Weka on .NET

Matic Potočnik, Gregor Majcen

***Abstract*—This paper will describe and benchmark several approaches to using Weka on the .NET Framework.**

***Index Terms*—Weka, .NET, benchmark, machine learning.**

I. Introduction

THIS paper will provide the reader with a tutorial and performance notes on using the Weka data-mining library on the Microsoft .NET framework. Weka originally runs on the Java Virtual Machine (JVM), but there are methods which allow running Java code on .NET. In this paper we will describe several approaches and benchmark the performance of each approach on multiple machine learning algorithms. We will also list some possible use-cases of Weka on .NET. For a brief description of the technologies used, see Appendix A.

January 10, 2012

II. Java on .NET

In this section we will describe a few methods for using Java code on the .NET Framework. The text presumes usage of Microsoft Windows and that Java JDK, .NET Framework, Weka, IKVM, C# and F# compiler and Mono are already installed and that the correct directories have been added to the system PATH.

The exact versions of software used for generating the results in this paper are:

- Microsoft Windows 7 Professional x64
- Java 1.6.0_26
- Microsoft.NET Framework v4.0.30319
- Weka 3.6.6
- Microsoft Visual C# 2010 Compiler 4.0.30319.1
- Microsoft F# Compiler 2.0.0.0
- IKVM 0.46.0.1
- Mono 2.10.8

A. IKVM directly running Java code on .NET

This approach is the simplest if we just want to run Java on the .NET Framework, but its practical use is somewhat questionable as you cannot mix Java and .NET code this way. That means you cannot use any of the .NET libraries and code. But if you have the .class files precompiled you can run them on .NET without having Java installed on your system.

All the user needs to do to use this approach is to use the command-line command `ikvm` instead of `java` when running a program from the console. IKVM automatically accepts and uses additional Java libraries and no additional action is needed.

Weka example

```
javac -cp weka.jar;. Program.java
ikvm -cp weka.jar;. Program
```

The `ikvm` command accepts all the command-line switches `java` does, but some don't necessary do anything. For instance – memory switches such as `-xmx1024m` get ignored, as the .NET handles memory differently and does not hard-limit heap space the same way the JVM does. IKVM alerts us if some option is ignored or unsupported.

B. Using a IKVM-generated .dll with .NET/Mono

For this approach we need a Java library inside a `.jar` package. We can then proceed to generate a `.dll` library using `ikvmc`. The `ikvmc` command has several options, but the most important for our use is the `-target` parameter, which can take the following values:

- **exe** – generates an .exe that runs in a command window
- **winexe** – generates an .exe for GUI applications
- **library** – generates a .dll
- **module** – generates a .netmodule

We are going to use the `library` value, which generates a .dll which can then be referenced by .NET Framwork and Mono code. In C# programs we use the `using` directive and the `open` directive is used in F#.

Weka example

```
ikvmc -target:library weka.jar
```

After `ikvmc` has generated the .dll for us, we can use Visual Studio or the command-line compiler to compile our program with the .dll.

Weka example (.NET)

```
csc Program.cs -r:weka.dll,IKVM.OpenJDK.Core.↩
    dll,IKVM.Runtime.dll
Program.exe
```

Weka example (Mono)

```
dmcs Program.cs -r:weka.dll,IKVM.OpenJDK.Core.↩
    dll,IKVM.Runtime.dll
mono Program.exe
```

C. Using WekaSharp on .NET/Mono

Here a pre-built F# wrapper around Weka is used. It also uses IKVM, but there is also some native F# code, which simplifies Weka programming in F#. In this paper we've used Weka directly with the Weka.dll provided with WekaSharp, and didn't explore the options and benefits the wrapper provides.

Weka example (F#)

```
fsc Program.fsx -r:weka.dll -r:IKVM.OpenJDK.↩
    Core.dll -r:IKVM.Runtime.dll
Program.exe
```

III. Benchmarking

In this section we will discuss the benchmarking setup and methods used to generate the results.

A. Overview

We will use the following Weka algorithms:

- weka.classifiers.bayes.NaiveBayes
- weka.classifiers.trees.RandomForest
- weka.classifiers.trees.J48
- weka.classifiers.rules.ConjunctiveRule
- weka.classifiers.functions.MultilayerPerceptron
- weka.classifiers.functions.SMO

Measurements will be made for these languages/platforms:

- **java** – Java with Weka.jar
- **javaikvm** – Java with Weka.jar on .NET via IKVM
- **net** – C# with IKVM-generated Weka.dll on .NET
- **mono** – C# with IKVM-generated Weka.dll on Mono
- **fsharp** – F# with WekaSharp's Weka.dll on .NET

B. Benchmarking code

For benchmarking purposes, a functionally equivalent short benchmarking program was written in Java, C# and F#.

Benchmark (pseudocode)

```
read number of runs from args (default 1)
read classifier name from args (default All)
read dataset name from args (default "train")

for 1 to runs {
  measure reading time {
    read training data
    set the last attribute as class
    read test data
    set the last attribute as class
  }

  measure building time {
    forall classifiers {
      build classifier
    }
  }

  measure classifying time {
    forall classifiers {
      forall testsamples {
        predict testsample class
      }
    }
  }
}

print classifier name
print read time
print build time
print classify time
print total time
```

The only difference between implementations, that is worth noting was the use of `System.nanoTime()` in Java and the `Stopwatch` class in C# and F# for measuring time, but we've tested both approaches on .NET and found that there is no noticeable difference.

C. Code correctness

One of the concerns we had was whether or not the Weka library behaviour was still the same on .NET, as it was on the JVM, and also if our programs were really equivalent.

To test this we ran all the programs and checked if the classification accuracy is the same on all three. The test was run on all the selected algorithms. Below is a sample output of one of these runs. As you can see there are some additional trailing zeroes with the last line, which is the F# output, but as we've discovered this was due to typing, not due to numerical error. The other examples remained of the type `java.lang.Float`, but F# has a different type-sytem, which raised a type-error when doing arithmetic with mixed types, so we had to cast the numbers to F#'s `float` type which has a slightly different behaviour when converting float to string. Also, this difference appears in the calculation of the percentage – the classification results are exactly the same on all runs and so we have concluded that the code is correct.

Sample output

```
ConjunctiveRule 61004 out of 100000 correct (61.004%)
ConjunctiveRule 61004 out of 100000 correct (61.004%)
ConjunctiveRule 61004 out of 100000 correct (61,004%)
ConjunctiveRule 61004 out of 100000 correct (61.004%)
ConjunctiveRule 61004 out of 100000 correct (61.004000%)
```

D. Benchmarking script

To simplify and speed-up the benchmarking process we have constructed a script which runs all the benchmarks and saves the results into a file. The script was written in Windows Batch. The script allowed us to set the number of runs, desired algorithms, the name of the dataset and which languages/platforms are included in the benchmarked.

Benchmarking script (pseudocode)

```
set number of runs
set algorithms
set datasets

forall methods {
  forall algorithms {
    forall datasets {
      compile and run programs
    }
  }
}

save results
```

The script outputs the results in a format that is useful almost as-is in Mathematica for further analysis and graph plotting. Here is an example of a small Java run:

Result sample

```
langs={java};
java =
{"NaiveBayes",531,20,202,753};
java =
{"RandomForest",488,60,350,898};
java =
{"SMO",478,189,319,986};
java =
{"J48",469,29,21,519};
java =
{"MultilayerPerceptron",491,12814,1578,14883};
java =
{"ConjunctiveRule",465,22,17,504};
```

IV. Results

In this section we will present the benchmarking results.

A. Dataset

We used a bioinformatics dataset which was used as a part of a challenge/homework in an bioinformatics class on FRI [**?**]. The dataset was in CSV format (Comma-Separated Values), and Weka uses the ARFF (Attribute-Relation File Format), so we used an online CSV to ARFF converter to convert it into an appropriate format. We then then took a sample of only 500 rows from the training data. This was done because some machine-learning algorithms, such as SMO and MultilayerPerceptron, don't work well with multi-valued attributes, and we wished to run all classifiers on the same dataset. The testing data on which we executed the classification was 100000 rows long. Here is the ARFF header of the dataset in question:

Dataset header

```
@relation train

@attribute DNA {A,T,C,G}
@attribute gene_structure_on_plus {_,O,I,E}
@attribute gene_structure_on_minus {O,_,I,E}
@attribute UTR&TSS {_,5utr,tss,3utr}
@attribute expression {_,H}
@attribute histone_binding {NN,N,Z,P,PP}
@attribute histone_modifications {Z,P,N}
@attribute nucleosome_occupancy {Z,P,N}
@attribute polII_presence {Z,N,P}
```

B. Joint results

We will begin the presentation of the results with the graph of the total time for all algorithms. We ran the benchmark 10 times on the above dataset for each of the algortihms and each of the language/platform combinations.

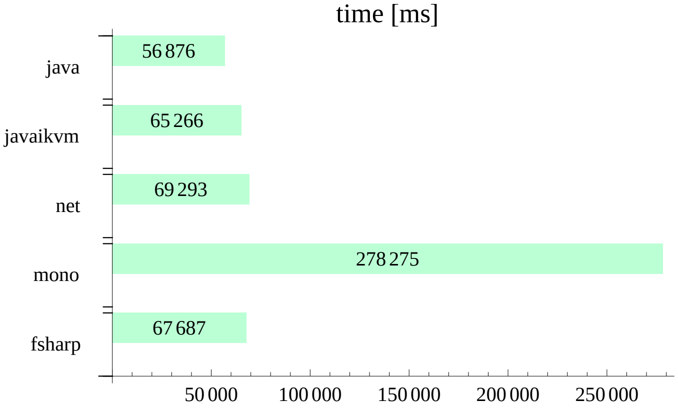

Fig. 1. Joint results

As we can see from the graph Java is the fastest, the programs on .NET Framework ran at aproximately the same speed, and the program ran on Mono was several times slower.

Next we will look at the joint performance across all algorithms for reading, classifier building and classifier execution.

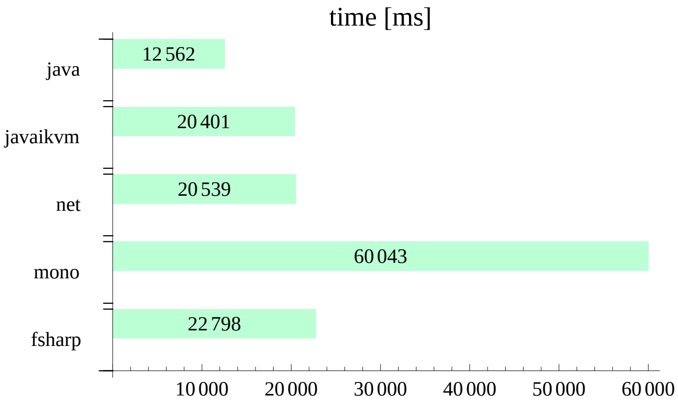

Fig. 2. Reading performance results

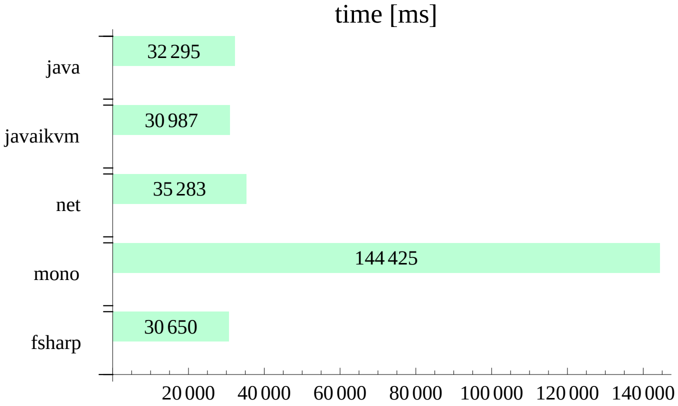

Fig. 3. Classifier building performance results

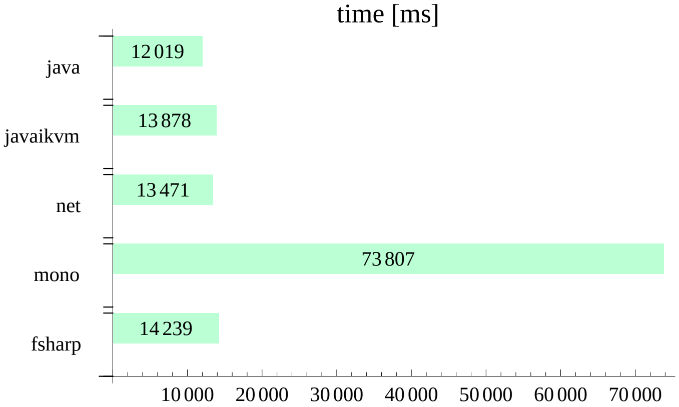

Fig. 4. Classifier execution performance results

C. Results by classifier

Here we will show the results for each classifier for two attributes – classifier building time(orange) and classifier executon time(green).

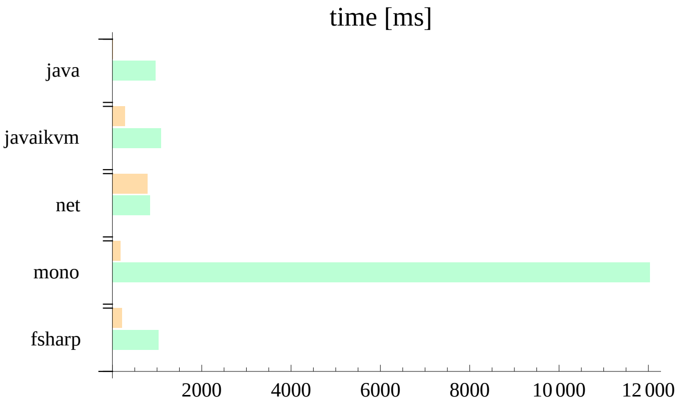

Fig. 5. NaiveBayes performance results

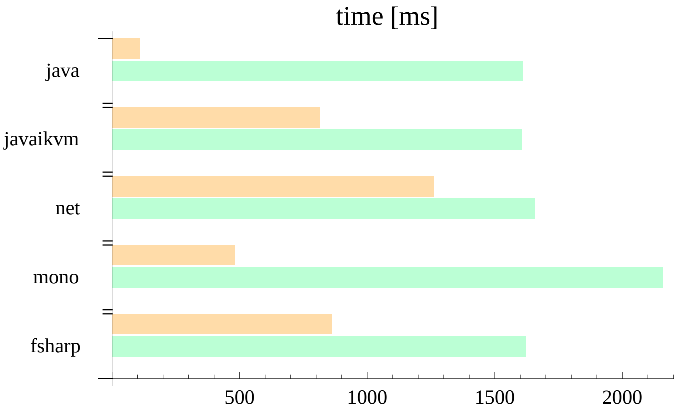

Fig. 6. RandomForest performance results

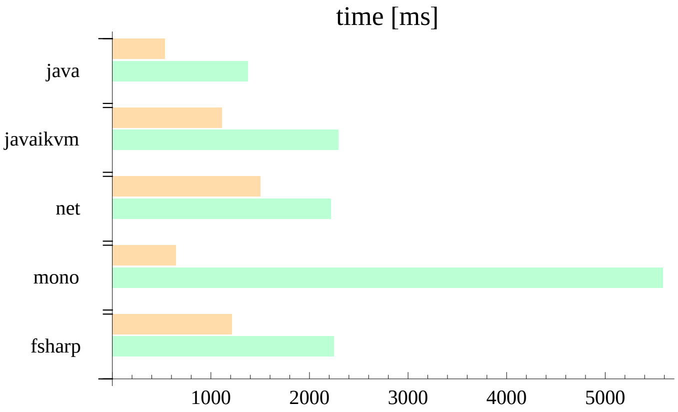

Fig. 7. SMO performance results

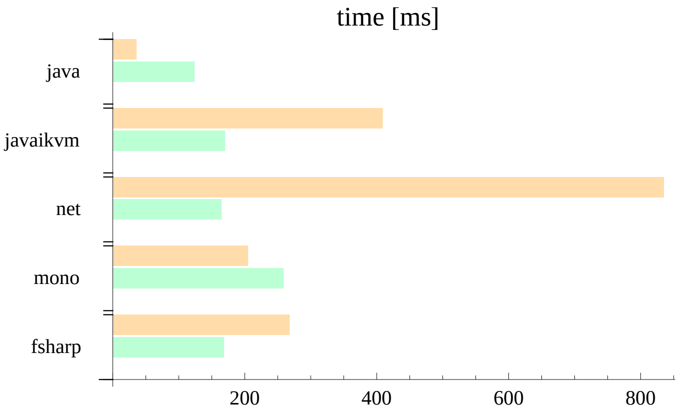

Fig. 8. J48 performance results

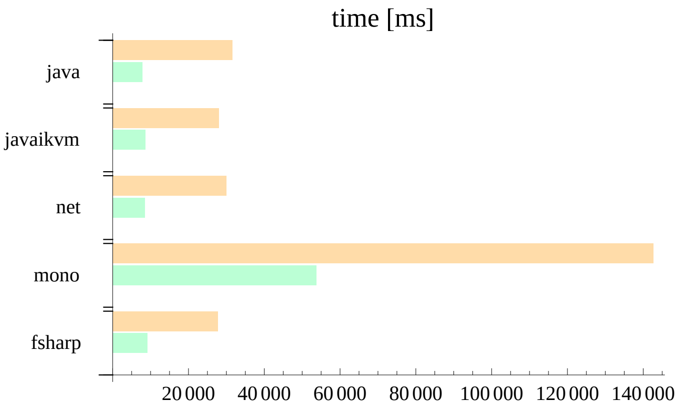

Fig. 9. MultilayerPerceptron performance results

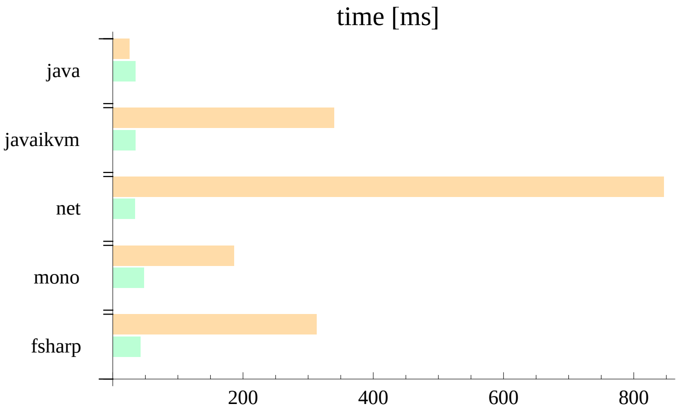

Fig. 10. ConjunctiveRule performance results

V. Conclusion

The results of the total classifier building performance were a bit of a surprise, as `javaikvm` and `fsharp` are actually faster than `java`. Later results on particular classifiers showed that the benchmark was skewed by the MultilayerPerceptron benchmark, which accounted for more than half of the total benchmarking time. Another abnormality is Mono – its performance was so bad, that it made the graphs hard to read. Here is the total graph again, with the MultilayerPerceptron and Mono results left out.

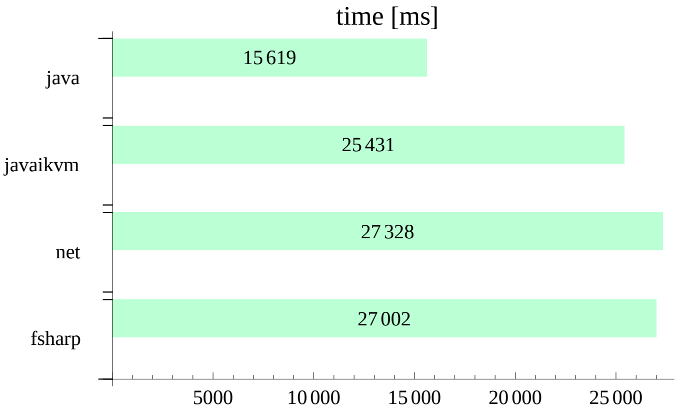

Fig. 11. Total performance results

The results show that Java is the fastest, by about 60%. But we still conclude that using Weka on .NET is possible and for some use-cases practical.

VI. Further work

More algorithms and different datasets could be tested – this paper doesn't include any regressional algoritms, and some algorithms used were not a good match for the dataset used. Weka also includes algorithms for preprocessing and filtering which were not featured here. The resulting data should also be weighted before being plotted, to avoid the skew in data that manifested itself here. We should also investigate how it was possible for .NET to be faster than Java in the MultilayerPerceptron benchmark, and why Mono performance was so bad in some cases.

Another thing which could be added are possible use-cases for Weka on .NET, such as using tools only available in the .NET ecosystem. Microsoft SQL server with its Analysis Services is a good example of a such tool.

Appendix A
Tools

In this appendix section we will briefly describe some technologies and tools we've used.

A. Weka

Weka (Waikato Environment for Knowledge Analysis) [1] is a widely used machine learning library, both in academia and business.

Fig. 12. Weka logo

Its development started in 1993 at the University of Waikato, New Zealand. Initially it was written in TCL/TK and C, but later in 1997 it was completely rewritten in Java. Weka is free software released under the GNU General Public License.

B. Mono

Mono is a software platform designed to allow developers to easily create cross platform applications. Sponsored by Xamarin, Mono is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime.

Fig. 13. Mono logo

C. IKVM

IKVM.NET is an implementation of Java for Mono and the Microsoft .NET Framework. It includes the following components:

- A Java Virtual Machine implemented in .NET
- A .NET implementation of the Java class libraries
- Tools that enable Java and .NET interoperability

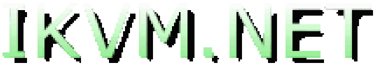

Fig. 14. IKVM logo

References

[1] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten (2009); The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.