# FinalReview

July 26, 2017

## 1 Final Exam Review

CSCI 1360E: Foundations for Informatics and Analytics

### 1.1 Material

- Anything in *all lectures* is fair game!
- Anything in *all assignments* is fair game!
- ...but there will be a *heavy preference* for everything after the midterm!

### 1.2 Topics

Data Science - Definition - Intrinsic interdisciplinarity - "Greater Data Science"

Python Language - Philosophy - Compiled vs Interpreted - Variables, literals, types, operators (arithmetic and comparative) - Casting, typing system - Syntax (role of whitespace)

Data Structures - Collections (lists, sets, tuples, dictionaries) - Iterators, generators, and list comprehensions - Loops (`for`, `while`), loop control (`break`, `continue`), and utility looping functions (`zip`, `enumerate`) - Variable unpacking - Indexing and slicing - Differences in indexing between collection types (tuples versus sets, lists versus dictionaries)

Conditionals - `if` / `elif` / `else` structure - Boolean algebra (stringing together multiple conditions with `or` and `and`)

Exception handling - `try` / `except` structure, and what goes in each block

Functions - Defining functions - Philosophy of a function - Defining versus calling (invoking) a function - Positional (required) versus default (optional) arguments - Keyword arguments - Functions that take any number of arguments - Object references, and their behaviors in Python

NumPy - Importing external libraries - The NumPy `ndarray`, its properties (`.shape`), and indexing - NumPy submodules - Vectorized arithmetic in lieu of explicit loops - NumPy array dimensions, or *axes*, and how they relate to the `.shape` property - Array broadcasting, uses and rules - Fancy indexing with boolean and integer arrays

File I/O - Reading from / writing to files - Gracefully handling file-related errors

Linear Algebra - Vectors and matrices - Dot products and matrix multiplication - Dimensions of *data* versus dimensionality of *NumPy arrays*

Probability and Statistics - Axioms of Probability - Dependence and independence - Conditional probability - Bayes' Theorem - Probability distributions - First-, second-, and higher-order statistics

Data Visualization and Exploration - Plotting data (line plots, scatter plots, histograms) - Plotting images, or matrices *as* images (colormaps) - Strategies for visualizing data of different dimensions - Accounting for missing data - Matplotlib, pandas

Natural Language Processing - Bag of words model - Preprocessing (stop words, stemming) - TF-IDF

Image Processing - Pixel representation (RGB, grayscale) - Thresholding - Convolutional filters (blurring and sharpening) - Segmentation

Machine Learning - Supervised versus unsupervised learning - Classification algorithms (KNN, SVM) - Clustering algorithms (K-means, Spectral) - Bias-variance trade-off - Cross-validation - Training and testing

Open Data Science - Pillars of Open Science - Anatomy of an open source project - Importance of licensing

### 1.3 Final Exam Logistics

- The format will be just like the midterm. That is to say: very close to that of JupyterHub assignments (there may or may not be autograders to help).
- It will be **180 minutes**. Don't expect any flexibility in this time limit, so plan accordingly.
- You are **NOT** allowed to use internet resources or collaborate with your classmates (enforced by the honor system), but you **ARE** allowed to use lecture and assignment materials from this course, as well as terminals in the JupyterHub environment or on your local machine.
- I will be available on Slack for questions most of the day tomorrow, from 9am until about 3pm (then will be back online around 4pm until 5pm). Shoot me a direct message if you have a conceptual / technical question relating to the final, and I'll do my best to answer ASAP.

### 1.4 JupyterHub Logistics

- The final will be released on JupyterHub at **12:00am on Friday, July 28**.
- It will be collected at **12:00am on Saturday, July 29**. The release and collection will be done by automated scripts, so believe me when I say there won't be any flexibility on the parts of these mechanisms.
- Within that 24-hour window, you can start the exam (by "Fetch"-ing it on JupyterHub) whenever you like.
- **ONCE YOU FETCH THE FINAL, YOU WILL HAVE 180 MINUTES--3 HOURS--FROM THAT MOMENT TO SUBMIT THE COMPLETED FINAL BACK.**
- Furthermore, it's **up to you** to keep track of that time. Look at your system clock when you click "Fetch", or use the timer app on your smartphone, to help you track your time use. Once the 180 minutes are up, the exam is considered late.
- In theory, this should allow you to take the final when it is most convenient for you. Obviously you should probably start no later than 9:00PM on Friday, since any submissions after midnight will be considered late, even if you started at 11:58PM.

## 1.5 Tough Assignment Questions and Concepts

### 1.5.1 From A5

**Do not as I say, and not as I do?** Ok, for some reason, a lot of people outright ignored directions on this homework and lost points as a result.

In Part A, you computed the dot product of two arrays. Some people used loops; this was both explicitly disallowed in the directions, *and* it's more work than using NumPy array broadcasting!

```
In [1]: def dot(arr1, arr2):
            if arr1.shape[0] != arr2.shape[0]:
                return None
            p = arr1 * arr2  # Multiplies corresponding elements of the two arrays...no loops ne
            s = p.sum()      # Computes the sum of all the elements...still no loops needed!
            return s
```

Another example is Part G, asking you to write a function that reverses the elements of an array. According to the instructions, you were **not allowed** to use either the `[::-1]` notation, or the `.reversed()` function. Doing so anyway means you missed this bit of `range` cleverness:

Recall that `range` can take up to three arguments: a starting point, an ending point, and an increment. 99% of the time, you're starting at 0 and incrementing by 1, so the only argument you ever really give it is the length.

But if you think about it, you could use this to *start* at the *end*, *end* at the *start*, and *increment by -1*.

Even cooler: you could turn this `range` object into a list of indices, and then use that to *fancy index* the original array, effectively reversing it in one step. Observe:

```
In [1]: def reverse_array(arr):
            start = arr.shape[0] - 1  # We're STARTING at the END
            stop = -1  # Recall that range() always stops at "index - 1"
            step = -1  # Since we start at the end, we increment by -1

            # np.arange is the same as range(), just auto-returns a NumPy array
            # You could use range() here, too; you'd have to wrap it in list()
            indices = np.arange(start, stop, step)
            print(indices)

            reversed = arr[indices]    # Literally just...fancy index.
            return reversed

In [2]: import numpy as np
        before = np.array([10, 20, 30, 40, 50])
        after = reverse_array(before)
        print("Before reversal: ", before)
        print("After reversal: ", after)
```

```
[4 3 2 1 0]
Before reversal:  [10 20 30 40 50]
After reversal:  [50 40 30 20 10]
```

**Polite `imports`** This isn't technically an error, but rather a very strong convention. Whenever you are importing modules, these imports should **all go in one place: the very top of the file.**

There was a common tendency to have import statements inside of functions. This won't cause problems in terms of bugs, but does get very, very confusing if you're dealing with more than 1 function. After all, if you `import numpy as np` inside of one function, it may or may not be available in another function.

So, rather than try to figure out if you need to import NumPy over and over in every single function... just do all your `import` statements once at the very top of your program.

**`try/except` overenthusiasm** It's great to see you know when to use `try/except` blocks! However--and this is very common--there's such as thing as *too much* use of these blocks.

Here's what happens if you put too much of your code inside a `try` block: it **catches, and therefore effectively HIDES, errors and bugs in your code that have nothing to do with the errors you're trying to handle gracefully.**

Two guidelines for using `try/except` blocks:

- Keep the amount of code under a `try` statement to an *absolute minimum*. For example, if you're reading from a file, put only the calls to `open()` and `read()` inside the `try` block.
- *Always* give an error *type* that you're trying to handle; that way, if an unexpected error of a different type crops up, the block won't inadvertently hide it. It's easy enough to simply say `except:`, but I strongly urge you to specify the error type you're trying to handle.

One final note here: **please, please, please, do not EVER nest `try/except` statements.** This will only multiply the problems I've described. If you're trying to handle multiple types of errors, use multiple `except` statements.

### 1.5.2 From A6 (but also A3, so this is copy/pasted from the Midterm Review)

**`if` statements don't always need an `else`.**

I saw this a lot:

```
In [10]: def list_of_positive_indices(numbers):
             indices = []
             for index, element in enumerate(numbers):
                 if element > 0:
                     indices.append(index)
                 else:
                     pass  # Why are we here? What is our purpose? Do we even exist?
             return indices
```

`if` statements are adults; they can handle being short-staffed, as it were. If there's literally nothing to do in an `else` clause, you're perfectly able to omit it entirely:

```
In [11]: def list_of_positive_indices(numbers):
             indices = []
             for index, element in enumerate(numbers):
                 if element > 0:
                     indices.append(index)
             return indices
```

### 1.5.3 From A7 (but also A4, so this is copy/pasted from the Midterm Review)

`len(ndarray)` **versus** `ndarray.shape`

For the question about checking that the lengths of two NumPy arrays were equal, a lot of people chose this route:

```
In [19]: # Some test data
         import numpy as np
         x = np.random.random(10)
         y = np.random.random(10)

In [20]: len(x) == len(y)

Out[20]: True
```

which works, but only for *one-dimensional arrays*.

For anything other than 1-dimensional arrays, things get problematic:

```
In [21]: x = np.random.random((5, 5))   # A 5x5 matrix
         y = np.random.random((5, 10))   # A 5x10 matrix

In [22]: len(x) == len(y)

Out[22]: True
```

These definitely are **not** equal in length. But that's because `len` doesn't measure *length* of matrices...it only measures the number of rows (i.e., the first axis--which in this case is 5 in both, hence it thinks they're equal).

You definitely want to get into the habit of using the `.shape` property of NumPy arrays:

```
In [7]: x = np.random.random((5, 5))   # A 5x5 matrix
        y = np.random.random((5, 10))   # A 5x10 matrix

In [8]: x.shape == y.shape

Out[8]: False
```

We get the answer we expect.

**The dangers of** `in`

```
In [10]: l = [2985, 42589, 13, 574, 57425, 574]

         225 in l

Out[10]: False
```

The keyword `in` is a great tool for testing if some value is present in a collection. But it has a potential downside.

Think of it this way: how would you implement `in` yourself? Probably something like this:

```
In [8]: def in_function(haystack, needle):
            for item in haystack:
                if item == needle:
                    return True
            return False
```

```
In [9]: h1 = [10, 20, 30, 40, 50]
        n1 = 20
        print(in_function(h1, n1))
```

```
True
```

```
In [10]: n2 = 60
         print(in_function(h1, n2))
```

```
False
```

Note that this function **requires a loop**.

So here, then, is the danger: if you're searching for a specific item in a *very large collection*, this can take awhile. Even more dangerous if you're running this search *inside of another loop*.

This created a few failed autograder tests in A7, because JupyterHub automatically kills a test if it takes too long. It's entirely possible your code was correct in theory--and I tried to distribute points accordingly--but it nonetheless lost you points at first.

Something to keep in mind: if you're already using a loop, chances are you probably don't need to use `in`.

**Dependent or Independent?** The question related to coin flips--two random variables $X$ and $Y$, counting the number of heads and tails, respectively--confused a lot of folks.

Several said they were independent variables because "each coin flip is an independent event." That is true, but it is the correct answer **to the wrong question.** The question isn't if each coin flip is independent, but if the number heads and tails in 1000 flips are independent.

Let's say, out of 1000 flips, you observed 600 heads, so you know $X = 600$. Does this give you any information about $Y$, the number of tails?

**Oh yes: you know immediately that $Y = 400$.** This is the very definition of **dependent variables**: if you can directly compute one from the other, that means knowing one gives you information about the other.

### 1.5.4 From A8

**Features, features everywhere** There was a *lot* of trouble with Part F of the homework: implementing `featurize`, which entailed computing a matrix of word frequencies.

It was explained that the number of rows of this matrix should correspond to the number of documents (books), and the number of columns correspond with the number of unique words across all the documents.

To compute the dimensions of this matrix took two steps: 1. Iterate through the number of books/documents. 2. Combine all the words from the books/documents together, and count how many *unique* words there are.

```
In [11]: import numpy as np

         def featurize(*books):
             rows = len(books)  # This is your number of rows.

             vocabulary = global_vocabulary(books)  # Your function from Part E!
             cols = len(vocabulary)  # This is your number of columns.

             feature_matrix = np.zeros(shape = (rows, cols))  # Here's your feature matrix.
```

Of course, that code isn't complete: you still have to fill in the features, which are the word counts.

To do this, you can use your `word_counts` function from Part B. But you have to build a word count dictionary for *each book, one at a time.*

Once you have the word count dictionary for a book, you'll then have to loop through the words *in that book* and add the counts to the right column of the feature matrix you built.

```
In [12]: import numpy as np

         def featurize(*books):
             rows = len(books)  # This is your number of rows.

             vocabulary = global_vocabulary(books)  # Your function from Part E!
             cols = len(vocabulary)  # This is your number of columns.

             feature_matrix = np.zeros(shape = (rows, cols))  # Here's your feature matrix.

             # Loop through the books, generating a word count dictionary for each.
             for row_index, book in enumerate(books):  # enumerate() is very helpful here
                 wc = word_counts(book)
                 for word, count in wc.items():  # This loops through the words IN THIS BOOK.

                     # Which column does this go in?
                     col_index = vocabulary.index(word)

                     # Insert the count!
                     feature_matrix[row_index, col_index] = count

             return feature_matrix
```

### 1.5.5 From A9

Remember the point of *optional/default* arguments: they're *optional*, so you don't have to specify them when you call a function; if you don't, they take on a pre-defined *default* value.

In Q1, Part C, when calling the `skimage.measure.label()` function, some people were specifying all the default arguments with their default values. This is perfectly ok, but just know that's a lot of typing you don't actually have to do. Whatever default value was specified in the documentation on the website is the value that argument takes if you don't provide it.

#### 1.5.6 From A10

**Bias/Variance trade-off** This is a critical concept to understand. Remember what these two quantities are:

- *Bias* refers to how far the average estimate of the model is from the true, underlying average
- *Variance* refers to how far around this average the predictions of the model spread out

An ideal model has small bias and small variance. In practice it's usually impossible to achieve this, as decreasing one tends to increase the other: you can either make consistent predictions that are off from the true average (high bias, low variance), or scattered predictions that tend to average out to the true value (low bias, high variance).

Every model you ever create will reflect this interplay, explicitly or not. K-nearest neighbors is a great example because this interplay is very explicit as you vary $k$, the neighborhood size.

- When $k$ is small, this means you're using a very small neighborhood of data points to predict a new one. You can imagine how this could give rise to vastly different predictions in different neighborhoods, since the neighborhood used to make a prediction is so small. However, this also means the prediction will be extremely localized and probably more accurate. This is **low bias, but high variance.**

- When $k$ is large, this means you're averaging the votes of a massive neighborhood of data--possibly even the entire dataset. Therefore, it doesn't really matter where the new data point is landing, since its "neighborhood" is pretty much the entire universe, so the prediction will be the same no matter what. This is **high bias, low variance.**

Support Vector Machines (SVMs) also embody this bias-variance tradeoff, but much less explicitly. Their internal algorithm is so much more sophisticated than KNNs that there is no one variable you can use to modulate the trade-off; instead, it's largely hard-coded into SVMs: they're designed to have **high bias, but low variance**. Thhis hard-coding may seem like a disadvantage, but it really depends on the situation.

**Cross-validate all the things!** Cross-validation is an immensely valuable tool that is central to the design of any predictive model. The goal of cross-validation is to estimate how good your model is at *generalizing* to new data.

It works by taking your data and splitting it into *folds*: most of these folds are used to train the model, and the last fold is used to test it.

In the homework, some submission didn't abide by this: they trained on the whole dataset, then tested on the whole dataset. **This is NOT cross-validation:** the training and testing subsets **MUST** be mututally exclusive: for example, use the first 75% of the data as training, and the last 25% as testing.

### 1.6 Other Questions from the Google Hangouts Review Session

#### 1.6.1 Please make sure to fill out the course evaluations!

http://eval.franklin.uga.edu/

#### 1.6.2 Good luck!


goodluck