmic scale was required due to the significantly higher run time associated with the neural network algorithm, particularly on the larger dimension datasets (AVEC, MacMastor(MM)).

There are similar patterns for each Dataset, determined by the classifier. KNN is faster than the Decision Tree, which is faster than SVM, that is much faster than neural networks.

As the classifiers take approximately the same amount of run time in both the YouTube (YT) and AMI Corpus (AMI), which have the same feature dimension, but differing numbers of those features, we can infer that run time is dependant not only on classifier, but feature dimension.

Further discussion, particularly related to neural networks, are left to the next section.
B. Memory Usage

Memory usage for each classifier and data set on the EeePC and Desktop can be seen in Figures 3 and 4. These figures show the maximum memory usage in MB on a linear scale. A Logarithmic scale was not required as there was not so large a discrepancy as with run time. It should be noted that the upper limit of the two graphs are different. The EeePC has 2GB RAM, and uses at most 1.6GB for WEKA, while the Desktop has 4GB RAM, and uses just over 1.8GB for WEKA.

Again there are classifier dependant patterns, with (broadly) neural networks taking more memory than SVMs, and KNNs taking about as much space as Trees, both of which are less than SVMs. However here the Dataset plays a more noticeable role. Because of higher dimension data, and the resultant need to have those features accessible, the datasets can be ordered by memory usage and dimension as AVEC, MacMastor, YouTube, AMI. The fact that AMI and YouTube are not identical here indicates that the number of features as well as their dimension plays a part, however it seems that dimension is still the most important, as the difference is small.

Finally, it should be noted that, while both tasks clearly need the same amount of data available, the EeePC’s physical restrictions on RAM cause its memory usage to appear smaller. EeePC memory usage is however, at most, 80% of total physical RAM, while Desktop memory usage reaches only 50%.

This may be one of the factors which causes the Desktop
Although page faults are not one of our metrics, because they help to elucidate the relationship between memory usage and run time, they are included here in table 1.

On average, the page faults for the EeePC are twice those of the desktop. However this does not hold true for all classifier,dataset pairings. In the case of the AMI,KNN row for example, in table 1, the number of page faults is almost identical, whereas the MacMastor,SVM line shows a 5x increase. This is because of the amount of memory each classification algorithm actually requests, and indicates that the improvement of memory management for WEKA will improve the run time due to a decrease in page faults.

V. DISCUSSION

Our preliminary results yield some information about WEKA’s performance classifying four different affective computing data sets. Firstly, it is important to note that all the independent variables (hardware, data set, and classifier) affect our metrics (runtime and memory usage) in different ways, and to greater or lesser extents. For all datasets and machines, the
neural network takes the longest time to run: well over 3 hours for the largest dataset (AVEC). This is due to algorithm itself; pseudo-code for a multilayer perceptron can be seen below:

```plaintext
// forward pass
O = neural−net−output(network, e)
T = teacher output for e
Calculate error (T − O) at output units

// backward pass
For nodes from hidden to output layer:
    Compute delta_w_h for all weights
For notes from input to hidden layer:
    Compute delta_w_i for all weights
Update the weights in the network
Until all e (sufficiently) classified
Return the network
```

The time taken is due to the many iterations required, and the (unnecessarily) sequential processing of each feature vector. This constant re-reading of the training vectors also increases the memory usage for this algorithm. This is also a problem for the support vector machine, which was developed as an alternative to neural networks in the 90s [cortes 1997]. However because of the use of kernels in SVMs fewer training cycles are required.

Accepting the decrease in performance due to classifier, the next most obvious difference in our results is that due to the hardware available. The EeePC, though it has a similar processor to the Linux desktop, takes approximately double the time to complete classification. Interestingly, this seems to be due to the halved memory on the system, as there are far more page faults \(\text{ref figure here}\).

This is an important finding for the affective community, who have never performed a systematic test of their tools, and suggests that they should focus on memory size as well as increasing clock speed when attempting to design real time systems.

Finally, the changes based on data sets. Two of our data sets had features extracted using PCA, with dimension=20. This resulted in the smallest feature vectors, and so the shortest run times and smallest memory footprint. However, this came at the cost of expensive feature extraction. If a real time affective system can rely on offline feature extraction, this may be a solution. But we will not focus on feature extraction as it falls outside the scope of this work.

The other datasets had larger feature vectors; X and Y respectively. And this proved to be a key indicator of efficiency. However, this is also an area where concurrency and locality may be exploited, as seen in our future work.

### VI. Future Work

There are two main areas of the WEKA code that we have chosen to modify: the training of classifiers and reading and conversion of data into features. We believe that we can improve memory usage by modifying how data is read and converted before being sent to the classifiers. We are also going to parallelize the training segment of four different classifiers. By identifying areas that can be parallelized in the SMO classifier, the KNN classifier, the NN classifier, and the J48 classifier, we hope to increase the throughput of training and therefore decrease overall run time.

Before WEKA can begin training a classifier, it must first read the data file and convert it to an ARFF file. We have identified this process as a location where it is possible to reduce the amount of memory space necessary for WEKA. In order to measure the amount of space that WEKA uses in Linux overall, we have decided to use the time system call with the verbose option to collect information about the maximum resident set size and number of page faults, both of which are supplied by the verbose option of the time system call. To get this same information on our Windows system, we have written a C++ program that will grab details from a process using the GetProcessMemoryInfo function. We can get both the peak working set size (the equivalent of maximum resident set size on the Linux system) from the PeakWorkingSetSize member and the number of page faults.

#### TABLE I

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Classifier</th>
<th>Number of Page Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Eee PC</td>
</tr>
<tr>
<td>A MI</td>
<td>SVM</td>
<td>32294.40</td>
</tr>
<tr>
<td>A MI</td>
<td>K-NN</td>
<td>15652.47</td>
</tr>
<tr>
<td>A MI</td>
<td>Tree</td>
<td>18536.13</td>
</tr>
<tr>
<td>A Vec</td>
<td>SVM</td>
<td>39854.50</td>
</tr>
<tr>
<td>A Vec</td>
<td>K-NN</td>
<td>39732.00</td>
</tr>
<tr>
<td>A Vec</td>
<td>Tree</td>
<td>40460.10</td>
</tr>
<tr>
<td>M M</td>
<td>SVM</td>
<td>53421.00</td>
</tr>
<tr>
<td>M M</td>
<td>K-NN</td>
<td>37520.80</td>
</tr>
<tr>
<td>M M</td>
<td>Tree</td>
<td>38182.10</td>
</tr>
<tr>
<td>Y T</td>
<td>SVM</td>
<td>22954.40</td>
</tr>
<tr>
<td>Y T</td>
<td>K-NN</td>
<td>16301.50</td>
</tr>
<tr>
<td>Y T</td>
<td>Tree</td>
<td>20150.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Desktop</td>
</tr>
<tr>
<td>A MI</td>
<td>SVM</td>
<td>15652.47</td>
</tr>
<tr>
<td>A MI</td>
<td>K-NN</td>
<td>11700.13</td>
</tr>
<tr>
<td>A MI</td>
<td>Tree</td>
<td>14900.13</td>
</tr>
<tr>
<td>A Vec</td>
<td>SVM</td>
<td>20804.10</td>
</tr>
<tr>
<td>A Vec</td>
<td>K-NN</td>
<td>19721.80</td>
</tr>
<tr>
<td>A Vec</td>
<td>Tree</td>
<td>20557.50</td>
</tr>
<tr>
<td>M M</td>
<td>SVM</td>
<td>16771.60</td>
</tr>
<tr>
<td>M M</td>
<td>K-NN</td>
<td>14675.50</td>
</tr>
<tr>
<td>M M</td>
<td>Tree</td>
<td>15709.00</td>
</tr>
<tr>
<td>Y T</td>
<td>SVM</td>
<td>14591.00</td>
</tr>
<tr>
<td>Y T</td>
<td>K-NN</td>
<td>11980.40</td>
</tr>
<tr>
<td>Y T</td>
<td>Tree</td>
<td>14287.60</td>
</tr>
</tbody>
</table>

The time taken is due to the many iterations required, and the (unnecessarily) sequential processing of each feature vector. This constant re-reading of the training vectors also increases the memory usage for this algorithm. This is also a problem for the support vector machine, which was developed as an alternative to neural networks in the 90s [cortes 1997]. However because of the use of kernels in SVMs fewer training cycles are required.
faults from the PageFaultCount member that belong to the PROCESS_MEMORY_COUNTERS structure returned by the GetProcessMemoryInfo function. The peak working set size of a process is a measurement of the largest amount of memory that process used during its run time. If we can reduce this number, it means that the overall memory needed for that process has been reduced. We would be happy with a 10% decrease in the peak working set size, though we will aim for 25%. Reducing the number of page faults does not indicate a reduction in memory needed. Less page faults will actually mean less time spent looking up pages, which will result in a faster process. As such, it is not one of our two main metrics, but it is an interesting measure to keep track of nonetheless, and it is relatively trivial to do so as long as we are looking at the peak working set size.

Training WEKA classifiers is currently done serially and modifying this section of WEKA to run in parallel should increase the overall speed classification. To determine the amount of time that WEKA takes to classify our data sets in Linux, we will be using the verbose option of the time function as well. The verbose option of the time function will supply us with an elapsed time measure which corresponds to the amount of time it took to run the process from beginning to end. On the Windows system we will use a similar metric which we will access through the GetProcessTimes function. The GetProcessTimes function will return a pointer to a FILETIME structure that contains the creation time of the process as well as a pointer to a FILETIME structure that contains the exit time of the process. From there it is just a matter of calculating the difference to obtain a comparable measurement to the elapsed time gotten from the verbose option of time. This metric is the most accurate measurement of time taken by a process on both machines, but it is an interesting metric to keep track of nonetheless, and it is relatively trivial to do so as long as we are looking at the peak working set size.

We would be happy with a 10% decrease in the peak working set size, though we will aim for 25%. Reducing the number of page faults does not indicate a reduction in memory needed. Less page faults will actually mean less time spent looking up pages, which will result in a faster process. As such, it is not one of our two main metrics, but it is an interesting measure to keep track of nonetheless, and it is relatively trivial to do so as long as we are looking at the peak working set size.

Training WEKA classifiers is currently done serially and modifying this section of WEKA to run in parallel should increase the overall speed classification. To determine the amount of time that WEKA takes to classify our data sets in Linux, we will be using the verbose option of the time function as well. The verbose option of the time function will supply us with an elapsed time measure which corresponds to the amount of time it took to run the process from beginning to end. On the Windows system we will use a similar metric which we will access through the GetProcessTimes function. The GetProcessTimes function will return a pointer to a FILETIME structure that contains the creation time of the process as well as a pointer to a FILETIME structure that contains the exit time of the process. From there it is just a matter of calculating the difference to obtain a comparable measurement to the elapsed time gotten from the verbose option of time. This metric is the most accurate measurement of time taken by a process on both machines, but it is an interesting metric to keep track of nonetheless, and it is relatively trivial to do so as long as we are looking at the peak working set size.

We would be happy with a 10% decrease in the peak working set size, though we will aim for 25%. Reducing the number of page faults does not indicate a reduction in memory needed. Less page faults will actually mean less time spent looking up pages, which will result in a faster process. As such, it is not one of our two main metrics, but it is an interesting measure to keep track of nonetheless, and it is relatively trivial to do so as long as we are looking at the peak working set size.

Training WEKA classifiers is currently done serially and modifying this section of WEKA to run in parallel should increase the overall speed classification. To determine the amount of time that WEKA takes to classify our data sets in Linux, we will be using the verbose option of the time function as well. The verbose option of the time function will supply us with an elapsed time measure which corresponds to the amount of time it took to run the process from beginning to end. On the Windows system we will use a similar metric which we will access through the GetProcessTimes function. The GetProcessTimes function will return a pointer to a FILETIME structure that contains the creation time of the process as well as a pointer to a FILETIME structure that contains the exit time of the process. From there it is just a matter of calculating the difference to obtain a comparable measurement to the elapsed time gotten from the verbose option of time. This metric is the most accurate measurement of time taken by a process on both machines, and as such we have chosen to use it as the primary way to evaluate our improvement of WEKA’s speed. As with the memory size, we would be happy with a 10% decrease in time taken to classify a dataset, but we will aim for an improvement of 25%.

REFERENCES