600.112: Intro Programming for Scientists and Engineers

Assignment 4: Learning from Echocardiograms*

Peter H. Fröhlich phf@cs.jhu.edu
Joanne Selinski joanne@cs.jhu.edu

Due Dates: 3pm Wednesdays 10/7 & 10/14

Introduction

This assignment for *600.112: Introductory Programming for Scientists and Engineers* explores how we can use *data* to make *predictions* about the world. Specifically we'll look at a very simple approach to *supervised machine learning* based on *vector similarity*. In the end, you'll be able to predict (very roughly) for how long someone who has suffered a heart attack will survive based on a few measurements from their *echocardiogram*.

There are three things to do:

- First you'll write a module (*not* a complete program!), called `stats.py`, to perform basic statistics on sequences of numeric values; you'll then *use* this module to write a program `analyze.py` that computes some basic statistics about our echocardiogram data set. (Submit both of these in a zip file for part A on Blackboard.)
- Second you'll write a module called `vecs.py` to perform some basic calculations on vectors, including finding the "closest" or "most similar" vector out of many given vectors.
- Third you'll use your module for vectors from part 2 to write a program called `predict.py` that—given training data from our echocardiogram data set—will predict the survival time for patients *it wasn't trained on*.
- Starting with this assignment we will be using the **doctest** tool to do thorough unit testing on our functions that return values. See part 5 of the posted Installing Python Tools document for instructions on installing a doctest plugin for IDLE, as well as class slides on using it.

You must submit a complete zip file with all *.py and *.txt files needed to run your solutions as detailed below, and on Blackboard before the deadlines. Also, don't forget to include descriptive doc strings for all your programs and the functions in them. It's best to write these as you go along. Always use the "check module" operation in Idle to make sure your files don't have any syntax errors before submitting them. Improperly written doc strings (missing the closing characters for example) can result in a 0 for your program!

*Disclaimer: This is *not* a course in physics or biology or epidemiology or even mathematics. Our exposition of the science behind the projects cuts corners whenever we can do so without lying outright. We are *not* trying to teach you anything but computer science!

Figure 1 Output of the `analyze.py` program.

```
Age statistics

Minimum: 35.0
Median: 61.0
Average: 60.67
Maximum: 80.0
Deviation: 7.86
Variance: 61.78

Survival time statistics

Minimum: 10.0
Median: 27.0
Average: 29.56
Maximum: 57.0
Deviation: 12.05
Variance: 145.09
```

1 Doing Statistics [10 points]

For this problem you will write *two* Python files: First create a module called `stats.py` that will contain a number of functions to perform basic statistics on sequences of numeric values, and second write a program called `analyze.py` that will use `stats.py` to compute statistics on our echocardiogram data set. Please be sure to use the names `stats.py` and `analyze.py` and nothing else! Figure 1 shows what the output of your program will look like for the `training.data` data set.

Let's start with the `stats.py` module. So far we've only ever *used* modules from the Python standard library and the provided protein.py, we've never *written* a module ourselves. Luckily it turns out that writing modules is not very different from writing regular Python programs! In fact the *only* difference is that *all* the code is inside functions, so you cannot "run" a module by itself.

Remember how we introduced functions? Writing a function allowed us to give a name to a piece of code that we wanted to *reuse* again and again: Instead of having to copy and paste the code, we could just call the function. Modules allow us to package together *several related functions* which can then be *reused* again and again. Even better, we can reuse modules in *several* programs whereas we can only reuse functions in *one* program, namely the one we put the function into.

In order to make all this more concrete, let's look at a first version of the `stats.py` module, a version that doesn't really do anything yet but illustrates the basic structure:

```
def average(seq):
    pass

def median(seq):
    pass

def variance(seq):
    pass

def deviation(seq):
    pass
```

What we have here is just a Python file with a bunch of functions in it. And that's all a module really is: A Python file with a bunch of functions in it!

As soon as we have a this `stats.py` file, we can actually *import* it and then call its functions. However, if you launch idle from the programming menu VirtualBox it might not find your module

file automatically. Instead, first launch LXTerminal. Then use the unix `cd` command to change to the directory where your module is saved, for example: `cd ipse/hw04` would navigate to the hw04 folder inside the ipse folder. Now launch idle from the unix command line to run in the background: `idle &` and then proceed to use this new shell to explore your module:

```
>>> import stats
>>> stats.average([])
>>> stats.variance([])
>>> stats.undefined()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  AttributeError: 'module' object has no
  attribute 'undefined'
```

Of course neither `average` nor `variance` do anything yet, but we can at least call them. Note that Python will respond with an error message if we try to call a function that we did *not* actually define in the `stats.py` module. So in other words, if we have a number of functions that we would like to reuse again and again in many programs, we can put them into a module and then later *import* the module to access them.

Now you may wonder how we will *test* a module like `stats.py` if we cannot actually *run* it. The answer is that we'll use *unit tests* in the form we discussed in lecture! For example, here are the documentation as well as some test cases for the `average` function:

```
def average(seq):
    """
    Average value of sequence.

    >>> average([])
    0.0
    >>> average([1] * 100)
    1.0
    >>> average([1, 2, 3])
    2.0
    >>> average([1.0, 2.0, 3.0, 4.0])
    2.5
    """
    pass
```

Of course all these test cases will fail for now, but at least we have a way to make sure the function works as expected. The code to implement the `average` function is not very complicated: we simply have to sum up all the elements in the given sequence and then divide by the length of the sequence. The first "tricky" thing is that we want an empty list to have an average of zero, so we need to handle that with an `if` statement. The other "tricky" thing is that we want to always return a floating point result, so we'll need to insure that the division has at least one floating point operand. We'll use a float type cast on the total to do that. Here is the finished `average` function that passes all its test cases:

```
def average(seq):
    """
    Average value of sequence.

    >>> average([])
    0.0
    >>> average([1] * 100)
    1.0
    >>> average([1, 2, 3])
    2.0
    >>> average([1.0, 2.0, 3.0, 4.0])
    2.5
    """
    size = len(seq)
```

```
    if size == 0:
        return 0.0
    # call built-in Python sum function
    total = sum(seq)
    return float(total) / size
```

Now we can actually play with the function in the interpreter from outside of `stats.py` as well:

```
>>> import stats
>>> stats.average([1, 2, 3])
2.0
>>> stats.average([1.0, 2.0, 3.0, 4.0])
2.5
```

Each of the remaining functions `median`, `variance`, and `deviation` will need documentation and test cases as well, and here they are. Note that for these functions we similarly want to always return a floating point result for the best accuracy. Also note how providing test cases helps to document and clarify the exact behaviour of the function for different parameters.

```
def median(seq):
    """
␣␣␣␣Median␣value␣of␣a␣sequence.␣For␣a
␣␣␣␣sequence␣of␣odd␣length,␣the␣median
␣␣␣␣is␣the␣middle␣element␣of␣the␣sorted
␣␣␣␣sequence;␣for␣a␣sequence␣of␣even
␣␣␣␣length,␣the␣median␣is␣the␣average
␣␣␣␣of␣the␣two␣middle␣values.

␣␣␣␣>>>␣median([])
␣␣␣␣0.0
␣␣␣␣>>>␣median([1])
␣␣␣␣1.0
␣␣␣␣>>>␣median([1,␣1])
␣␣␣␣1.0
␣␣␣␣>>>␣median([1,␣1,␣2,␣3,␣10])
␣␣␣␣2.0
␣␣␣␣>>>␣median([3,␣1,␣10,␣1,␣1])
␣␣␣␣1.0
␣␣␣␣>>>␣median([1,␣1,␣2,␣4,␣10,␣12])
␣␣␣␣3.0
␣␣␣␣>>>␣median([1,␣1,␣4,␣4,␣10,␣12])
␣␣␣␣4.0
␣␣␣␣"""
    pass

def variance(seq):
    """
␣␣␣␣Variance␣of␣a␣sequence.␣The␣variance
␣␣␣␣of␣sequence␣S␣is␣the␣average␣of␣the
␣␣␣␣squared␣distances␣of␣each␣element␣of
␣␣␣␣S␣from␣the␣average␣of␣S.

␣␣␣␣>>>␣variance([])
␣␣␣␣0.0
␣␣␣␣>>>␣variance([1])
␣␣␣␣0.0
␣␣␣␣>>>␣variance([1,␣1,␣1,␣1])
␣␣␣␣0.0
␣␣␣␣>>>␣round(variance([1,␣2,␣3,␣4,␣5,␣6]),4)
␣␣␣␣2.9167
␣␣␣␣>>>␣variance([1,␣2,␣3,␣4,␣5,␣6,␣7])
␣␣␣␣4.0
```

```
    >>> round(variance([1.0, 2.0, 3.0, 4.0, 5.0, 6.0]), 3)
    2.917
    """
    pass

def deviation(seq):
    """
    Standard deviation of a sequence.
    The standard deviation of sequence
    S is the square root of the variance
    of S.

    >>> round(deviation([]), 1)
    0.0
    >>> round(deviation([1]), 1)
    0.0
    >>> round(deviation([1, 2, 3, 4, 5, 6, 7]), 1)
    2.0
    """
    pass
```

We won't repeat all of these again, so make sure you check back when we discuss how to write the code for each function.[1]

Now let's write the code for the `median` function. First note that we need a sorted sequence in order to properly compute the median value. We can't assume that the parameter value is sorted already. Next note that we need to calculate the median of a sequence of *even* length differently from the median of a sequence of *odd* length. One way to check if an integer is even or odd is to divide it by 2 and look at the *remainder* of that division: If the remainder is 0 the integer was even, if it is 1 the integer was odd. In Python, the *modulo operator* `%` computes (for positive integers anyway) the remainder. This leads to the following code:

```
def median(seq):
    # call built-in python length function
    size = len(seq)
    if size == 0:
        return 0.0
    # call built-in python sorting function
    # save result with a new name
    sseq = sorted(seq)
    mid = size // 2  # integer division
    if size % 2 == 1:  # odd length
        return float(sseq[mid])
    else:
        # call the average function we wrote
        return average(sseq[mid - 1:mid + 1])
```

Note that we can call our `average` function from within the `median` function since they are both defined in the same module!

The `variance` function sounds much more complicated at first. We need to compute an average, then compute the squared distance of each element in the sequence from said average, and then finally average all of those squared distances again. However, with the `average` function at our disposal, two of those three steps become very simple. Computing *one* squared distance is also easy, but note that we need squared distances for *all* the original values! We could use a loop to compute this, but there's an easier way: We simply build a new sequence using the `append` function on lists, and then call our average function on it! It may come as a bit of a surprise, but the resulting code for `variance` is

[1] Notice that we're using the `round` function in some of the test cases. Given a floating point value `x`, the call `round(x, 4)` will return `x` rounded to four significant digits after the decimal point. In our test cases, `round` allows us to write down only the first few digits of a potentially very long floating point value.

actually *shorter* than either of the previous two functions:

```
def variance(seq):
    a = average(seq)
    s = []  # empty list
    for x in seq:
        s.append((x - a) ** 2)
    return average(s)
```

Finally we have to write the code for `deviation` which requires that we compute a square root. We could write code for that task, but Python actually comes with a `math` module that provides a `sqrt` function for exactly that purpose. So what we'll do is we'll *import* the `math` module into the `stats` module we are writing! Once we do that, `deviation` also becomes very easy to write. Here are the missing pieces of the module as well as the `deviation` function itself:

```
"""Basic statistics for sequences."""

import math

...

def deviation(seq):
    return math.sqrt(variance(seq))
```

And we're done with the `stats.py` module! At this point you may want to run all the test cases one more time.

After all this code it may be hard to recall that we're not done yet: We still need to write the `analyze.py` program that produces the output shown in Figure 1. It's clear that we'll use the functions we have in `stats.py` to do the actual statistics, but what does the input look like? The first three lines of the file `training.txt` that is posted on the assignment page look like this:

```
71 11 0 0.260 9 4.600 1
72 19 0 0.380 6 4.100 1.700
55 16 0 0.260 4 3.420 1
...
```

Each line contains seven numbers, and each of these numbers has a certain meaning. As a reminder, the data set describes patients who had a heart attack. The first number on each line is the *age* at which a patient suffered their heart attack. The second number of each line corresponds to the *months* they were alive after their heart attack. So the first patient had a heart attack at 71 years of age and died 11 months after the heart attack.[2] The remaining five numbers describe the condition of the patient's heart in more detail; for us it won't be important what all the numbers mean, but just in case you are interested:

3rd Pericardial effusion (1=yes 0=no)

4th Fractional shortening

5th E-point septal separation

6th Left ventricular end-diastolic dimension

7th Wall motion index

For the `analyze.py` program we actually only need the age and the survival time of each patient.

At this point it's a good idea to develop a basic outline for the program by thinking through what we'll have to do. We'll certainly need to *read in* the `training.txt` file line by line (something we've done before). From each line, we need to *extract* the patient's age and survival time. Since we

[2]The exact cause of death is not necessarily related to the heart condition, but that's something we'll conveniently ignore.

are supposed to compute statistics over all patients, and since our functions in `stats.py` expect that we pass in a list, we'll have to *build* two lists: One containing all the ages, the other containing all the survival times. The output we're required to produce is almost identical for both ages and survival times, so we'll write a *function* instead of copy/pasting the same code twice. Finally the output should only have two digits after the decimal point for each value, something we'll have to take care of somewhere as well; presumably we'll write yet another *function* for that purpose.

With a rough plan in hand, we can write the first version of the program. Let's start by just reading in the file and printing it back out:

```
def main():
    data = open("training.txt")
    for line in data:
        line = line.strip()
        print line
    data.close()

main()
```

As always, this first version doesn't do anything close to what we want to end up with, but it gives us a starting point that actually works.[3] The `line` variable holds a complete line as a single string, but we need to extract individual fields. Luckily Python strings have a very convenient function for this purpose:

```
>>> "Here is a string".split()
['Here', 'is', 'a', 'string']
```

The `split` function breaks apart a string on whitespace, returning a list of the individual "words" (more generically called "tokens") in the string. This will work just the same on one of our lines, and we'll be able to access individual fields by indexing into the list as follows:

```
>>> fs = "71 11 0 0.260 9 4.600 1".split()
>>> fs
['71', '11', '0', '0.260', '9', '4.600', '1']
>>> fs[0]
'71'
>>> fs[1]
'11'
```

One more thing to note: We get back the data we want in the form of strings, but of course we want to have them in the form of actual numbers to perform our statistical computations on them. So we end up with this next version of our program:

```
def main():
    data = open("training.txt")
    for line in data:
        line = line.strip()
        fields = line.split()
        age = float(fields[0])
        time = float(fields[1])
        print age, time
    data.close()

main()
```

Now we need to add the lists that collect all the survival times and all the ages of the various patients instead of printing each individual value:

[3] To save space, we won't include any documentation in the following code examples. Of course the version you hand in should have all the required documentation in place.

```
def main():
    ages = []
    times = []

    data = open("training.txt")
    for line in data:
        line = line.strip()
        fields = line.split()
        age = float(fields[0])
        time = float(fields[1])
        ages.append(age)
        times.append(time)
    data.close()

    print ages
    print times

main()
```

This is all the processing we need to do on the data we read in. Now that we have two lists of floating point values, we can compute statistics on them using the `stats` module we wrote before. So we have to turn our attention to doing that and also to formatting the output. For each list of values, we need to print the same sorts of statistics, just with a different heading in front of the numbers. So we'll write a function `print_statistics` that takes two parameters: The heading to print, and the list of data to process. Let's write a first version of that function so that we can finish the `main` program:

```
def print_statistics(name, values):
    print name
    print values

def main():
    ages = []
    times = []

    data = open("training.txt")
    for line in data:
        line = line.strip()
        fields = line.split()
        age = float(fields[0])
        time = float(fields[1])
        ages.append(age)
        times.append(time)
    data.close()

    print_statistics("Age", ages)
    print_statistics("Survival time", times)

main()
```

If you look at Figure 1 closely, you'll notice that we need to add the "statistics" after the heading itself. Then we need to compute and print each of the following values for the data in question: minimum, maximum, average, median, variance, and standard deviation. Each of these values is printed with at most two digits after the decimal point. Let's attack that last aspect first and write a function that prints one line of the actual data output:

```
def print_value(name, value):
    print name + ":", round(value, 2)
```

So what we hand this function is a name like "Maximum" and the corresponding value, and the function will take care of formatting everything properly. With this in place, it's easy (if tedious) to finish the

`print_statistics` function itself:

```
import stats

...

def print_statistics(name, values):
    print
    print name, "statistics"
    print
    print_value("Minimum", min(values))
    print_value("Median", stats.median(values))
    print_value("Average", stats.average(values))
    print_value("Maximum", max(values))
    print_value("Deviation", stats.deviation(values))
    print_value("Variance", stats.variance(values))
```

Don't forget to add `import stats` at the top of your program so that our statistics functions can actually be accessed. And done!

2 Vector Tools [25 points]

The second thing you will write is a module called `vecs.py` that contains several functions operating on vectors. For the rest of this assignment, we'll consider lists of floating point numbers to be "vectors" in certain contexts, and we mean "vectors" in the sense of mathematical vectors, i.e. quantities with a length and an orientation. The origin of each vector is the origin of the axes, `[0.0, 0.0]` for 2-dimensional vectors, `[0.0, 0.0, 0.0, 0.0]` for 4-dimensional vectors, etc. So you can think of the length as the distance from the origin. For example, the lists `[0.0, 1.0]` and `[1.0, 0.0]` are the (two-dimensional) *unit vectors* (length 1) along the x and y axes, respectively. Obviously a list with n values corresponds to an n-dimensional vector; don't be freaked out by this, vector is vector regardless of how many dimensions it has.

The module `vecs.py` will provide the following functions:

dot(a, b) is the dot product (scalar product) of vectors a and b (calculated as $\sum(a_i * b_i)$)

length(a) is the (Euclidian) length of vector a (the same as $\sqrt{\text{dot(a, a)}}$ it turns out)

cosine_distance(a, b) is the cosine of the angle between vectors a and b (calculated by $\cos\phi = \frac{\text{dot(a, b)}}{\text{length(a)length(b)}}$)

angle_distance(a, b) is the angle (in radians) between vectors a and b (see above, just use `math.acos` to get the angle from the cosine)

squared_euclidian_distance(a, b) is the sum of squared component distances $\sum(a_i - b_i)^2$

euclidian_distance(a, b) is the square root of the above sum

If you're not familiar with any of these concepts, you can get some explanations with pictures at www.mathisfun.com and en.wikipedia.org.

Finally, and most importantly, the module must provide a function which is intended to find and return the vector in a data list that most closely matches a particular sample vector according to a specified distance metric. This requires a more detailed explanation:

find_closest_vector(data, sample, distance)

Here **data** is a *sequence of vectors* while **sample** is a *single* vector. All vectors involved must have the same dimensionality, otherwise the operation won't work correctly. The **distance** parameter tells the

function which of the various *distance functions* we've written above to use. In other words, the code inside of **find_closest_vector** doesn't call any of the previously defined distance functions directly, it calls whichever function was passed in under the name **distance**. What the function computes and returns is the *index* of the vector in **data** that has the smallest *distance* to the **sample** vector. For example:

```
>>> import vecs
>>> vecs.find_closest_vector(
... [[0.0, 1.0], [1.0, 0.0]],
... [0.1, 0.7],
... vecs.angle_distance)
0
>>> vecs.find_closest_vector(
... [[0.0, 1.0], [1.0, 0.0]],
... [0.2, 0.1],
... vecs.angle_distance)
1
```

In the first case, the angle between the **sample** and the x-axis is smaller, so the index 0 indicating the x-axis (first vector in the data list) is returned. In the second case, the angle between the **sample** and the y-axis is smaller, so the index 1 indicating the y-axis (second vector in the data list) is returned.

Notice that we can nest multiple lists (vectors) inside another sequence, in this case our **data** set. If you remember that a sequence can contain values of any type, then it stands to reason that it can in fact contain other sequences. All the brackets and commas can be confusing though. Consider this sample interaction:

```
>>> vec1 = [1.0, 2.0, 3.0]
>>> vec2 = ['a', 'b', 'c', 'd']
>>> seq = [vec1, -32, vec2, "end"]
>>> seq
[[1.0, 2.0, 3.0], -32, ['a', 'b', 'c', 'd'], 'end']
>>> seq[0]
[1.0, 2.0, 3.0]
>>> seq[2]
['a', 'b', 'c', 'd']
>>> vec1[1]
2.0
>>> seq[0][1]
2.0
>>> seq[3][0]
'e'
```

Here, our sequence **seq** contains a sequence of floats (**vec1**) as its first element, then a single integer, followed by a sequence of characters **vec2**, and lastly a string (which, don't forget, is also a sequence). We can access whole elements of **seq**, and also individual elements of its nested sequences. We'll work more with nested sequences in future assignments as well.

Implementing this function means that you have to compute the distance between the **sample** and *each* vector in **data**; you need to keep track of the smallest distance so far and of the *index* in **data** for which that smallest distance was computed; when you have tried every possible choice from **data** you'll know which vector is closest to the **sample** and you can return that vector's index in **data**.

Here is the first version of the module that you should start from:

```
def dot(a, b):
    pass
```

```
def length(a):
    pass

def squared_euclidian_distance(a, b):
    pass

def euclidian_distance(a, b):
    pass

def cosine_distance(a, b):
    pass

def angle_distance(a, b):
    pass

def find_closest_vector(data, sample, distance):
    pass
```

Remember that you'll need to add not only the code but also the documentation and the test cases! Since many of these functions will return floating point values, it's important that you use the `round` function again in your test cases (see the example of this in Part 1). For example, here is a test case "for free" that illustrates why this is important:

```
>>> round(angle_distance(
... [0.0, 1.0], [0.0, -1.0]), 3)
3.142
```

If you didn't include the `round` you'd have to type π to a ridiculous number of digits in order to make the test case pass, something you don't want to do.

By the way, the longest of these functions requires 8 lines of code, many only require one. The hard part for this problem is to come up with the appropriate test cases! Don't just have one test case, that's certainly not enough to make sure that a function works.

A suggestion: In lab, write the test cases for a function first, discuss them with your TA, then write the code to make the test cases pass; do this one function at a time. The majority of the points for this part of the assignment are for your (extensive) test cases, not the actual code!

3 Predicting Survival [15 points]

The third program you will write is going to use our echocardiogram data set to "learn" how patient age and the various heart measurements relate to the survival time of a patient after suffering a heart attack. Please call your program `predict.py` and nothing else! Figure 2 shows what the output of your program will look like, at least approximately.

The approach we'll take for our predictions is rather naive. We'll read the `training.txt` data set and for each patient we'll store their survival time in one list and the *remaining* data *as a vector of floats* in another list. This will require a little bit of sequence manipulation since the survival time is the second element in each line of input. We'll then read the `validate.txt` data set and for each new patient we'll *predict* their survival time by finding the *closest vector* to their measurement data in our training data. When searching for the closest vector in the training data, you should only use the age and echocardiogram measurements from each new patient (not their survival time since this is what we are trying to predict). Since you have to do the same sequence manipulation for the training set and the new set of patients, you should write a function for that.

Say that for the first patient from `validate.txt` we find that the 7th vector of our training data set is closest; that means that our predicted survival time would be the 7th survival time we read in earlier. In other words, when we're presented with a new set of echocardiogram measurements (and patient age)

Figure 2 Output of the `predict.py` program using squared Euclidian distance for similarity.

```
Survival times

Predicted: 53.0 Actual: 26.0 Error: 1.04
Predicted: 53.0 Actual: 12.0 Error: 3.42
Predicted: 25.0 Actual: 49.0 Error: 0.49
Predicted: 53.0 Actual: 49.0 Error: 0.08
Predicted: 12.0 Actual: 47.0 Error: 0.74
Predicted: 40.0 Actual: 41.0 Error: 0.02
Predicted: 25.0 Actual: 33.0 Error: 0.24
Predicted: 44.0 Actual: 29.0 Error: 0.52
Predicted: 50.0 Actual: 41.0 Error: 0.22
Predicted: 34.0 Actual: 26.0 Error: 0.31
Predicted: 27.0 Actual: 15.0 Error: 0.8

Error statistics

Minimum: 0.02
Median: 0.49
Average: 0.72
Maximum: 3.42
Deviation: 0.91
Variance: 0.82
```

we find the closest set of echocardiogram measurements (and patient age) for which we already know the survival time and we predict that.

However, just spitting out predictions is not enough: We would like to know *how good* our predictions actually are! Luckily the second data set *also* has survival times, namely the *actual* survival time for that patient. By comparing our predicted survival time to the actual survival time for all new patients, we can get some idea of whether our approach is actually better than just flipping coins at random. So for each new patient we compute the *relative error*

$$e = \frac{|\text{predicted} - \text{actual}|}{\text{actual}}$$

we made, and we collect all the errors and finally compute some statistics as shown at the end of Figure 2. Apparently we're not doing too well, at least not when using the squared Euclidian distance metric: We're 70% wrong on average, but at least in the median we're slightly better than just flipping coins (but see below).

Note that it's important that none of the patients from our second data set are in the first data set! If they had been, we'd always produce perfect predictions because for one of our many vectors, the distance to a "new" patient would be zero. This process of using one data set for *training* and another one for measuring how good our predictions are is called *cross-validation*; without cross-validation and some evidence that we're actually making decent predictions, we may just as well save ourselves the trouble of making predictions at all.

Obviously the current approach to predicting survival times is not too great. What you should do once you have finished your program is play with the various distance metrics that the `vecs.py` module offers. See if you can find a metric that does better. If you do, try to figure out *why* it does better and talk about your insights as a comment in your program file.