Building A Statistical Baseball Model with UNIX and AWK

By Steve Sabaugh

Outside of the UNIX kernel are several tiny programs and utilities. There is also a programming language native to UNIX called AWK. AWK comes standard with UNIX and Unix-like operating systems like Linux. AWK is a data-driven language especially useful for data extraction, report making, and “one-liners” or very short programs. Also, as we will see later on it can be a very powerful data analytics tool. Its syntax is very similar to C and C-like languages like Java, so it is very easy to learn if you know one of them, or it is worth learning if you want to learn those languages. It also has very practical applications for office jobs. You can automate a lot of very boring and repetitive tasks, and extract pertinent data from very large files quickly. File handling in other programming languages is trickier because you have to handle file error and memory exceptions but AWK has none of those. AWK interfaces directly with the OS and not an interpreter so it is much faster than Python which makes a huge difference with big files. AWK gets its name from the last names of its 3 creators at Bell laboratories, the same place that developed the UNIX operating system and the C language, not to mention the countless other critical developments in computer science and our everyday lives; Alfred Aho, Peter Weinberger, and Brian Kernighan. Aho just received the 2020 Turing Award for his contributions to the field of Computer science and Brian Kernighan literally wrote the book on C.

In our ‘awk’ folder we have our .txt file with Boston Red Sox’s Hall of Fame Left Fielder, Ted Williams’s lifetime stats. We can look at the file with a short AWK program from the command line prompt. First, we invoke awk in the command line, then type our AWK program code enclosed in single quotes [‘’] followed by the input file that our AWK program will process, in this case, ted.txt.

```
awk '{print $0}' ted.txt
```

demo | output

AWK is made up of pattern-action statements.

```
pattern { action }
```

Think of it like an ‘if statement’ without writing ‘if’ and parentheses () around the condition or in this case the pattern. The program will read the input file and “if” the current line in the file it is reading (NOTE: it is important to understand that AWK reads files line-by-line) matches the pattern, the action in curly brackets {} will be performed. Now, that is a lot of data to sift through so let us take a look just at the header. We will use a built-in variable in AWK named NR that gives us the number of lines read.

```
awk 'NR == 1 {print}' ted.txt
```

demo

Now you saw before I put [dollar sign]0 and this time I just used print. Anything after a \$ is a field number and $0 is all the fields in a line, however, if you omit \$0, AWK will assume you want to print the whole line. To print the second line we just compare NR equality to 2. We can also omit the action statement altogether and AWK will assume we want to print the whole line.

```
awk 'NR == 2' ted.txt
```

demo | output

The file ted.txt has its fields separated by commas. By default, AWK separates fields by white space. To change the field separator to a comma we change the built-in variable FS. We do this by using a special pattern called BEGIN. In other words, “if you’re at the beginning of the file, change the field separator to a comma.”

```
BEGIN {FS = ","}
```

Since there are many fields in our Ted Williams stats, we should count how many fields each line has. I will use another built-in AWK variable NF and print that variable's value to the screen after line 1 has been read.

```
awk 'BEGIN {FS = ","} NR == 1 {print NF}' ted.txt
```

demo | output

Now that we know we have 30 fields per line, it would be helpful to make a table to number the header so we know what the field numbers are without having to squint our eyes and count on the screen. We can do this with a simple 'for' loop in our action block.

```
awk 'BEGIN {FS = ","} NR == 1 {for (i = 1;i < NF + 1;i++){ print $i " = " i "\n"}}' ted.txt
```

demo

Notice that the 'for' loop in AWK is the same as it is in JAVA and other C languages, and there is no need for a concatenation operator like in other languages in the print statement. We can also save this table to a text file so we have it later for reference.

```
awk 'BEGIN {FS = ","} NR == 1 {for (i = 1;i < NF + 1;i++){ print $i " = " i "\n" > "tedTable.txt"}}' t

cat tedTable.txt
```

demo

I want to give an example of UNIX pipes '|' and a small UNIX program called sort. Since our file is already sorted in order chronologically. Let's look at just years and home runs fields 1 & 12.

```
awk 'BEGIN {FS = ","} {print $1 "\t" $12}' ted.txt
```

We can "pipe|" our AWK program through sort in UNIX, sorting our file by home runs and in reverse starting with the most hit and the year to the least hit and the year by using sort with a reverse option.

```
awk 'BEGIN {FS = ","}$1 >= 1939 && $1 < 1961 || $1 == "Year" {print $12 "\t" $1}' ted.txt |sort -r
```

demo

Now let's do some data analysis on Ted Williams. Ted Williams, it has been argued, is the greatest left-handed hitter of all time. He was also the last hitter to hit .400 in a season. So let's search for the season(s) he hit .400 and above. We'll use our handy table we just printed to the screen to see that field 18 is BA or batting average. I will also print the year, field 1 followed by a blank space then the BA.

```
awk 'BEGIN {FS = ","} NR > 1 && $18 > .3999{print $1 " " $18}' ted.txt
```

demo | output

He did it in 3 seasons. Did Ted Williams play full seasons those years? Let's grab some more data. For .400 to be impressive you have to qualify for the batting title, meaning you have to play close to a full season. In his day a full season was 155 games, so a season with more than 100 games played would put Ted in contention for the batting title. We'll also look at how old he was. To format these nicely, we'll use a 'printf()' function, which is exactly the same as C or Java.

```
awk 'BEGIN {FS = ","} NR > 1 && $18 > .3999 {printf("%d %d %3d %.3f\n",$1, $2, $5, $18)}' ted.txt
```

demo | output

We see that in 1952 and 53 Ted only played in a few games. That is because Ted was fighting in the Korean War, which you probably learned about from social studies class. In 1941 he legitimately hit .406 on the season and won the batting title making him the last player to hit .400 or more in a season. Ted was in the Marine Corps as a Fighter Pilot. Despite being an all-star ballplayer, he was not in Korea as a goodwill ambassador. He saw real combat. In fact, his wingman was John Glenn, who would go on to be the first American to orbit the earth on Mercury 6 in 1962. He was too valuable of a pilot not to fight in Korea because

he had many years of experience in America's previous conflict, WW2.

We're going to use that built-in variable again NR to give us a handy reference of numbered lines in our data set. Let's take another look at Ted's stats.

```
awk '{print NR " " $0}' ted.txt
```

demo

Notice lines 6-8. Those lines read

```
Did not play in major or minor leagues (Military Service)
```

As you may have also learned in social studies class, WW2 was raging during 1942, 43, and 44. Baseball continued to be played as players came and went from military service or were disqualified from fighting. One quality that Ted had that made him such a great hitter was superior eyesight. He could pick up the spin of the ball better than those with 20/20 vision. Thus, having more time to react if a ball would be in the strike zone or not. This superior vision made Ted an Ideal candidate as a pilot and he was in peak form for a fighter pilot at the age of 24. Unfortunately for baseball fans, he was also just reaching his peak form for playing baseball. Baseball writers and fans alike, from those who watched the "splendid splinter" play in person, as my dad did, or those who know him only from film, stats, and old box scores like myself, always debate and dream about what "could've been" if old "teddy ball game" didn't miss those 3 years at his peak. Ted Williams is enshrined in Baseball's Hall of Fame, so it is not like he's cheated of any glory, just a world series title, but with AWK we can build a quick predictive model of "might've been."

The simplest way to do this is to do a regression analysis. Linear regression is one of the first predictive modeling techniques used in machine learning and modeling. The technique we'll use for brevity and speed is the least-squares method. So, in a nutshell, a regression analysis takes 2 variables x and y (NOTE: I mean variables in the statistical sense of the word as in data points, not in the programming sense as in a named memory location whose value changes as the program runs). The x variable is our independent variable, and our y variable is our dependent variable or the one we want to predict or analyze. In other words, based on past x,y relationships, what can we predict y will be if x = n. In our model, our independent x variable will be Ted's age and our dependent y variable will be his batting average. We will base our model on his previous 4 full seasons, where x and y (age and avg) are both known and we will write a program that will give us a formula for the best fit for a "curve" that is a straight line (thus the linear in "linear regression") that would match his positive trajectory of batting average through his prime years with only his age or x as the known variable. This model cannot, therefore, predict slumps.

So first, we need to manipulate the data from those first 4 seasons. Since we have line numbers handy, we can use a traditional compound boolean statement with the built-in NR variable.

```
awk 'NR >= 2 && NR < 6 {print NR " " $0}' ted.txt
```

demo

The linear regression formula is

```
Y = a + bX
```

Here is how we will solve that formula, the use that formula for our predictive model. So far, all of our awk programs have been done "in-line" or on the command line because they were very small. Our statistical model will still be a small program, but sometimes you might find it easier to write the program in a separate text file and you can write it out in a more traditional way you are used to writing programs in other languages that use curly braces {} like Java or the C languages. You could use any text editor and create a new file TedWarYears.txt.

```
BEGIN {
    FS = ",";   #field seperator
    count = 0   #total number of previous seasons (to avoid using magic numbers)
}
NR >= 2 && NR < 6 { #populating arrays of 1939-1941 seasons
    x[NR] = $2; #age array
    y[NR] = $18;    #batting average array
    count++

}
END {
    for(i = 1;i < count + 1;i++){   #we loop through our arrays to get sums and sums of
    sumX = sumX + x[i];     #products to produce linear regression formula
    sumX2 = sumX2 + x[i]*x[i];
    sumY = sumY + y[i];
    sumXY = sumXY + x[i]*y[i]
    }               #solving for a and b in our formula
    b = (count * sumXY - sumX * sumY) / ( count * sumX2 - sumX * sumX);
    a = (sumY - b*sumX) / count;

    for (i = 24; i < 27; i++){  #here we use our formula to predict ages 24-27 batting avg
    pAvg = a + (b * i);     #this is our linear regression formula with i as our "x" variable
    printf("Age %d: %.3f projected batting avg.\n",i, pAvg)
    }
}
```

Notice in the BEGIN action I created a count variable. This was used to avoid using magic numbers. AWK is designed for doing “quick and dirty” number crunching and data analysis or creating small useful programs on the fly. Popular conventions and norms are not usually necessary because AWK is not meant for building enterprise-level solutions. However, due to the didactic nature of this exercise, I thought it was important to create a variable to “count” how many records I was counting. If I wrote this program for myself, I would have put 5 instead of count + 1 in my first ‘for’ loop header, and when solving for “a” I would’ve divided by 4, etc., but old habits are hard to break.

Notice that in the second ‘for’ loop I use “i” as my loop variable when it would’ve been more accurate to use x because 24 - 27 are the ages or x variables. Here, I thought it better to use “i” as the traditional abbreviation for index because x[] was already in use as my array name. So “i” takes x’s place in the traditional curve fitting formula y = a + (bx).

An important thing to consider with arrays in AWK. Note how I populate the x[] and y[] arrays in the BEGIN action statement. The array’s first subscript is the number 2, not 0 like most languages or 1 in BASIC or Visual Basic. That is because AWK uses associative arrays like dictionaries in Python or hash tables in JS or Java. A subscript could be essentially any data type (integer, float, string). This is a very powerful feature for creating quick databases and even creating your own programming language with AWK plus a whole lot of other useful programs but also is convenient for populating arrays with the NR variable and being able to skip the header.

We also did all of our number-crunching in a special AWK pattern called END. This is where you program after all of your input file has been read into your program.

If you are used to Java or other C languages, you may be thinking I erroneously omitted some semicolons [;] to terminate some statements. In AWK, semicolons [;] are used to divide multiple statements in an action block, NOT to terminate every line, like in Java or C languages. Think of AWK’s usage of the semicolon [;] in action block similar to how it is used in ‘for’ loop headers in Java and C [(initialization; loop conditional; update)] Notice in the ‘for' loop we do not terminate the update section in the semicolon [;].

When finished, we run the program from a file from the command line we just have to use this command.

```
awk -f "filename.ext" inputfile.ext
```

Now to see what our model predicts Ted Williams would have hit in his ages 24, 25 & 26 seasons if he had not gone and fought in WW2 and did not slump.

```
awk -f "TedWarYears.txt" ted.txt
```

demo | output

There you have it. The debate and speculation are settled. If you are familiar at all with baseball statistics you may be able to tell that our model is a bit generous to the old “splendid splinter.” Die-hard fans of his, like my father, would say “yeah, that’s about right.” Teddy “Ballgame’s” legend is alive in me mostly because of hearing stories from my dad. Korean war era veterans like himself thought he was the greatest. Fresh from the air over the heat of the battle on the Korean peninsula, Ted came back from the war and finished the season by batting over .400 as if he never left. Now adding some weights and different regression methods would bring those numbers down to earth a little bit, especially those last two years, which seem impossible, but then again maybe our model is perfect because it just confirms the legend. This was a fun exercise, but this is exactly what you would do if your boss would ask you to predict sales agents on the floor to potential sales or cops on street to crimes committed.