

AI/GUPPI

The Artificial Intelligence Generic Usable Pain-Free Perceptron Instantiator

By Ken Cooney

What Is a Perceptron?

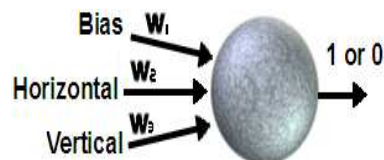
A perceptron is one of the simplest ways for a computer to learn something. Let's say someone has an item and you want to know if it's a tricycle. You can determine if it's a tricycle by asking questions. Does it have three wheels? Does it have a motor? Does it have pedals? Does it have a seat? Does the seat hold more than one person (to rule out a rickshaw)? These questions are answered with yes or no (true or false). A computer can do this, too. You can tell it the answers to the questions and then say "this is a tricycle". If the computer incorrectly says it's not a tricycle, it can make corrections. The same holds if you provide the answers to the questions and then says "this is not a tricycle" and the computer incorrectly says it is. It makes corrections. After a while, if there is a correlation in the data, it will learn how to identify a tricycle.

Why would we want a program that recognizes a tricycle?

Well, it may not be interesting to make a program that recognizes a tricycle based on true or false answers for a group questions. I did that example to help explain what a perceptron can do and give you an idea how it works in the big picture view. Perceptrons can be useful. You can make a perceptron recognize a written number. With a set of ten perceptrons, you can have each perceptron recognize a different number (zero through nine). After the perceptrons learn how to recognize a specific number, you can have it determine what number it is. In the larger view, you could have one that perceives a lot of written characters. Technically an S and a 5 would look similar and would require something else to determine which it really is, but that's an advanced version of learning that would require something like a Bayesian Network. But, needless to say, perceptrons could be useful.

How Perceptrons Work

Don't worry, we won't get bogged down in technobabble. I'll keep this simple using an example, which works perfectly since in a nutshell a perceptron is very basic. We'll look at a simple perceptron with two inputs and a bias. Don't worry about what a bias is just yet. Let's imagine that you have an orchard and in one part of the orchard you have apple trees and in the other part of the orchard, you have mango trees. The orchard is 50 feet long by 50 feet wide (or 50 feet along the horizontal and 50 feet along the vertical). You want to see if there is a way that you can build a wall in the orchard so the apple trees are on one side of the wall and the mango trees are on the other side of the wall. So, the inputs are horizontal and vertical (and a bias).



The data you pass into the perceptron is the horizontal and vertical locations of each tree. One tree is 5 feet away from the south-west corner along the horizontal and 10 feet away from the south-west corner along the vertical. It's an apple tree. So it's in the set of apple trees. The datum for this is (5, 10, 1). You have a mango that is 40 feet from the same corner along the horizontal and 30 feet from the same corner along the vertical. It's a mango, so it's not in the set of apple trees. The datum is (40, 30, 0). You do this for all the trees.

Now, you input the data into the perceptron; you always input a 1 for the bias. The program tries to imagine a wall in the orchard where apple trees are on one side and mango trees are on the other. If it finds a tree on the wrong side of the wall, it needs to shift the wall a bit. In reality, the perceptron doesn't know what the data is, but it's trying to learn if there is a straight line it can draw to separate the two sets. In essence, it is solving $aX+bY$ where the X and Y is horizontal and vertical data and a and b are the weights. If $aX+bY$ is greater than or equal to one, it's in the set. Otherwise, it's not in the set.

We haven't mentioned the bias, yet. The bias can shift the wall over. Basically, this makes the equation $aX + bY + c$ where c is a constant (bias * weight). Now, if the perceptron expects an apple tree to be on one side of its imaginary fence and the tree is on the wrong side, the perceptron adjusts the weights so it's more likely that the apple tree will be on the correct side of the fence. If there is a way to build a wall between the two types of trees, it will find it. It will have to loop through the entire set of data several times to do so, but it'll find a solution.

Now, to take this one step further, let's say you tell the program that there's a new tree that's starting to grow and it's located at 18 horizontal, 20 vertical. You ask the perceptron "Is it an apple tree or a mango tree?" The program will run the data (18, 20) and will return a 1 if the perceptron believes (or perceives) that it's an apple tree or a 0 if it believes that it's a mango tree.

Problems With Perceptrons

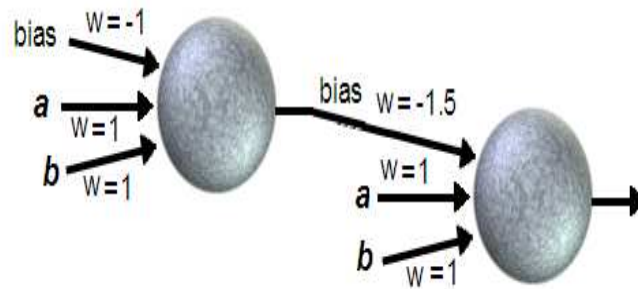
In 1969, a famous book entitled *Perceptrons* by Marvin Minsky and Seymour Papert showed that it was impossible for a perceptron to learn an XOR function. They incorrectly surmised that a perceptron with three or more layers couldn't learn the XOR.[1] In my simple example (see *How Perceptrons Work*), I mentioned a tree classifier. The program came up with the weights so a wall can separate the two groups of trees. Well, an XOR is a case where either A or B is true, but not both. So, $a \text{ XOR } b$ looks like this:

X	T	1	0
	F	0	1
		T	F
		Y	

As you can tell, there is no straight line (or "wall") that you can draw so the ones are on one side and the zeroes are on the other. Well, that was a let down. We spent all this time saying how great perceptrons are and now we find out it can't do an XOR. Well, Steven Grossburg

published a paper in 1973 that stated you could create an XOR using two perceptrons.[2] Of course regardless of Grossburg's findings, the Minsky/Papert paper was often cited by other publications and for ten years there was a drastic decline in AI research until Neural Network research started a resurgence of interest in AI.

In any case, here's how to create an XOR with two perceptrons:



The first perceptron is an AND. It will only return a 1 if a and b are 1 (true). The second perceptron, you pass the same values of a and b , but you use the output of the AND as the bias and multiply it by -1.5 . So, if a and b are true, the calculation for the second perceptron is: $(1 * -1.5) + (1 * 1) + (1 * 1) = 0.5$ and since 0.5 is less than 1 , the output is a zero. If a is true and b is false (or vice versa), you get an output of 1 . Obviously if a and b are false, you get an output of zero.

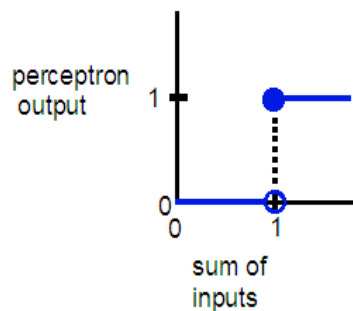
There you go. You probably started reading this not knowing much about perceptrons and now you can create something that two MIT PHDs said was impossible. ☺

So, all is well with the world, right? Well, not exactly. There are still some issues with perceptrons. A perceptron doesn't know if a problem has no solution. It will try to come up with a correlation, continue to adjust the weights, and run forever. Also, a perceptron (or a network of perceptrons) is only as good as the training information you provide it. If you don't provide a sufficient set of test data, the perceptron may incorrectly classify items. Also, once a perceptron has finished learning, it can't learn any more. So, if you come up with new inputs or new datum that comes up with a new conclusion, you'll need to train a new perceptron (or perceptron network) from scratch.

In conclusion, perceptrons are still pretty good at classifying something once it learns how to classify it, presuming it learns using a good set of training data. If you know what it can and cannot do, you can still use them for learning some things. Also, perceptrons are very easy to code with is a plus.

Introduction to AI/GUPPI

As I mentioned before, AI/GUPPI stands for The Artificial Intelligence Generic Usable Pain-Free Perceptron Instantiator. Basically, it's a program that allows a person to create a perceptron with little or no coding. The inputs into the perceptron can be boolean (one for true and zero for false), integers, or numbers with decimals (otherwise known as floats). You can add as many inputs as you wish. You can determine the learning rate (how much the perceptron adjusts the weights). You can even come up with your own function. If you don't want to make a function, the standard step sum function (stepSum) has been provided. A "step sum" is a fancy way of saying "if the sum of the inputs times weights is equal to or greater to one, output a one; otherwise output a zero."



The circles in the picture are a bit exaggerated, however this image above shows that when the sum of inputs is zero, the output (blue) is zero. When it reaches one, the output is a one. It's called a step, because well ... it looks like a stair step.

Your First Perceptron

We'll start with an easy one to get your feet wet. No coding is required. This perceptron will have three boolean inputs and a bias. It will use boolean values for the test data. Essentially, it's an AND perceptron. It will output a one if all three inputs are one. Otherwise, it'll output a zero. (see *The AND Perceptron*).

Before we get started, I'll go into how to create a perceptron with this program. For every perceptron, you'll need.

A perceptron file – this is the number of inputs and initial weights.

A training file – this has data for training the perceptron.

A testing file – this validates that the training works.

An identification file – this is optional. This has inputs for items you want to categorize. Remember in my example where we found a new tree and wanted to know if it was an apple tree or a mango tree? Well, that data goes in here.

The Perceptron File

The perceptron file is used for creating a perceptron. It must be named Perceptron.txt. It consists of the following properties:

PERCEPT

The file must have two or more percepts; one of which is a bias. These are the inputs to the perceptron. The program will always assume the last input defined in the file is the bias. The descriptions are for your benefit and understanding. The program doesn't reference them. You can also specify a method that adjusts the input (ex: does a square root of the input value). By default, it'll use the unadjusted weight value. There are two methods available in the Function class: `getSquareInput` and `getSquareRootInput`. Of course, you can make your own method.

Format 1: PERCEPT| initial weight | description

Format 2: PERCEPT| initial weight | description | class | method

Example 1: PERCEPT|0| x value

Example 2: PERCEPT|1| x value | perceptron.Function | getSquareRootInput

Note: In some countries, the comma is used for a decimal point instead of the period. This program will accept values of "1,05" and convert it to "1.05". In order to do that, use the `DECIMALPOINT` parameter (see below).

DECIMALPOINT

This is optional. If you live in a country that uses commas for decimal points, you'll want to define this value. By default, decimal points are periods. By changing this to the comma, the program will accept all inputs and outputs that have numbers like "5,00" (instead of "5.00"). Valid values of `DECIMALPOINT` are the period and the comma. All other values will be ignored.

Example: `DECIMALPOINT|,`

Note: If you are using a comma for decimal points, this needs to be the first property defined in the perceptron file or following the debug property (see `DEBUG`).

LEARNRATE

This is how much the perceptron learns; how much it adjusts the weights (see below for the learning rate). If this parameter is not in the file, the learning rate defaults to 1.

Format: LEARNRATE|number

Example 1: LEARNRATE|0.05

Example 2: LEARNRATE|1.0

By default, the weights change based on the learning rate as follows:

$$\text{New weight} = \text{old weight} + ((\text{expected result} - \text{determined result}) * \text{learning rate} * \text{input})$$

Example 1: The old weight is 5. We expect a 1 and the perceptron determined it was a 0. The learning rate is 0.5. The input was 2. Looking at the formula:
New weight = $5 + ((1 - 0) * 0.5 * 2) = 5 + 1 = 6$.

Example 2: The same values, but we expect a 0 and the perceptron determined it was a 1.
New weight = $5 + ((0 - 1) * 0.5 * 2) = 5 + (-1) = 4$.

In either example, the perceptron has a better chance of coming up with the right answer, and if not, the weight is adjusted again (lower or higher).

The learn rate will change if you use a method for adjusting the input. The new weight will adjust. For example, if the percept takes the square root of the input, the weight is adjusted as follows:

$$\text{New weight} = \text{old weight} + ((\text{expected result} - \text{determined result}) * \text{learning rate} * \text{squareRoot}(\text{input}))$$

FUNCTION

For advanced developers, you can create a function that takes the inputs (percepts) and returns a 1 or a 0. This parameter is for defining the function to use. If you don't specify a function, it totals all of the (input * weight) values and checks to see if the total is greater than or equal to one (this is called a step sum).

Format: FUNCTION| package.class | method

Example: FUNCTION|applicationp.Functions|stepSum

See *Advanced Topic: Making Your Own Function* for details on making functions.

DEBUG

This is optional. When creating a new perceptron or using a new function, you may want to turn debugging on so you can see what's happening. Default is off. You'll want this as your first property in the file or have it immediately after the DECIMALPOINT property (see DECIMALPOINT). Valid values are on and off.

Example: DEBUG|on

The Training and Test Files

Both of these files have the same format. You should have one input for all PERCEPTs (excluding the bias which always has an input of 1). The last value should be the expected value. The expected value must be a 0 or a 1.

The training file should be called Training.txt and the testing file should be called Testing.txt. Both files are case sensitive, so they start with capital Ts.

Example test data for a perceptron with two percepts (not including the bias).

```
5|5|1
10|10|0
```

In the above example, inputs of 5, 5 should return a 1. Inputs of 10, 10 should return a 0.

It's recommended to have a lot of test and training data. Fifty is usually a good number unless there are fewer test cases (such as training an OR perceptron). If any of the tests fail, the program will indicate which tests fail.

The Identifying File

This file is optional. If you have one, it tries to identify what these things are. The format is the same as the training and test files, except you don't have the expected value (since you don't know what it is). The identifying file should be called Identifying.txt. It's case sensitive; so it's a capital I.

Example test data for a perceptron with two percepts (not including the bias).

```
3|6  
9|1
```

The identifying file will be processed after the training and testing files are processed.

Running AI/GUPPI

Go to the directory where you extracted AI/GUPPI. You'll see two runGuppi files.

Windows: execute runGuppi.bat

UNIX: execute runGuppi.sh

You'll be prompted for the directory where you put the data files. You can either use relative path (“..data”) or absolute path (“C:\temp”) directories.

Future Changes to the Program

I have some thought of future changes to make the program easier to use. The changes are listed in the order which I plan on tackling them. No promises on when I'll get to it as I'm still working on my master.

1. Add a project file. This file will contain the locations of the various files. In other words, each project file will store the filenames for the perceptron, training, testing, and identifying data. This also means the filenames will be more generic (ex: instead of forcing a naming convention of “Perceptron.txt” you can use *filename.gpf* [generic perceptron file] where filename can be any legitimate filename).
2. Add a GUI front end. The GUI will allow you to load, edit, and save all files (project, training, testing, identification), run training and tests, turn on debugging, etc. I'll make it so the program can still run from a shell script for UNIX developers and those who prefer the DOS command prompt.
3. Add a way to save the trained perceptron. Technically you can do this by updating the perceptron with the final outputs. Thus, why it is a lower priority.
4. Use XML for the datafiles. I'll make sure it also can use pipe delimited files.

Final Word

Remember, this is only one way of getting a computer to learn something. There are other ways, including (but not limited to) Bayesian Probabilities, Hidden Markov Models, and Neural Networks. These three are advanced topics but if you're ambitious, check them out.

Even if you don't go further into the AI world than playing with the Generic Perceptron, I hope you had fun in trying it out! If you think it's neat, feel free to send me feedback at my website.

Ken Cooney

<http://www.kencooney.com/>

References:

[1] M. Minsky and S. Papert. Perceptrons. Cambridge, MA. MIT Press. 1969

[2] S. Grossberg. Studies of Applied Mathematics. September 1973.
<http://cns.bu.edu/Profiles/Grossberg/Gro1973StudiesAppliedMath.pdf>

[3] NPRM on Tire Pressure Monitoring System FMVSS No. 138. http://www.nhtsa.dot.gov/cars/rules/rulings/TPMS_FMVSS_No138/part5.6.html

See also:

Perceptrons: Basic Neural Networking.

<http://www.aihorizon.com/essays/generalai/perceptrons.htm>

R. Rojas. Neural Networks. Springer-Verlag, Berlin. 1996.

http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/1996/NeuralNetworks/K3.pdf

http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/1996/NeuralNetworks/K4.pdf

Appendix 1 - Perceptrons

Perceptron 1: The AND Perceptron

This will be a perceptron that looks at x and y and z. Each of these three variables can have a value of true or false. If all three are true, the perceptron returns true. Otherwise, it returns false. It's very simplistic, but it's a good start for getting your feet wet.

Perceptron.txt file

```
PERCEPT|0|x value  
PERCEPT|0|y value  
PERCEPT|0|z value  
PERCEPT|0|bias  
LEARNRATE|0.1
```

Training.txt file

```
0|1|0|0  
1|1|0|0  
0|0|1|0  
1|1|1|1  
0|0|0|0  
0|1|1|0  
1|0|0|0  
1|0|1|0  
0|0|1|0
```

Testing.txt file

Same file

Some things to try:

1. Once you got this working, try removing one of the training sets that expects an output of zero. Make sure the testing set has all nine cases. Do all nine test cases pass?
2. Try just including the "0|0|0|0" and "1|1|1|1" cases in the training set, do all nine the tests pass? Why or why not?
3. Make an OR perceptron with three inputs. Try putting all possible combinations into both the training and testing files. Once you get it to work, figure out what's the smallest training set you can use.
4. Make a perceptron that perceives (x and y) or z.

Perceptron 2: The Hydroplaning Perceptron

I found one formula for determining if a car will hydroplane. [3] I'm sure there are better formulas somewhere, since this one presumes the factors are:

- a) any water on the road
- b) the pressure your tire is inflated to
- c) the speed your car is going

It doesn't take into the account the amount of water on the road. Basically the formula is:

$$\text{hydroplaning speed} = 10.35 * \text{square_root}(\text{tire pressure})$$

Which is equivalent to:

$$10.35 * \text{square_root}(\text{tire pressure}) - \text{hydroplaning speed} = 0$$

For our perceptron, we can change it to:

$$\text{if } ((10.35 * \text{square_root}(\text{tire pressure})) + \text{speed} + 1 \geq 1) \text{ output a 1. Otherwise output 0.}$$

We'll need a new function for summation:

$$\begin{aligned} \text{summation} &= w1 * \text{square_root}(t1) + w2 * \text{speed} + 1 * \text{bias}; \\ \text{if this is greater than or equal to 1, output 1. Otherwise output 0.} \end{aligned}$$

Hydroplaning Perceptron - First Attempt

Before we get started, I'll let you know that we'll be using a typical approach that a person may take if that person was new to perceptrons. I'm going to make a new function for the perceptron so the calculating takes into account of the first percept's square root. I put this in the Functions class. You can make it part of the package (package perceptron), but it's not necessary so long as the class is in the classpath.

Note: You may need to create a dummy “main” class and execute it so the function compiles. Some compilers won't compile code unless it's explicitly called.

```
public int tirePressureSum(ArrayList<Percept> perceptList,
                          ArrayList<Float> oneTest)
{
    // add bias input of one if none found
    if (oneTest.size() == perceptList.size()-1)
        oneTest.add((float)1.0);

    // learning size || training size
    if ( (oneTest.size() == perceptList.size()+1) ||
        (oneTest.size() == perceptList.size()) )
    {
```

```

float total=0;
for (int i=0; i<perceptList.size();i++)
{
    Percept percept = perceptList.get(i);
    percept.setInput(oneTest.get(i));
    if (i>0)
        total=total+percept.product();
    else
        total= ((float)Math.sqrt(percept.getInput())) *
                percept.getWeight();
}
if (total>=1.00)
    return 1;
}
else
    System.out.println("Incorrect number of values.");

return 0;
}

```

Perceptron.txt file

```

PERCEPT|-240.0|tire inflation (PSI)
PERCEPT|230.14|speed
PERCEPT|-5900|bias
LEARNRATE|0.1

```

Training.txt file

```

30|0|0
25|0|0
20|0|0
30|56.7|1
25|51.8|1
20|46.3|1
30|56.2|0
25|51.3|0
20|45.8|0

```

Testing.txt file

```

30|56.5|0
25|51.6|0
20|46.1|0
30|56.8|1
25|51.9|1
20|46.4|1

```

It took an insanely long time for this Perceptron to learn this relationship, and that was with tinkering. With the training data shown above, the perceptron learns the relationship in 51 loops through the data set. If you start at all zeroes, the program can't determine the relationship. If I use boundaries that are plus or minus of one half the "hydroplaning speeds", I can come up

with a result sooner than using small boundaries (which makes sense). So, if (30,56.7), (25,51.8), and (20, 46.3) are in the hydroplaning set [3], adding 0.5 to each of the speeds are also in the set and subtracting 0.5 from each of the speeds are outside the set. Giving it a head start, I get the weights: (-241.0, 231.60016, -5900.0). In another run with different initial weights, I got completely different end result weights: (-276.0, 263.15005, -6629.0).

Since I added the plus or minus .5 boundary, tests inside that boundary range may (and did) fail. For both tests, the set of weights (25.0, 51.6) shouldn't be in the set but the perceptron thought it was. So, it learned the relationship to a degree. Basically, the perceptron found a line between my boundaries that satisfied all of the training data.

Why I Needed to Tinker (AKA Playing with the values):

The fact that we use square root of the tire pressure caused the perceptron some headaches. This is not because the perceptron can't handle square roots; this is because I incorrectly set up the perceptron. While the function I created takes into account the square root, the weight correction does not. This is evident when you see that the PSI weight and the speed weight is very close. We know that the PSI weight should be a little over ten times the speed weight. So, with tinkering, it learned A relationship, it didn't learn THE relationship. So, this tells me the learning function needs to adjust as well, so it takes into account the square root.

Lesson learned:

I need to adjust the weights to take into account the PSI calculation used a square root. As I said before, this is a typical mistake that people may make if they're new to perceptrons.

Hydroplaning Perceptron - Second Attempt

Instead of creating a new function for the perceptron, I'll tell the percept to take the square root of the input. As I mentioned before, the error will correctly adjust the weight.

Perceptron.txt file

```
PERCEPT|0|tire inflation (PSI)|perceptron.Function|getSquareRootInput
PERCEPT|0|speed
PERCEPT|0|bias
LEARNRATE|0.5
```

Testing.txt file

```
30|0|0
25|0|0
20|0|0
25|51.8|1
30|56.7|1
20|46.3|1
25|51.7|0
30|56.6|0
```

20|46.2|0

Testing.txt file

25|51.9|1

30|56.8|1

20|46.4|1

25|51.6|0

30|56.5|0

20|46.1|0

I was able to reduce the difference between the “in the set” and “outside the set” values to 0.5. In other words (25,51.8) is in the set and (25,51.7) is out of the set. Not surprisingly, the tests pass since they aren't inside the range that I used for training the perceptron. However, since we've reduced the MPH to a tenth of a mile per hour, I'm not overly concerned. This should be sufficient for identifying if a car will hydroplane.

The end result is as follows (see Appendix 3 for the first 5 loops and the last 5 loops):

```
It took 51907 loops through the test data to learn the input.
Percept: W(-289.11926) - tire inflation (PSI) input
(getSquareRootInput)
Percept: W(28.207544) - speed input
Percept: W(-12.0) - bias input
Running tests...
30.0, 56.5, 1.0 expected 0 and got 0.
25.0, 51.6, 1.0 expected 0 and got 0.
20.0, 46.1, 1.0 expected 0 and got 0.
30.0, 56.8, 1.0 expected 1 and got 1.
25.0, 51.9, 1.0 expected 1 and got 1.
20.0, 46.4, 1.0 expected 1 and got 1.
All tests passed!
```

Note: If you take the PSI weight and divide it by the speed weight, you get 10.249 which is exceptionally close to 10.35. This is a 1% margin of error.

Things to try (if you got the time):

1. Reduce the learning rate to 0.25. It'll actually come to a solution in 51058 loops (try it!)
The weights will be different but the PSI weight divided by speed weight is still 10.249.
The weights are PSI (-143.01395), speed (13.954246), bias (-5.5)
Try smaller learning rates and see what you get.
2. Manually calculate the speed cars will hydroplane for the following tire pressures:
22 PSI
24 PSI
27 PSI
Try putting those values (round speeds to the nearest tenth of a mile).
Do the tests pass?

Appendix 2 - Advanced Topic: Creating New Methods

Making Your Own Function

As I mentioned earlier, you can make your own function. The one provided (stepSum), totals all of the weights and sees if they are greater than or equal to one. If so, the perceptron function returns a 1. Otherwise, it returns a zero.

The function should be defined as follows:

```
public int functionName(ArrayList<Percept> perceptList,
                       ArrayList<Float> oneTest)
{
    // The input may not have a value for the bias input.
    if (oneTest.size() == perceptList.size() -1)
        oneTest.add((float)1.0);    // add bias input of one

    // Verify we have one of two valid arraylist sizes.
    // used for identifying || used for training
    if ( (oneTest.size() == perceptList.size()+1) ||
        (oneTest.size() == perceptList.size()) )
    {
        float total=0; // initialize total
        for (int i=0; i<perceptList.size(); i++)
        {
            Percept percept = perceptList.get(i);
            percept.setInput(oneTest.get(i));
            // product = weight(i) * oneTest.get(i)
            float myProduct = percept.product();
            // Total the values of the products
            total= total + myProduct;
        }
        // return 1 if you get an expected result from function
        // ex: boolean expectedResult = (total > 1.00);
        if (expectedResult)
            return 1;
    }
    else
        System.out.println("Incorrect number of values.");
    return 0;
}
```

You'll need to import `perceptron.Percept` if you use a package name other than `perceptron`. Once compiled, you can use it, so long as the compiled class is listed in the `CLASSPATH`.

A word of caution: I do not recommend calling a method that is not part of this application unless you have written it or compiled it yourself (and if you have compiled it, you should figure out what it does and determine if it is harmless). This program basically executes the method and that means it'll do whatever the method says to do. So, if a mischievous person decided to make it remove a bunch of files on your system, it will do that. You have been warned.

Making Your Own getInput Method

This is considerably easier. There require a Percept as an input. Here's the methods that are currently available:

```
public float getPerceptInput(Percept percept)
{
    return percept.getInput();
}

// for returning square root of a perceptron input
public float getSquareRootInput(Percept percept)
{
    return (float)Math.sqrt(percept.getInput());
}

// for returning square root of a perceptron input
public float getSquaredInput(Percept percept)
{
    return (float)Math.pow(percept.getInput(), 2);
}
```

Once again, you'll need to import `perceptron.Percept` if you use a package name other than `perceptron`. You can use your new method, so long as the compiled class is listed in the `CLASSPATH`. The same “word of caution” applies to these as well.

Appendix 3 – Perceptron Attempt 2 Output (Debug Mode)

This is the start of the run up to the 5th iteration and the last 5 iterations of the run. The decimal point in this example is the period.

```
Loading perceptron...

Percept: W(0.0) - tire inflation (PSI) input (getSquareRootInput)
Percept: W(0.0) - speed input
Percept: W(0.0) - bias input
Learn rate is: 0.5.

Loading training data...

Added:30.0, 0.0, 1.0 Expected output is: 0.
Added:25.0, 0.0, 1.0 Expected output is: 0.
Added:20.0, 0.0, 1.0 Expected output is: 0.
Added:25.0, 51.8, 1.0 Expected output is: 1.
Added:30.0, 56.7, 1.0 Expected output is: 1.
Added:20.0, 46.3, 1.0 Expected output is: 1.
Added:25.0, 51.7, 1.0 Expected output is: 0.
Added:30.0, 56.6, 1.0 Expected output is: 0.
Added:20.0, 46.2, 1.0 Expected output is: 0.
```

Loading testing data...

Added:30.0, 56.5, 1.0 Expected output is: 0.
Added:25.0, 51.6, 1.0 Expected output is: 0.
Added:20.0, 46.1, 1.0 Expected output is: 0.
Added:30.0, 56.8, 1.0 Expected output is: 1.
Added:25.0, 51.9, 1.0 Expected output is: 1.
Added:20.0, 46.4, 1.0 Expected output is: 1.

Training perceptron...

Iteration 1

=====

30.0, 0.0, 1.0 expected 0 and got 0.
25.0, 0.0, 1.0 expected 0 and got 0.
20.0, 0.0, 1.0 expected 0 and got 0.
25.0, 51.8, 1.0 expected 1 and got 0.
0 old weight: 0.0 new weight: 2.5
1 old weight: 0.0 new weight: 25.9
old bias weight: 0.0 new bias weight: 0.5
30.0, 56.7, 1.0 expected 1 and got 1.
20.0, 46.3, 1.0 expected 1 and got 1.
25.0, 51.7, 1.0 expected 0 and got 1.
0 old weight: 2.5 new weight: 0.0
1 old weight: 25.9 new weight: 0.049999237
old bias weight: 0.5 new bias weight: 0.0
30.0, 56.6, 1.0 expected 0 and got 1.
0 old weight: 0.0 new weight: -2.738613
1 old weight: 0.049999237 new weight: -28.25
old bias weight: 0.0 new bias weight: -0.5
20.0, 46.2, 1.0 expected 0 and got 0.

Iteration 2

=====

30.0, 0.0, 1.0 expected 0 and got 0.
25.0, 0.0, 1.0 expected 0 and got 0.
20.0, 0.0, 1.0 expected 0 and got 0.
25.0, 51.8, 1.0 expected 1 and got 0.
0 old weight: -2.738613 new weight: -0.23861289
1 old weight: -28.25 new weight: -2.3500004
old bias weight: -0.5 new bias weight: 0.0
30.0, 56.7, 1.0 expected 1 and got 0.
0 old weight: -0.23861289 new weight: 2.5
1 old weight: -2.3500004 new weight: 26.0
old bias weight: 0.0 new bias weight: 0.5
20.0, 46.3, 1.0 expected 1 and got 1.
25.0, 51.7, 1.0 expected 0 and got 1.
0 old weight: 2.5 new weight: 0.0
1 old weight: 26.0 new weight: 0.14999962
old bias weight: 0.5 new bias weight: 0.0

```
30.0, 56.6, 1.0 expected 0 and got 1.  
0 old weight: 0.0 new weight: -2.738613  
1 old weight: 0.14999962 new weight: -28.15  
old bias weight: 0.0 new bias weight: -0.5  
20.0, 46.2, 1.0 expected 0 and got 0.
```

Iteration 3

=====

```
30.0, 0.0, 1.0 expected 0 and got 0.  
25.0, 0.0, 1.0 expected 0 and got 0.  
20.0, 0.0, 1.0 expected 0 and got 0.  
25.0, 51.8, 1.0 expected 1 and got 0.  
0 old weight: -2.738613 new weight: -0.23861289  
1 old weight: -28.15 new weight: -2.25  
old bias weight: -0.5 new bias weight: 0.0  
30.0, 56.7, 1.0 expected 1 and got 0.  
0 old weight: -0.23861289 new weight: 2.5  
1 old weight: -2.25 new weight: 26.1  
old bias weight: 0.0 new bias weight: 0.5  
20.0, 46.3, 1.0 expected 1 and got 1.  
25.0, 51.7, 1.0 expected 0 and got 1.  
0 old weight: 2.5 new weight: 0.0  
1 old weight: 26.1 new weight: 0.25  
old bias weight: 0.5 new bias weight: 0.0  
30.0, 56.6, 1.0 expected 0 and got 1.  
0 old weight: 0.0 new weight: -2.738613  
1 old weight: 0.25 new weight: -28.05  
old bias weight: 0.0 new bias weight: -0.5  
20.0, 46.2, 1.0 expected 0 and got 0.
```

Iteration 4

=====

```
30.0, 0.0, 1.0 expected 0 and got 0.  
25.0, 0.0, 1.0 expected 0 and got 0.  
20.0, 0.0, 1.0 expected 0 and got 0.  
25.0, 51.8, 1.0 expected 1 and got 0.  
0 old weight: -2.738613 new weight: -0.23861289  
1 old weight: -28.05 new weight: -2.1499996  
old bias weight: -0.5 new bias weight: 0.0  
30.0, 56.7, 1.0 expected 1 and got 0.  
0 old weight: -0.23861289 new weight: 2.5  
1 old weight: -2.1499996 new weight: 26.2  
old bias weight: 0.0 new bias weight: 0.5  
20.0, 46.3, 1.0 expected 1 and got 1.  
25.0, 51.7, 1.0 expected 0 and got 1.  
0 old weight: 2.5 new weight: 0.0  
1 old weight: 26.2 new weight: 0.35000038  
old bias weight: 0.5 new bias weight: 0.0  
30.0, 56.6, 1.0 expected 0 and got 1.  
0 old weight: 0.0 new weight: -2.738613  
1 old weight: 0.35000038 new weight: -27.949999
```

old bias weight: 0.0 new bias weight: -0.5
20.0, 46.2, 1.0 expected 0 and got 0.

Iteration 5

=====

30.0, 0.0, 1.0 expected 0 and got 0.
25.0, 0.0, 1.0 expected 0 and got 0.
20.0, 0.0, 1.0 expected 0 and got 0.
25.0, 51.8, 1.0 expected 1 and got 0.
0 old weight: -2.738613 new weight: -0.23861289
1 old weight: -27.949999 new weight: -2.0499992
old bias weight: -0.5 new bias weight: 0.0
30.0, 56.7, 1.0 expected 1 and got 0.
0 old weight: -0.23861289 new weight: 2.5
1 old weight: -2.0499992 new weight: 26.300001
old bias weight: 0.0 new bias weight: 0.5
20.0, 46.3, 1.0 expected 1 and got 1.
25.0, 51.7, 1.0 expected 0 and got 1.
0 old weight: 2.5 new weight: 0.0
1 old weight: 26.300001 new weight: 0.45000076
old bias weight: 0.5 new bias weight: 0.0
30.0, 56.6, 1.0 expected 0 and got 1.
0 old weight: 0.0 new weight: -2.738613
1 old weight: 0.45000076 new weight: -27.849998
old bias weight: 0.0 new bias weight: -0.5
20.0, 46.2, 1.0 expected 0 and got 0.

...

Iteration 51903

=====

30.0, 0.0, 1.0 expected 0 and got 0.
25.0, 0.0, 1.0 expected 0 and got 0.
20.0, 0.0, 1.0 expected 0 and got 0.
25.0, 51.8, 1.0 expected 1 and got 0.
0 old weight: -289.11926 new weight: -286.61926
1 old weight: 28.007547 new weight: 53.907547
old bias weight: -12.0 new bias weight: -11.5
30.0, 56.7, 1.0 expected 1 and got 1.
20.0, 46.3, 1.0 expected 1 and got 1.
25.0, 51.7, 1.0 expected 0 and got 1.
0 old weight: -286.61926 new weight: -289.11926
1 old weight: 53.907547 new weight: 28.057547
old bias weight: -11.5 new bias weight: -12.0
30.0, 56.6, 1.0 expected 0 and got 0.
20.0, 46.2, 1.0 expected 0 and got 0.

Iteration 51904

=====

30.0, 0.0, 1.0 expected 0 and got 0.
25.0, 0.0, 1.0 expected 0 and got 0.

20.0, 0.0, 1.0 expected 0 and got 0.
25.0, 51.8, 1.0 expected 1 and got 0.
0 old weight: -289.11926 new weight: -286.61926
1 old weight: 28.057547 new weight: 53.957546
old bias weight: -12.0 new bias weight: -11.5
30.0, 56.7, 1.0 expected 1 and got 1.
20.0, 46.3, 1.0 expected 1 and got 1.
25.0, 51.7, 1.0 expected 0 and got 1.
0 old weight: -286.61926 new weight: -289.11926
1 old weight: 53.957546 new weight: 28.107546
old bias weight: -11.5 new bias weight: -12.0
30.0, 56.6, 1.0 expected 0 and got 0.
20.0, 46.2, 1.0 expected 0 and got 0.

Iteration 51905

=====

30.0, 0.0, 1.0 expected 0 and got 0.
25.0, 0.0, 1.0 expected 0 and got 0.
20.0, 0.0, 1.0 expected 0 and got 0.
25.0, 51.8, 1.0 expected 1 and got 0.
0 old weight: -289.11926 new weight: -286.61926
1 old weight: 28.107546 new weight: 54.007545
old bias weight: -12.0 new bias weight: -11.5
30.0, 56.7, 1.0 expected 1 and got 1.
20.0, 46.3, 1.0 expected 1 and got 1.
25.0, 51.7, 1.0 expected 0 and got 1.
0 old weight: -286.61926 new weight: -289.11926
1 old weight: 54.007545 new weight: 28.157545
old bias weight: -11.5 new bias weight: -12.0
30.0, 56.6, 1.0 expected 0 and got 0.
20.0, 46.2, 1.0 expected 0 and got 0.

Iteration 51906

=====

30.0, 0.0, 1.0 expected 0 and got 0.
25.0, 0.0, 1.0 expected 0 and got 0.
20.0, 0.0, 1.0 expected 0 and got 0.
25.0, 51.8, 1.0 expected 1 and got 0.
0 old weight: -289.11926 new weight: -286.61926
1 old weight: 28.157545 new weight: 54.057545
old bias weight: -12.0 new bias weight: -11.5
30.0, 56.7, 1.0 expected 1 and got 1.
20.0, 46.3, 1.0 expected 1 and got 1.
25.0, 51.7, 1.0 expected 0 and got 1.
0 old weight: -286.61926 new weight: -289.11926
1 old weight: 54.057545 new weight: 28.207544
old bias weight: -11.5 new bias weight: -12.0
30.0, 56.6, 1.0 expected 0 and got 0.
20.0, 46.2, 1.0 expected 0 and got 0.

Iteration 51907

```
=====
```

```
30.0, 0.0, 1.0 expected 0 and got 0.  
25.0, 0.0, 1.0 expected 0 and got 0.  
20.0, 0.0, 1.0 expected 0 and got 0.  
25.0, 51.8, 1.0 expected 1 and got 1.  
30.0, 56.7, 1.0 expected 1 and got 1.  
20.0, 46.3, 1.0 expected 1 and got 1.  
25.0, 51.7, 1.0 expected 0 and got 0.  
30.0, 56.6, 1.0 expected 0 and got 0.  
20.0, 46.2, 1.0 expected 0 and got 0.
```

It took 51907 loops through the test data to learn the input.

```
Percept: W(-289.11926) - tire inflation (PSI) input  
(getSquareRootInput)
```

```
Percept: W(28.207544) - speed input
```

```
Percept: W(-12.0) - bias input
```

```
Running tests...
```

```
30.0, 56.5, 1.0 expected 0 and got 0.  
25.0, 51.6, 1.0 expected 0 and got 0.  
20.0, 46.1, 1.0 expected 0 and got 0.  
30.0, 56.8, 1.0 expected 1 and got 1.  
25.0, 51.9, 1.0 expected 1 and got 1.  
20.0, 46.4, 1.0 expected 1 and got 1.
```

```
All tests passed!
```

Here's a subset of that run with the decimal point being a comma:

```
Loading perceptron...
```

```
Percept: W(0,0) - tire inflation (PSI) input (getSquareRootInput)
```

```
Percept: W(0,0) - speed input
```

```
Percept: W(0,0) - bias input
```

```
Learn rate is: 0,5.
```

```
Loading training data...
```

```
Added:30,0 0,0 1,0 Expected output is: 0.  
Added:25,0 0,0 1,0 Expected output is: 0.  
Added:20,0 0,0 1,0 Expected output is: 0.  
Added:25,0 51,8 1,0 Expected output is: 1.  
Added:30,0 56,7 1,0 Expected output is: 1.  
Added:20,0 46,3 1,0 Expected output is: 1.  
Added:25,0 51,7 1,0 Expected output is: 0.  
Added:30,0 56,6 1,0 Expected output is: 0.  
Added:20,0 46,2 1,0 Expected output is: 0.
```

```
Loading testing data...
```

```
Added:30,0 56,5 1,0 Expected output is: 0.  
Added:25,0 51,6 1,0 Expected output is: 0.  
Added:20,0 46,1 1,0 Expected output is: 0.
```

```
Added:30,0 56,8 1,0 Expected output is: 1.
Added:25,0 51,9 1,0 Expected output is: 1.
Added:20,0 46,4 1,0 Expected output is: 1.
```

Training perceptron...

Iteration 1

=====

```
30,0 0,0 1,0 expected 0 and got 0.
25,0 0,0 1,0 expected 0 and got 0.
20,0 0,0 1,0 expected 0 and got 0.
25,0 51,8 1,0 expected 1 and got 0.
0 old weight: 0,0 new weight: 2,5
1 old weight: 0,0 new weight: 25,9
old bias weight: 0,0 new bias weight: 0,5
30,0 56,7 1,0 expected 1 and got 1.
20,0 46,3 1,0 expected 1 and got 1.
25,0 51,7 1,0 expected 0 and got 1.
0 old weight: 2,5 new weight: 0,0
1 old weight: 25,9 new weight: 0,049999237
old bias weight: 0,5 new bias weight: 0,0
30,0 56,6 1,0 expected 0 and got 1.
0 old weight: 0,0 new weight: -2,738613
1 old weight: 0,049999237 new weight: -28,25
old bias weight: 0,0 new bias weight: -0,5
20,0 46,2 1,0 expected 0 and got 0.
```

...

Iteration 51907

=====

```
30,0 0,0 1,0 expected 0 and got 0.
25,0 0,0 1,0 expected 0 and got 0.
20,0 0,0 1,0 expected 0 and got 0.
25,0 51,8 1,0 expected 1 and got 1.
30,0 56,7 1,0 expected 1 and got 1.
20,0 46,3 1,0 expected 1 and got 1.
25,0 51,7 1,0 expected 0 and got 0.
30,0 56,6 1,0 expected 0 and got 0.
20,0 46,2 1,0 expected 0 and got 0.
```

It took 51907 loops through the test data to learn the input.

Percept: W(-289,11926) - tire inflation (PSI) input
(getSquareRootInput)

Percept: W(28,207544) - speed input

Percept: W(-12,0) - bias input

Running tests...

```
30,0 56,5 1,0 expected 0 and got 0.
25,0 51,6 1,0 expected 0 and got 0.
20,0 46,1 1,0 expected 0 and got 0.
30,0 56,8 1,0 expected 1 and got 1.
```

```
25,0 51,9 1,0 expected 1 and got 1.  
20,0 46,4 1,0 expected 1 and got 1.  
All tests passed!
```