


CIS531v2: Problem-Solving With Machine Learning



Live Session 1

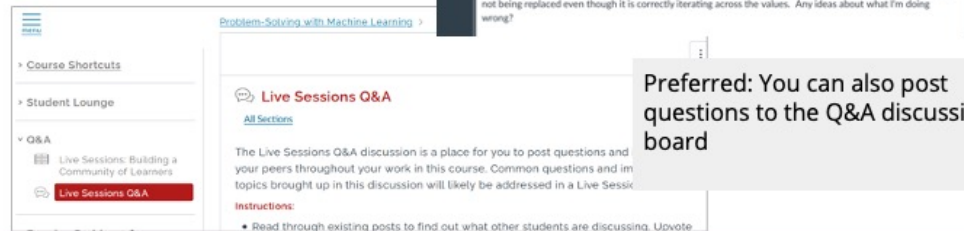
Brian Feeny

Note: This presentation is based substantially on work done by Melynda Eden, eCornell Machine Learning Facilitator, for version 1 of this course.

Brian Feeny, Course Facilitator

How to get assistance:

- Private message me through Canvas



Preferred: You can also post questions to the Q&A discussion board

My name is Brian Feeny, and I am your course facilitator. If you have specific questions for me about the course material, assignments, or questions about project code, send me a message through Canvas. You can post questions or comments about the course material to the Live Sessions Q&A discussion board, but please do not post significant amounts of code, or full or partial solutions.

Today's Live Session

CIS531v2 - Course Info and Overview

Data Science / Machine Learning - features and labels, loss function

k-nearest-neighbors algorithm

Assignments - Class Discussions, Frame a Machine Learning Problem, Applications and limitations of k-NN, Use a Jupyter Notebook

Code - NumPy, Indexing / Slicing, Subsetting, Functions

Lab Project tips & tricks

Assignments for next week - Euclidean Distance and Facial Recognition

CIS531v2: Problem-Solving With Machine Learning

- Rigorous course that includes discussions, written assignments, coding exercises, and a project
- Prior knowledge and familiarity with Python, linear algebra (matrix multiplication), statistics, and basic calculus (derivatives) **is assumed**
- It is recommended that you try to get through modules 1, 2 and 3 the first week, saving the second week for module 4

This course is CIS531 Problem-Solving With Machine Learning. This is the first of seven courses that make up the Machine Learning Certificate. This course will help you reframe real-world problems in terms of supervised machine learning, and the final project of this course is implementing the k-Nearest Neighbors algorithm to build a facial recognition system.

The machine learning courses are rigorous, and require prior experience with Python programming. You should be familiar with linear algebra and understand matrices and matrix operations. There are two self-paced linear algebra courses available to you in Canvas (low-dimension and high-dimension). If you haven't already done so, I encourage you to go through these linear algebra courses, as they will give you a better understanding of matrix operations that are applied in modules 3 and 4 and future courses that make up this certificate program.

Regarding this course, I highly recommend getting through modules 1, 2 and 3 the first week, and having the second week to go through module 4 which includes the Euclidean Distance assignment and the Facial Recognition project. This final project is challenging, and may take some time to complete.

Deeper Dives from Dr. Weinberger

[Lecture 1 "Supervised Learning Setup" -Cornell CS4780 Machine Learning for Decision Making SP17](https://www.youtube.com/watch?v=MrLPzBxG95I)

<https://www.youtube.com/watch?v=MrLPzBxG95I>

[Lecture 2 "Supervised Learning Setup Continued" -Cornell CS4780 SP17](https://www.youtube.com/watch?v=zj-5nkNKAow)

<https://www.youtube.com/watch?v=zj-5nkNKAow>

[Lecture 3 "k-nearest neighbors" -Cornell CS4780 SP17](https://www.youtube.com/watch?v=oymtGIGdT-k)

<https://www.youtube.com/watch?v=oymtGIGdT-k>

[Lecture 4 "Curse of Dimensionality / Perceptron" -Cornell CS4780 SP17](https://www.youtube.com/watch?v=BbYV8UfMJSA&t=2478s)

<https://www.youtube.com/watch?v=BbYV8UfMJSA&t=2478s>

Lecture 4 begins concepts that apply to a future course.

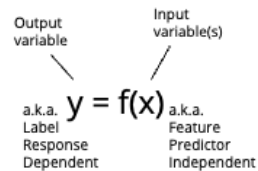
Avoid the Appearance of Plagiarism

<https://drive.google.com/file/d/19w90Kmz8zP4ajS2rfpeL9D03ss3ZN0D2/view?usp=sharing>

Key points:

- Show your work - comment out, do not delete, mistakes, print statements, test cells, etc. Messy notebook that shows original thought is better than pristine code that happens to look like someone else's work
- Reference helpful websites - put links to discussion boards, Q&A forums (Stack Overflow, etc.) that contained information you found useful in developing your code
- When in doubt, ask your facilitator!

Supervised Machine Learning



Classification (thing vs. not thing - draw a boundary)

Regression (predict specific value)

$$y = B_0 + B_1x_1 + B_2x_2 + B_3x_3 + \dots$$

Beta
Offset
Coefficient

Weight
Coefficient

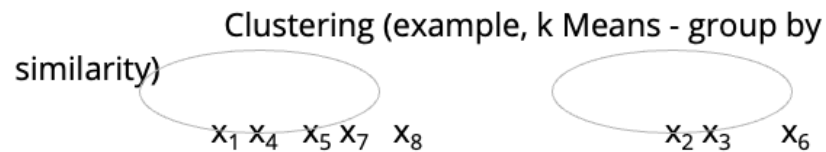
There are different types of machine learning. With supervised learning, your data is labeled - you have input variables (x) and an output variable (Y). Input variables, also called independent variables, inputs, predictors, features - all referring to the same thing. Outputs, also called labels, responses, or dependent variables - all referring to the same thing.

Regression and classification problems are two types of supervised learning problems. With classification, we are looking to classify something into one or more classes. We could classify emails as “spam” or “not spam”. If we have two classes, it is binary classification. We can have more than two classes; for example, we can classify images of fruit: “apples”, “oranges”, “bananas”, and “strawberries” for example. With a regression problem, you are trying to predict something that is a real or continuous value. For example, you might predict the price of a house based on square footage, zip code, number of bedrooms, and number of bathrooms.

Unsupervised Machine Learning

Unlabeled data (so no "y" known)

x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8



With unsupervised learning, there are input variables (x), but the data is not labeled, so no corresponding output variables, so there are no correct answers. Unsupervised learning can be used as an exploratory data analysis technique used to discover the underlying structure of the data or distribution of the data. An example of an unsupervised learning problem can be customer segmentation, or understanding different groups of customers with similar buying habits. You might have lots of customer data, but the data is not divided into named categories, so your data is unlabeled. Unsupervised learning usually involves clustering or association, where you are trying to find similarities within the data set and breaking the data into logical groups. K-means clustering is an example of an unsupervised machine learning algorithm. With k-means clustering, there are no labels, so there are no predefined classes. Given k groups to look for, k-means iterates over the data to find similarities, and segments the data into k clusters, where

each data point belongs to only one group. Data points in each group should be similar, with differences between data points belonging to different groups.

Sometimes people confuse k Means and k-Nearest Neighbors, but they are two different algorithms. k-Nearest Neighbors, which will be learned in depth in this course, is a supervised learning algorithm, and as I just mentioned, k Means is unsupervised. The focus of this course and certificate program is supervised machine learning, and we'll talk more about k-NN in a minute.

.

Data

Training Data - 80%

Validation Data - 10%

Testing Data - 10%

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	1	1/1/21	33742	3337	61632	73896	60537	65075	12942	30147	35162	24314	58802	54221	32784	35283	40941
2	2	1/5/21	98752	31260	22132	71361	12843	47627	56861	48727	44859	22653	4266	39379	5468	96544	76968
3	3	1/5/21	206	64620	13683	30466	10068	61620	47164	10881	91304	76063	46516	64551	66622	67120	7898
4	4	1/6/21	66718	2686	26648	91614	40792	64063	9916	88636	62481	6226	13606	2389	66303	17966	33073
5	5	1/6/21	63463	2211	88889	71863	11343	53619	29726	11893	5832	90413	51762	7407	37795	5963	32960
6	6	1/6/21	34677	18849	44173	78364	22776	7874	75721	24976	88964	33646	243	51420	47952	27712	39013
7	7	1/7/21	65236	20550	66795	27656	33243	47070	56673	15420	9234	89621	54226	62939	66620	9690	50739
8	8	1/6/21	21963	20016	55252	6846	24438	24069	924	56271	65446	76028	84414	40795	66451	46667	31739
9	9	1/6/21	81348	19694	4132	86633	36293	80960	64396	73820	16396	76886	82473	31989	88234	74361	32718
10	10	1/10/21	7996	76460	33293	6738	61773	27841	34914	93896	6469	94663	18096	7236	67470	67199	9646
11	11	1/10/21	80286	76591	68736	94868	52966	12959	21612	39991	64683	4066	42727	41334	85367	66196	87802
12	12	1/10/21	33168	57611	31780	41867	74630	24652	38643	97750	56282	49669	34979	90968	13776	71738	30688
13	13	1/10/21	26209	48202	46437	46480	13738	67090	77167	10667	88667	56163	38734	9168	38642	14678	42217
14	14	1/14/21	61699	48316	11618	65926	98	12846	84966	96294	46694	46662	48231	1236	24663	384	88668
15	15	1/16/21	62168	62961	51881	54638	20783	16737	97406	18426	56263	64697	55181	47626	6496	63719	40196
16	16	1/16/21	75033	16784	47792	36736	48129	6667	28836	73844	24748	84664	27689	19176	21253	66472	96319
17	17	1/17/21	69279	18625	68330	26536	62281	81447	34317	30162	62077	73664	8356	58821	26631	77868	72556
18	18	1/18/21	33308	41808	26128	46978	7186	68372	3787	78994	64272	76576	18873	99670	56267	46366	23128
19	19	1/19/21	7267	17644	84617	69186	83667	18362	88848	68387	46195	96727	63414	16894	64499	61877	91243
20	20	1/20/21	61637	73666	41877	29874	63612	4399	92676	22433	31901	64766	61388	26382	6223	23676	78816
21	21	1/21/21	43866	18186	34653	18789	52893	94966	18649	44279	12379	78886	90111	28171	1613	11218	94148
22	22	1/22/21	79241	88906	31646	21954	67473	56520	56993	86081	20336	63661	81468	3162	67199	56237	73766
23	23	1/23/21	7808	3614	42626	62437	60677	69177	9912	26568	66188	76761	73646	63674	38863	63768	9962
24	24	1/24/21	66179	74694	64244	66670	85032	26944	74997	90921	16064	17062	27076	294	21806	96008	28669
25	25	1/26/21	46227	14371	71620	71120	46861	81772	34691	20240	89131	64473	26214	16341	67996	48637	95322
26	26	1/26/21	20972	10552	96462	96487	46996	8190	64666	2942	63966	76376	64677	60762	63388	76216	13627
27	27	1/27/21	72966	26633	74841	12618	74612	23306	46162	63440	46717	16106	63997	1723	64900	46881	21748
28	28	1/28/21	47826	91006	76296	34689	25392	46230	38816	21996	44679	12460	51092	36610	16795	26429	10671
29	29	1/28/21	39926	83636	46163	34387	63061	62611	16426	99967	86888	22891	99601	43974	77873	71237	99736
30	30	1/30/21	96319	32962	1429	67741	89629	69676	64149	86346	17683	31616	63972	44787	66433	26680	46368

In supervised learning, we take our data set, and we divide our data into training, validation, and testing data.

Data

Training Data - 80%

Validation Data - 10%

Testing Data - 10%

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	1	1/1/21	47144	12122	48175	50656	57066	53866	59679	58408	55414	49472	59151	52200	52640	73857	81883
2	2	1/2/21	32712	54762	23181	65745	38487	97286	31788	81213	41395	50281	47613	58813	85649	18689	50363
3	3	1/3/21	34127	66111	91865	19486	42669	28796	68813	23858	79952	61655	63667	48968	14838	95578	63571
4	4	1/4/21	48815	55421	22827	12134	31628	69642	51855	86551	33155	5223	22318	14349	33412	94994	76213
5	5	1/5/21	16137	79857	28538	72851	37577	23858	76564	49498	93298	39172	98837	37329	67996	73423	34575
6	6	1/6/21	61642	536	48174	17718	96189	38757	31819	14856	36178	57539	85626	88441	15646	66173	29618
7	7	1/7/21	37661	77869	62528	3938	95586	38281	46143	63881	71882	63479	83798	37188	42566	79642	38654
8	8	1/8/21	69089	99183	42534	49574	65887	86791	42283	43282	57911	3859	49579	7647	53795	17516	73978
9	9	1/9/21	82703	23854	92818	8455	95596	24840	52481	39831	53828	63115	24736	68640	97887	33283	95881
10	10	1/10/21	11266	31785	49666	91612	71433	6167	36441	77565	45995	71688	6883	31451	8464	37255	82545
11	11	1/11/21	71384	62642	76177	72717	62836	68637	94858	88814	88137	34777	98823	78548	43968	82864	52123
12	12	1/12/21	58378	49947	64387	39532	45952	35483	43143	34256	67546	12981	15878	8912	41222	68388	96881
13	13	1/13/21	79821	71889	5237	3854	68888	14779	16885	17597	72949	32547	16637	88232	81851	53645	3488
14	14	1/14/21	58549	67858	26836	22261	75433	96273	74789	38992	45529	25489	15623	42188	63243	57786	28861
15	15	1/15/21	35885	58795	9268	79481	48859	81191	46381	51117	88587	53585	34549	98623	94225	6188	4417
16	16	1/16/21	63676	81881	28118	24871	98852	8827	95239	7768	73126	42832	7472	34576	18649	53449	38878
17	17	1/17/21	22938	87315	52159	91272	88863	84892	44039	8452	45886	38487	63857	28881	88332	5593	88718
18	18	1/18/21	62535	14347	75388	383	13181	6526	58637	27514	44848	25456	43445	39617	47883	93227	47228
19	19	1/19/21	74335	76317	32497	57963	93879	3789	53788	53342	75986	58888	64433	63891	8547	72866	58785
20	20	1/20/21	39054	47688	68765	81583	9691	65243	88534	38482	88498	42322	18633	38729	75552	2219	29628
21	21	1/21/21	18886	23849	5314	45132	63838	54893	76432	28762	1858	28881	37288	52188	18864	78844	52168
22	22	1/22/21	18889	26414	81783	62428	63883	77348	94735	28883	63895	98551	27543	12240	53380	84896	83013
23	23	1/23/21	66758	84357	85231	34883	61143	67293	97754	28864	23825	64227	97714	48819	16821	19875	9889
24	24	1/24/21	81244	87710	65889	22587	23275	70183	28878	93637	64885	723	88321	82666	45891	23312	21314
25	25	1/25/21	74818	71717	96579	89312	31878	68335	48648	88188	78527	73812	83342	38119	57842	18154	91841
26	26	1/26/21	58215	5489	18438	38892	42154	8877	11423	58626	12598	18238	24898	49425	72820	28863	18882
27	27	1/27/21	47885	91155	98877	4238	57553	67558	52785	88852	81328	32814	63435	75888	68561	481	61768
28	28	1/28/21	2776	24113	22866	36438	8795	15588	26389	63856	38886	50298	65258	77888	22149	66677	38818
29	29	1/29/21	85115	64848	36135	84288	63883	92277	34887	11488	76327	48651	54886	77883	13211	77882	78523
30	30	1/30/21	98844	98828	64815	93931	73488	2588	76481	25385	98854	68888	11955	27783	21953	32724	5821

Some data scientists just divide data into training and testing, but it is better to also carve out a portion of the data to serve as validation data.

Data

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	1	1/1/21	35742	2037	81632	73995	69537	66575	12942	38147	31162	24314	69892	54221	30794	35293	48941
2	2	1/2/21	98712	37528	22132	71391	12643	47527	55961	49727	44608	23933	4260	38176	9459	98844	79598
3	3	1/3/21	298	84628	13063	32466	10859	61628	47164	10881	91304	75963	46610	64651	59522	67120	7880
4	4	1/4/21	66718	2686	29648	95114	49792	64863	9915	88936	82481	8225	10800	2849	66393	17996	33973
5	5	1/5/21	34877	10849	44173	79394	22776	7874	75721	24976	88964	33645	243	51420	47652	27712	39293
6	6	1/6/21	65356	25589	66706	37956	32540	43796	55573	12420	9224	89521	64220	62939	59529	6090	58759
7	7	1/7/21	81348	79594	47132	86533	36233	86595	54395	73820	10396	79896	12473	31989	85234	74551	32718
8	8	1/8/21	7968	75483	33293	6738	61770	27841	99614	93046	4459	49663	18998	7236	67479	67199	9646
9	9	1/9/21	85286	79081	88736	54988	63965	12959	20512	39961	64863	8065	42727	41304	85387	66196	87882
10	10	1/10/21	33165	57891	31705	41957	75					49569	29879	58886	13775	71775	30596
11	11	1/11/21	38289	48262	46437	46440						59193	38734	9189	38842	14576	42271
12	12	1/12/21	21589	48215	11018	59325						46552	45231	1206	24653	354	85568
13	13	1/13/21	75083	76384	47792	35735	81					94994	27089	18123	21233	66472	96319
14	14	1/14/21	69279	10625	69330	26535	62881	81447	38317	30152	82017	73894	8360	58821	29631	71868	75294
15	15	1/15/21	33388	41088	29128	45978	7185	68372	3787	78084	64272	79576	16873	99670	95257	48355	23128
16	16	1/16/21	51327	72665	41877	29874	53632	4395	50578	22433	31961	64796	41386	24382	5223	23876	77019
17	17	1/17/21	45668	76166	36663	18759	63893	14668	18849	44225	12378	78888	30191	28171	1513	15218	94748
18	18	1/18/21	79241	88966	31646	21994	67473	66420	96893	86681	29336	43061	81468	3162	67199	66237	73786
19	19	1/19/21	7888	3614	42525	52437	58577	58177	9812	26568	56168	75791	75549	63674	38863	63708	9962
20	20	1/20/21	49227	54371	71020	71120	62851	81772	30591	25243	88131	64473	25274	15341	67595	48827	32322
21	21	1/21/21	72668	26033	74941	13516	74612	23386	45162	83440	45717	18196	83997	1733	94969	45881	21748
22	22	1/22/21	47826	91605	76296	34608	26362	46230	30915	21996	44878	12468	61052	35410	10795	26429	15671
23	23	1/23/21	39925	83935	48153	34397	63881	92011	15426	99957	86808	22891	99501	43874	77873	71237	98739
24	24	1/24/21	95319	32952	1439	57761	89529	59675	54149	85365	17656	31010	59372	44787	69533	25880	45395
25	25	1/25/21															
26	26	1/26/21	21993	39615	59252	9348	2443					79828	84474	46708	50451	45857	31738
27	27	1/27/21	62158	62351	61681	54528	2678					5097	65181	47526	6495	62715	48198
28	28	1/28/21	65179	74894	54244	65670	8863					17993	27870	794	21895	98668	28868
29	29	1/29/21															
30	30	1/30/21	63463	2291	88886	71963	11343					90813	61762	7487	37715	6863	32960
31	31	1/31/21	7297	17944	86917	69105	83057					59727	68414	16894	94899	61677	91243
32	32	1/32/21	20872	10522	96462	96887	60596					39339	54677	68782	63398	70516	13627
33	33	1/33/21															

Training Data - 80%

Validation Data - 10%

Testing Data - 10%

As explained in the course, it is ideal to train the data on the training data set, and evaluate the model on the validation set. We make improvements, and then retrain, and run the model against the validation data again. Continue doing this until you reach the desired level of accuracy as measured by the loss function. Then, make one final evaluation using the test data set. Since we only use the test data once, we avoid the issue of our model adapting to the test data, so we can trust the accuracy of our model analysis.

Data

		features (x)															label (y)	
		D	C	S	E	F	G											
1	1/1/21	35742	3357	81632	73995	69537												
2	1/2/21	98752	37528	22132	71391	52643	47527	50561	46727	44058	22053	4200	38976	5453	90544		75056	
3	1/3/21	298	84628	13063	35466	10859	61628	47164	10881	91304	75963	46610	64651	59522	67125		7896	
4	1/4/21	66718	2686	29648	95114	49792	64863	9915	88936	82481	8225	10800	2849	66333	11996		33973	
5	1/5/21	34677	10849	44173	79394	22776	7874	75721	24976	88954	33645	245	51420	47652	27712		39261	
6	1/6/21	65526	25559	66705	37955	35540	43796	55573	15420	9224	69521	54220	62939	59529	6090		58759	
7	1/7/21	81148	79594	47132	86533	36233	86595	54395	73820	15396	79936	52473	31989	65234	74561		32716	
8	1/8/21	7998	75483	33263	6738	61770	27841	39614	93046	4459	69663	18986	7236	67479	67119		9666	
9	1/9/21	85286	79081	80730	54938	63965	12969	20512	39661	64863	8065	42727	41304	85367	66196		87882	
10	1/10/21	33165	57891	31705	41957	75				49569	29879	58885	13775	71735	30596			
11	1/11/21	30289	40502	46437	46401	15				55193	38734	9189	30442	14576	42271			
12	1/12/21	51589	40215	11018	59325	15				46552	45231	1206	24553	354	80584			
13	1/13/21	75083	76384	47792	35735	61				94994	27089	19123	21233	66472	96319			
14	1/14/21	69279	10625	69330	26535	62051	81447	38317	30152	82017	73894	8360	58821	29631	71858		75294	
15	1/15/21	33388	41088	29126	45978	7185	68372	3787	78084	64272	79576	16873	99670	69527	40355		23128	
16	1/16/21	51327	72665	41877	29874	53632	4395	50578	22433	31951	64796	41386	24382	5223	23276		75019	
17	1/17/21	45660	76166	36653	18759	63893	14660	18849	44225	12378	78888	30111	29171	1513	15218		58148	
18	1/18/21	79241	88966	31646	21994	67473	66520	96893	86681	29336	43061	81408	3162	67139	66217		73195	
19	1/19/21	7888	3514	42525	52437	55577	59177	9812	26568	56168	75791	75549	63674	30863	53708		9962	
20	1/20/21	49227	54371	71020	71120	62051	81772	30591	20243	89131	64473	25274	15341	67595	48637		32322	
21	1/21/21	72668	26033	74941	13516	74612	23388	45162	83480	45717	18196	63997	1733	54959	45881		21748	
22	1/22/21	47826	91605	76296	34608	26362	46230	30915	21996	44878	12468	61052	35610	10795	26429		15671	
23	1/23/21	39925	83935	48153	34307	63081	92911	15426	99957	86808	22891	99501	43974	77973	71237		98739	
24	1/24/21	95319	32952	1439	57761	89529	59675	54149	85365	17656	31010	59752	44787	69533	25880		45395	
25																		
26	1/26/21	21993	39615	59252	5948	244				79828	84474	48708	50451	45857	31738			
27	1/27/21	62158	62351	51081	54528	2678				5097	65181	47526	6495	62715	40196			
28	1/28/21	65179	74894	54244	65670	8863				17993	27670	794	21895	98658	28885			
29																		
30	1/30/21	63683	2210	88886	71963	11343				12	90813	61762	7407	37715	6863		32960	
31	1/31/21	7297	17544	86917	69105	63057				15	59727	68414	16894	54499	61877		91243	
32	1/32/21	25872	10522	99402	96987	60196				16	38139	94677	65782	63308	79516		13627	

When thinking about our data in terms of input variables (x) and output label (y), we have n number of observations or data points. So in this data set, which to make things simple, looks like a small spreadsheet, each row is an observation, or a data point. All of the columns, except the last column represent features, so basically, one feature per column. If this data is for facial recognition, one column might be the width of the right eye, and one column might be the distance between the eyes. The last column is the label column - a person's name.

We give the model the training data, which includes the features and the label for this subset of the observations. When evaluating the model, first with the validation data, we **don't give the model the label column**. We give the model the features for each observation, and ask the model to predict the label. Then, we evaluate the model by comparing the label that the model gives to

the true label. We can make changes to the model, retrain it, and evaluate it again with the validation data, hopefully improving with each iteration. Finally, we test the model with the test data. Again, we just give the model the features for each observation, and ask the model to predict the label. We determine the accuracy of the model by comparing the model's predicted label with the true label.

Loss Function

Classification:

Zero-one - how many mistakes were made?

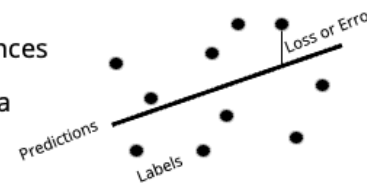
True Positives	False Negatives	Sensitivity
False Positives	True Negatives	Specificity

Example: Confusion Matrix

Regression:

Squared loss - penalizes larger differences

Absolute - does not penalize noisy data



As talked about in the course, we measure the accuracy of our model with a loss function. As we make improvements to the model, the goal is to increase accuracy - minimize mistakes, which will minimize the loss function. This course talks about the different loss functions - zero-one, squared, and absolute losses.

Zero-one is the simplest loss function. It counts how many mistakes were made. This can be used for classification problems, when you want to know how many incorrect labels were given versus the total number of labels.

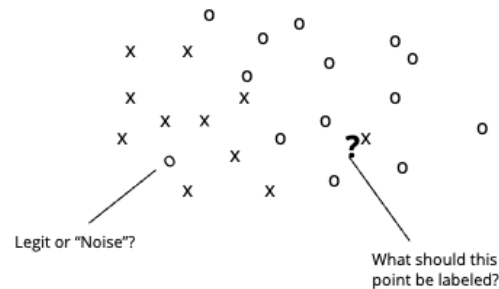
Squared loss function is often used in regression problems.

Squaring removes negative signs, and also gives more weight to larger differences. So, if we have noisy data, the squared loss function may not work well because it will give weight to data that is way off. In this case we may want to use absolute loss function, which is the absolute differences between our predicted and actual values. So, noisy data is not penalized like it is with the squared loss function.

k-Nearest Neighbors Algorithm

Parameter (a.k.a. hyperparameter) - Select optimum k

- k = the number of neighbors to consider



k-Nearest Neighbors takes labeled data, with the labels indicating a particular class, and the algorithm will predict the label for a new data point, test data, based on the labels of nearby training points. We can choose how many neighbors are considered. If we set k to 3, the three closest neighbors are considered. The three labels are taken into account, and majority rules. If there is a tie, the label chosen will be that of the nearest neighbor.

By increasing the number of nearest neighbors, we reduce the complexity of the decision boundary, thereby eliminating noise (misclassified data points). However, if we increase the number of nearest neighbors too high, groups of data points will then be misclassified in order to produce a smoother decision boundary. So, it is a balance to find the best k value to maximize accuracy.

Since we're relying on similarity between the new data point and the nearby training data points, the way we are determining which data are the closest, or most similar matters. The distance function

should return a distance that measures the similarity of the data in a meaningful way, such that data points that are measured as closer together are in fact more similar than data points measured as being farther apart.

k-Nearest Neighbors works best if you have data that fit into a few classes, rather than having data with so many features that very few data points are similar. As dimensionality increases (specifically, having a high number of uncorrelated dimensions), similarity among data points decreases.

Defining “Noise” in Data

<https://medium.com/mlearning-ai/dont-make-me-come-over-there-440f7eece4f3>

NumPy

Module 3 - NumPy and Jupyter Notebooks

Codio environment

- Introduction to Numpy
- Practice Matrix Multiplication
- Additional NumPy Exercises

The Codio environment is built into the course, and you'll see it in modules 3 and 4. In module 3, there are a series of exercises to help you learn NumPy. NumPy, is a Python library for scientific computing. As the course explains, using NumPy will allow for much more efficient code than writing loops in Python.

In the codio environment, you can walk through several exercises that will teach you some important NumPy functions. The Introduction to NumPy section in the course steps through arrays, attributes of arrays, vectors, and matrixes. Today we will look at reshape, some matrix operations, as well as indexing and slicing, and I'll give you a few extra examples that you can work through that might help you understand the NumPy functions better. I encourage you to take these examples and work through them on your own in codio. It's important to understand the NumPy functions and understand how they work. You'll likely need to use some of these functions in the final project, so it's important to gain an understanding of them in the codio practice environment.

Vectors in NumPy

Collapse

⏪ ⏩ ⏴ ⏵

Vectors in NumPy

You can use arrays to represent a vector, which is essentially a 1-D array. There are three ways to represent a vector using NumPy. You'll first start by using `np.array()` to create a 1-D array. Here's an example:

```
npvec = np.array([1,2,3])
```

This array, which we will refer to as a NumPy vector, has the shape of `(3,)`, meaning the array is indexed by a single index from 0 to 2.

You can then use `.reshape()` to transform a NumPy vector into a column or a row vector. For example, you could reshape our NumPy vector `npvec` into a column vector with the following line:

```
colvec = npvec.reshape((3,1))
```

Instructions

1. Create a NumPy vector `v1` that contains `[[3,4,5]]` using the `np.array()` function.
2. `print()` the vector `v1` to confirm you've correctly created the vector.
3. `print()` the vector's shape `v1.shape`. You should find that its shape is `(3,)`
4. Using the function `.reshape()`, reshape the `v1` NumPy vector into a column vector `v2` by changing the shape to `(3,1)`
5. `print()` `v2` to confirm your data is stored in a column vector.
6. Reshape the `v1` NumPy vector into a column vector `v3` by changing the shape to `(1,3)`
7. `print()` `v3` to confirm your data is stored in a row vector.

Terminal

```
In [7]: npvec = np.array([1,2,3])
In [8]: colvec = npvec.reshape((3,1))
In [9]: v1 = np.array([3,4,5])
In [10]: print(v1)
[3 4 5]
In [11]: print(npvec)
[1 2 3]
In [12]: print(colvec)
[[1]
 [2]
 [3]]
In [13]: v1.shape
Out[13]: (3,)
In [14]: v2 = v1.reshape((3,1))
In [15]: print(v2)
[[3]
 [4]
 [5]]
In [16]: v3 = v1.reshape((1,3))
In [17]: print(v3)
[[3 4 5]]
In [18]:
```

In codio you will have these instructions for reshaping a vector, which is good, but this is a pretty simple example.

Linear Algebra Concepts

Dot product - the sum of the products of the components of two vectors

Dot product is sometimes called the scalar product or the inner product

Dot product formula for vectors \mathbf{x} and \mathbf{y} :

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2$$

Next, let's briefly go over some linear algebra concepts that will be needed for module 4.

Dot product is the sum of the products of the components of two vectors. The dot product is the same thing as inner product. Be sure to go all the way through the NumPy codio exercises, and understand how to do dot product. You will need to be able to do this in the final project.

Dot Product - examples from Linear Algebra course

$$\mathbf{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

$$\mathbf{x} \cdot \mathbf{y} = 1 \cdot 3 + 2 \cdot 4 = 11$$

$$\mathbf{x} = \begin{pmatrix} -2 \\ 3 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}$$

$$\mathbf{x} \cdot \mathbf{y} = (-2) \cdot 4 + 3 \cdot 1 = -8 + 3 = -5$$

Here is are two examples of dot product. Both examples are taken from the Linear Algebra course that you all have access to.

Note that the dot product of \mathbf{x} with \mathbf{y} is the same as the dot product of \mathbf{y} with \mathbf{x} . So, order does not matter.

```
x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])
```

```
x1 = np.array([[1,2,3,4], [4,5,6,7], [7,8,9,10]])
```

```
In [37]: x1
```

```
Out[37]:  
array([[ 1,  2,  3,  4],  
       [ 4,  5,  6,  7],  
       [ 7,  8,  9, 10]])
```

```
In [38]: x2
```

```
Out[38]:  
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

```
x1.shape  
(3, 4)
```

```
x2.shape  
(3, 3)
```

Dot product = "inner" --
the inner shape values must
match for it to work

```
x1 @ x2
```

```
x2 @ x1
```

?

Let's look at a more complex example and make sure we understand what's going on here.


```
In [22]: x1.shape
Out[22]: (3, 4)

In [23]: x2.shape
Out[23]: (3, 3)

In [24]: x1 @ x2
-----
ValueError                                Traceback (most recent call last)
<ipython-input-24-f5addec7e971> in <module>()
----> 1 x1 @ x2

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc
signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 4)

In [25]: x2 @ x1
Out[25]:
array([[ 30,  36,  42,  48],
       [ 66,  81,  96, 111],
       [102, 126, 150, 174]])
```

```
In [37]: x1
```

```
Out[37]:
```

```
array([[1, 2, 3, 4],  
       [4, 5, 6, 7],  
       [7, 8, 9, 10]])
```

```
In [38]: x2
```

```
Out[38]:
```

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

$$1x1 + 2x4 + 3x7 = 1 + 8 + 21 = 30$$

```
In [37]: x1
```

```
Out[37]:
```

```
array([[1, 2, 3, 4],  
       [4, 5, 6, 7],  
       [7, 8, 9, 10]])
```

```
In [38]: x2
```

```
Out[38]:
```

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

$$1x1 + 2x4 + 3x7 = 1 + 8 + 21 = 30$$

$$1x2 + 2x5 + 3x8 = 2 + 10 + 24 = 36$$

```
In [37]: x1
```

```
Out[37]:
```

```
array([[1, 2, 3, 4],  
       [4, 5, 6, 7],  
       [7, 8, 9, 10]])
```

```
In [38]: x2
```

```
Out[38]:
```

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

$$1 \times 1 + 2 \times 4 + 3 \times 7 = 1 + 8 + 21 = 30$$

$$1 \times 2 + 2 \times 5 + 3 \times 8 = 2 + 10 + 24 = 36$$

$$1 \times 3 + 2 \times 6 + 3 \times 9 = 3 + 12 + 27 = 42$$

```
In [37]: x1
Out[37]:
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [7, 8, 9, 10]])
```

```
In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

$$1x1 + 2x4 + 3x7 = 1 + 8 + 21 = 30$$

$$1x2 + 2x5 + 3x8 = 2 + 10 + 24 = 36$$

$$1x3 + 2x6 + 3x9 = 3 + 12 + 27 = 42$$

$$1x4 + 2x7 + 3x10 = 4 + 14 + 30 = 48$$

```
In [37]: x1
Out[37]:
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [7, 8, 9, 10]])
```

```
In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

$$1 \times 1 + 2 \times 4 + 3 \times 7 = 1 + 8 + 21 = 30$$

$$1 \times 2 + 2 \times 5 + 3 \times 8 = 2 + 10 + 24 = 36$$

$$1 \times 3 + 2 \times 6 + 3 \times 9 = 3 + 12 + 27 = 42$$

$$1 \times 4 + 2 \times 7 + 3 \times 10 = 4 + 14 + 30 = 48$$

```
In [42]: x2 @ x1
Out[42]:
array([[30, 36, 42, 48],
       [66, 81, 96, 111],
       [102, 126, 150, 174]])
```

```
In [37]: x1
```

```
Out[37]:
```

```
array([[1, 2, 3, 4],  
       [4, 5, 6, 7],  
       [7, 8, 9, 10]])
```

```
In [38]: x2
```

```
Out[38]:
```

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

$$4 \times 1 + 5 \times 4 + 6 \times 7 = 4 + 20 + 42 = 66$$

```
In [37]: x1
```

```
Out[37]:
```

```
array([[1, 2, 3, 4],  
       [4, 5, 6, 7],  
       [7, 8, 9, 10]])
```

```
In [38]: x2
```

```
Out[38]:
```

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

$$4 \times 1 + 5 \times 4 + 6 \times 7 = 4 + 20 + 42 = 66$$

```
In [42]: x2 @ x1
```

```
Out[42]:
```

```
array([[30, 36, 42, 48],  
       [66, 81, 96, 111],  
       [102, 126, 150, 174]])
```



```
In [37]: x1
Out[37]:
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [7, 8, 9, 10]])

In [38]: x2
Out[38]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [42]: x2 @ x1
Out[42]:
array([[30, 36, 42, 48],
       [66, 81, 96, 111],
       [102, 126, 150, 174]])
```

<https://jasondeden.medium.com/matrix-multiplication-e2cf007d0755>

Inner Product vs. Product, etc.

Most matrix math operations require shapes to match, not just inner values, and produce results of that same shape as well

* + -

Transpose

```
In [45]: x1.T @ x2
Out[45]:
array([[ 66,  78,  90],
       [ 78,  93, 108],
       [ 90, 108, 126],
       [102, 123, 144]])
```

Transpose

```
In [45]: x1.T @ x2
Out[45]:
array([[ 66,  78,  90],
       [ 78,  93, 108],
       [ 90, 108, 126],
       [102, 123, 144]])
```

```
In [48]: x1
Out[48]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])
```

```
In [49]: x1.T
Out[49]:
array([[ 1,  4,  7],
       [ 2,  5,  8],
       [ 3,  6,  9],
       [ 4,  7, 10]])
```

```
In [50]: x1.T.shape
Out[50]: (4, 3)
```

Reshape

Note: reshape must be the same size as the original

```
In [2]: v2 = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
In [3]: print(v2)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
In [4]: v2.shape
Out[4]: (3, 4)
In [5]: v3 = v2.reshape(4,3)
In [6]: print(v3)
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
In [7]: v4 = v2.reshape(6,2)
In [8]: print(v4)
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
In [9]: v5 = v2.reshape(1,10)
ValueError                                Traceback (most recent call last)
~\python-input-9-45fe76ca9a20 in <module>:
----> 1 v5 = v2.reshape(1,10)
ValueError: cannot reshape array of size 12 into shape (1,10)
```

Here is another example that's a little more complex that might help illustrate what reshape can and can't do. So if we have a matrix with 3 rows and 4 columns, we can tell NumPy that we want to reshape this into 4 rows and 3 columns. We could also say that we want six rows and 2 columns. But we can't ask for a shape that is different from the number of elements that we have. For example, we can't say we want to reshape this into 1 row and 10 columns because our array has 12 elements not 10. We would get a similar error if we said reshape into 2 rows and 5 columns, or 3 rows and 2 columns. We have to choose a shape that will exactly fit the number of elements that we have.

Reshape

```
In [56]: x1
Out[56]: array([[ 1,  2,  3,  4],
                [ 4,  5,  6,  7],
                [ 7,  8,  9, 10]])

In [57]: x1.T
Out[57]: array([[ 1,  4,  7],
                [ 2,  5,  8],
                [ 3,  6,  9],
                [ 4,  7, 10]])

In [58]: x1.reshape(4,3)
Out[58]: array([[ 1,  2,  3],
                [ 4,  4,  5],
                [ 6,  7,  7],
                [ 8,  9, 10]])
```

Reshape

```
In [56]: x1
Out[56]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])
```

```
In [57]: x1.T
Out[57]:
array([[ 1,  4,  7],
       [ 2,  5,  8],
       [ 3,  6,  9],
       [ 4,  7, 10]])
```

```
In [58]: x1.reshape(4,3)
Out[58]:
array([[ 1,  2,  3],
       [ 4,  4,  5],
       [ 6,  7,  7],
       [ 8,  9, 10]])
```

```
In [59]: x1.T @ x2
Out[59]:
array([[ 66,  78,  90],
       [ 78,  93, 108],
       [ 90, 108, 126],
       [102, 123, 144]])
```

```
In [60]: x1.reshape(4,3) @ x2
Out[60]:
array([[ 30,  36,  42],
       [ 55,  68,  81],
       [ 83, 103, 123],
       [114, 141, 168]])
```

```
Other Matrix Op...
Collapse
Other Matrix Operations
Largest element in X
np.amax(X)
Largest elements along the first axis
np.amax(X, axis = 0)
Smallest element in X
np.min(X)
Smallest elements along the second axis
np.min(X, axis = 1)
Index of the smallest element in X
np.argmin(X)
Indices of the largest elements along the first axis
np.argmax(X, axis = 0)
Sum of all elements in X
np.sum(X)
Sum of elements along the first axis
np.sum(X, axis = 0)
Instructions
1. print() matrix Y to see its content.
2. Find the largest element in Y.
3. Find the smallest elements along the first axis of Y.
4. Find the index of the smallest element in Y.
5. Find the indices of the largest elements along the second axis of Y.
6. Find the sum of elements in Y along the second axis.

Terminal
In [85]: print(Y)
[[5 6]
 [7 8]]

In [86]: np.amax(Y)
Out[86]: 8

In [87]: np.amax(Y, axis=0)
Out[87]: array([7, 8])

In [88]: np.min(Y)
Out[88]: 5

In [89]: np.argmax(Y, axis=1)
Out[89]: array([1, 1])

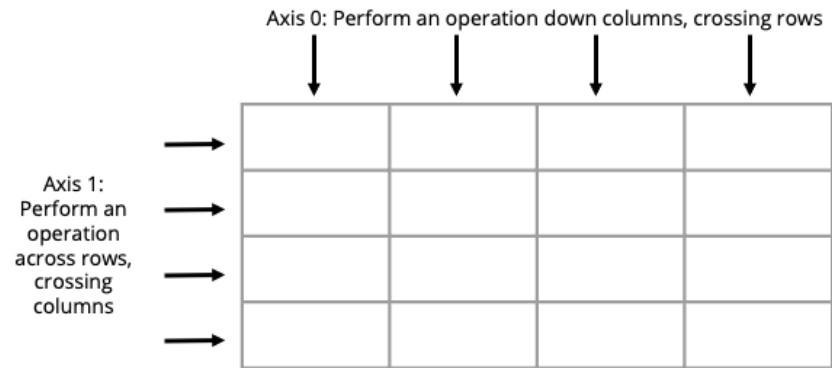
In [90]: np.sum(Y, axis=1)
Out[90]: array([11, 15])
```

Let's look at the Other Matrix Operations section of the Introduction to NumPy in codio.

The code on the right follows the steps on the left. I step through some of this, but I won't talk through every example. I will make these slides available so you can go through them later if you want.

Here we can see that `np.amax` gives us the largest value of the matrix, which is 8, and `np.min` gives us the smallest value, which is 5. `np.amax` along the axis 0 gives us 7 and 8, which means we're wanting the max value for each column. `np.argmax` gives us the indices of the largest value, 8, which is `[1, 1]`, meaning the 8 is in the row 1, column 1, which remember is the second row and second column. Finally, `np.sum` gives us the sum of the first row, 11, and the sum of the second row, 15.

NumPy Axes



In a NumPy array, axis 0 is the “first” axis. It is the axis that runs down the rows. Axis 1 is the “second” axis, and it runs across the columns.

NumPy Axes, NumPy Sum

<https://www.sharpsightlabs.com/blog/numpy-axes-explained/>

axis = 0, np.sum()



axis = 1, np.sum()



If we set the axis to 0, when we sum, we're collapsing/crossing the rows, and summing each column. If we set the axis to 1, when we sum we're collapsing/crossing the columns, and summing across each row.

So, check out that link and read through the explanation, and I think numpy axes will make more sense.

```

In [103]: Y2 = np.array([[9,5,4,3],[7,8,6,2]])

In [104]: print(Y2)
[[9 5 4 3]
 [7 8 6 2]]

In [105]: np.amax(Y2)
Out[105]: 9

In [106]: np.amax(Y2, axis=0)
Out[106]: array([9, 8, 6, 3])

In [107]: np.amax(Y2, axis=1)
Out[107]: array([9, 8])

In [108]: np.amin(Y2)
Out[108]: 2

In [109]: np.amin(Y2, axis=0)
Out[109]: array([7, 5, 4, 2])

In [110]: np.amin(Y2, axis=1)
Out[110]: array([3, 2])

```

Additional Examples

```

In [111]: np.argmax(Y2, axis=0)
Out[111]: array([0, 1, 1, 0])

In [112]: np.argmax(Y2, axis=1)
Out[112]: array([0, 1])

In [113]: np.argmin(Y2, axis=0)
Out[113]: array([1, 0, 0, 1])

In [114]: np.argmin(Y2, axis=1)
Out[114]: array([3, 3])

In [115]: np.sum(Y2)
Out[115]: 44

In [116]: np.sum(Y2, axis=0)
Out[116]: array([16, 13, 10, 5])

In [117]: np.sum(Y2, axis=1)

```

Here is another example that is not in the instructions. I encourage you to try this on your own in codio.

Create a matrix Y2 like I did here. So, we have Y2 with elements 9, 5, 4, 3 in the first row, which is row zero, and elements 7, 8, 6, and 2 in the second row, which is row one. So, as we would expect, 9 is returned as the maximum value for the matrix, and 2 is the minimum. I think with this larger matrix, it's easier to understand the other functions. Here we see that np.amax for axis=0 returns the maximum value for each column. Here we have 4 columns, which are columns 0 through 3. The max value for column zero is 9 from the top row, for column 1, it is 8 from the bottom row, for column 2, it is 6 from the bottom row, and for column 3, it is 3 from the top row. Next, np.amax for axis=1 returns the maximum value for each row, so 9 from row 0, and 8 from row 1. Since we have two rows, np.amin for axis 0 basically returns the opposite values, the minimum for each column, 7, 5, 4, 2. Np.amin for axis 1 returns the minimum for each row, 3 for row 0 and 2 for row 1.

So, `amax` and `amin` return values, and `argmax` and `argmin` returns indices. So, let's look at `np.argmax` along the 0 axis, and we get 0,1,1,0, which tells us for each column 0 through 3, which row contains the maximum value. So, for column 0, 9 is row 0, so that's the first zero returned. For column 1, 8 is in row 1, so that's the first 1 that's returned, for column 2, the 6 is in row 1, so that's the next 1 that was returned, and finally, for column 3, the 3 is in row 0. For `np.argmax` along the axis 1, we get back the number for the column that contains the max value for each row. So, for row 0, column 0 contains the max value, which is 9, and for row 1, column 1 contains the max value, which is 8. Same concept for `argmin`. For `np.argmin`, axis = 0, we're getting back the number for each column that contains the minimum value for each row. So, for column 0, row 1 contains the minimum value, 7. For column 1, row 0 contains the minimum value 5, for column 2, row 0 contains the minimum value 4, and finally for column 3, row 1 contains the minimum value 2. And for `np.argmin`, we get back 3, 3 because for row 0, the minimum value, which is 3, is in column 3, and for row 1, the minimum value, which is 2, is also in column 3. And, `np.sum` gives us 44 which is the sum of the entire matrix. `Np. sum` for axis=0 gives us the sum of each row, and `np.sum` for axis=1 gives us the sum of each column, which this screenshot is missing, but should be 21 and 23.

Self-Check #2

Collapse

Self-Check #2

Instructions

1. Find the maximum elements in `A` along the first axis (axis = 0) and add it to the sum of elements in `B` along the first axis.
2. Check your results. Your output should be:

```
array([12, 15, 18, 21])
```

Terminal

```
In [126]: print(A)
[[ 1  2  3  4]
 [ 7  8  9 10]]

In [127]: print(B)
[[4 5 6 7]
 [1 2 3 4]]

In [128]: maxA = np.amax(A, axis=0)

In [129]: print(maxA)
[ 7  8  9 10]

In [130]: sumB = np.sum(B, axis=0)

In [131]: print(sumB)
[ 5  7  9 11]

In [132]: self_check_2 = maxA + sumB

In [133]: print(self_check_2)
[12 15 18 21]
```

As part of the NumPy exercises, there are self checks. These are not for a grade, but all of these exercises are very important to do. You'll need to understand these NumPy functions in order to do the final project. This self check has to do with finding the maximum elements in a matrix along the first axis (axis=0), and adding it to the sum of the elements in a second matrix along the first axis.

Additional Examples

```
In [134]: C = np.array([[4,55,6,7],[11,2,33,4]])
In [135]: print(C)
[[ 4 55  6  7]
 [11  2 33  4]]
In [136]: print(Y2)
[[9 5 4 3]
 [7 8 6 2]]
In [137]: maxC = np.amax(C, axis=0)
In [138]: print(maxC)
[11 55 33  7]
In [139]: sumY2 = np.sum(Y2, axis=0)
In [140]: print(sumY2)
[16 13 10  5]
In [141]: self_check = maxC + sumY2
In [142]: print(self_check)
[27 68 43 12]
In [143]: []
```

Let's do an additional example of np.sum. Here we have two matrices, matrix C and also matrix Y2. So, just like in the previous example, we're going to get the max elements from matrix C along the first axis, and then we're going to add them to the sum of the elements from the first axis in matrix Y2. So, the max elements from matrix C along axis zero are 11, 55, 33, and 7. The sum of the elements in matrix Y2 along the first axis (axis=0) are 16, 13, 10, and 5. So now, let's add these together and we get 27, 68, 43, and 12.

Numpy Arange

`np.arange(2,12,2)` will give us the following array:

```
[2 4 6 8 10]
```

numpy.org - documentation for NumPy functions

`np.arange()`

- Start - start of the interval
- Stop - end of the interval, does not include this value
- Step - spacing between values

So, why do we get 2, 4, 6, 8, and 10 with `np.arange(2,12,2)`? And, how can we better understand NumPy functions?

So for any NumPy function that we don't understand, we can look at NumPy.org at the documentation.

The documentation for `arange` explains that the first number indicates the starting point, the second number is the stopping point, and the third number indicates the step, which is the spacing between values. It's important to note that the stop indicates the end of the interval, but does not include this value. So we're creating an array that starts with 2's, increases by 2's, and goes up to but not including 12. If we don't give NumPy the first number to indicate the starting point, it will default to zero. If we don't give NumPy the second number to indicate the end, it will default to the length of the array, and if we don't give NumPy the third number for the step, it will default to 1.

(ndarray object gets created - an n-dimensional array - collection of items of all

the same type, zero-based index)

Collapse

< > ?

Indexing and Slicing

NumPy arrays can be indexed and sliced, just like Python's list. Let's explore indexing and slicing on 1-D and multidimensional arrays.

One-Dimensional Arrays

Instructions

- Create the following 1-D array:
`x1 = np.arange(2,12,2)`
- `print()` this new array.
- Index a single element, such as `x1[3]`.
- Slice a portion of the array, such as `x1[1:3]`.
- Slice the last two numbers of the array using `x1[-2:]`.

Multidimensional Arrays

Instructions

- Create the following 2-D array:
`x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])`
- `print()` this new array.
- Find a full slice using `x2[:,:]`.
- Index an element at the first row, third column using `x2[0,2]`.
- Slice on both axes using `x2[1:, :2]`.
- Index by row, selecting the first row using `x2[0]`.

```
[In (4): x1 = np.arange(2,12,2)
[In (5): print(x1)
[ 2  4  6  8 10]
[In (6): i4 = x1[-1]
[In (7): print(i4)
8
[In (8): slice = x1[1:]
...
[In (9): print(slice)
[4 6]
[In (10): sliceend = x1[-2:]
[In (11): print(sliceend)
[ 8 10]
[In (12): x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])
...
[In (13): print(x2)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[In (14): fullslice = x2[:,:]
[In (15): print(fullslice)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[In (16): rowcol3 = x2[0,2]
[In (17): rowcol3
Out[17]: 3]
```

Terminal

```
[In (35): slicebothaxes = x2[:, :2]
[In (36): print(slicebothaxes)
[[4 5]
 [7 8]]
[In (37): for row in x2: print(row)
[1 2 3]
[4 5 6]
[7 8 9]
[In (38): ]
```

The next part of the Introduction to NumPy tutorial goes through Indexing and Slicing, so let's take a look at that more closely.

So, first it's asking us to create a 1-dimensional array using `np.arange(2,12,2)`. And this produces the array 2, 4, 6, 8, 10.

Indexing and Slicing

Multidimensional Arrays

Instructions

1. Create the following 3-D array:
`x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])`.
2. `print()` this new array.
3. Find a full slice using `x2[:,:]` .
4. Index an element at the first row, third column using `x2[0,2]`.
5. Slice on both axes using `x2[1:,:2]`.
6. Index by row, selecting the first row using `x2[0]`.
7. Index by column, selecting the first column using `x2[:,0]`.
8. Iterate through each row in a matrix:
`for row in x2: print(row)`.

```
In [12]: x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])
...:

In [13]: print(x2)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

In [14]: fullslice = x2[:,:]

In [15]: print(fullslice)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

In [16]: row1col3 = x2[0,2]

In [17]: row1col3
Out[17]: 3
```

Let's look at indexing and slicing examples. This slide steps through the exercises in the Indexing and Slicing section for Multidimensional Arrays. First, we create the array `x2`, and then we see that we can create a full slice, which is basically the original array, using open bracket, colon, colon, close bracket. We can return the element at the first row, third column with open bracket, zero comma two, close bracket because the zero refers to the row zero, and the 2 refers to the column 2, which is the third column.

Indexing and Slicing

Multidimensional Arrays

Instructions

1. Create the following 3-D array:
`x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])`.
2. `print()` this new array.
3. Find a full slice using `x2[:,:]` .
4. Index an element at the first row, third column using `x2[0,2]`.
5. Slice on both axes using `x2[1:,:2]`.
6. Index by row, selecting the first row using `x2[0]`.
7. Index by column, selecting the first column using `x2[:,0]`.
8. Iterate through each row in a matrix:
`for row in x2: print(row)`.

```
In [13]: print(x2)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
Terminal

In [35]: slicebothaxes = x2[1:,:2]

In [36]: print(slicebothaxes)
[[4 5]
 [7 8]]

In [37]: for row in x2: print(row)
[1 2 3]
[4 5 6]
[7 8 9]

In [38]:
```

In this example, we can slice on both axes by using open bracket, one colon comma colon two. This gives us the slice that is row 1 on, and all columns up to but not including column 2, so we skip row 0, and get rows 1 and 2, and we get column 0 and 1. This example can be a little confusing, so make sure you understand what's going on here.

Although it's not shown in the screenshot here, if you try the example on number 6, open bracket colon comma zero close bracket, it will return the first row, 1, 2, and 3. Example 7 shows us how to select the first column with open bracket colon comma zero close bracket. Finally, example 8 will print each row, so it will basically return the whole array. So, if we don't include a comma, we're providing row operations only. If we do include a comma, everything before the comma is row instructions, and everything after the comma are column instructions. Let's look at some more examples.

Array Slicing Under the Covers

We slice in 1-dimension by providing the following information:

`x1[start:end]`

`x1[start:end:step]`

`x1[:]`

Note: values can be negative, in which case it starts from the end and counts backwards

If we don't specify start, will assume 0

If we don't specify end, will assume the length of the array in that direction

If we don't specify the step (interval), will assume 1

`x1[:]` is the same as `x1[0:len(x1):1]` -- or simply `x1`

We can give NumPy slicing instructions as shown. We can provide some of these parameters, or none. The basic slice syntax is start end step. We can specify the start, or the end, or the step, or none. If we just put the name of the array, then open bracket, colon, colon, close bracket, we get the whole array back. Why? Because, if we don't tell NumPy where to start, it will assume 0. If we don't tell NumPy where to end, it will assume we want the whole thing, and if we don't tell NumPy an step value, which is the interval, NumPy will put in 1 because it assumes we don't want to skip values.

The step is not necessary. If we tell NumPy `x1` open bracket colon close bracket, so that we only have one colon, NumPy is going to look for the start and end values, and assume the step is 1

So this syntax will give us 1-dimensional slicing, basically giving us row-level operations.

2D Array Slicing

2-dimensional slicing, rows and columns:

`x1[start:end:step,start:end:step]`

`x1[start:end,start:end]`

Note: Useful technique if you want to create different data by separating out the last column

`X =[:, :-1]` gives us all rows and all columns except the last column

`y =[:, -1]` gives us all rows and only the last column

Note: in 2D slicing, you must at a minimum include a colon for the first dimension

So in 2-dimensions, to define a slice with rows and columns, we use a comma.

Remember, it's not necessary to define any of the parameters, and often we leave off the step because we don't need an interval.

If we have a dataset and we have several columns of features, and the last column is our labels, we might want to split that data into X and y, features and labels.

So if we tell NumPy open bracket, colon, comma, colon, minus 1, close bracket, we're not giving a start or stop value for the rows, so we want all rows, and we're not giving a start value for columns, but our end value is minus 1, meaning give us all columns except the last one.

If we tell NumPy open bracket, colon, comma, minus 1, we're asking for all rows, and this time we're only providing the starting value for columns, which we're saying minus 1, so give us only the last column.

In the final project, the x and y data are already done for you, but I thought this example would help illustrate 2-dimensional slicing.

It is possible to index and slice in 3-dimensions, but that's beyond the scope of this class.

You can visit [NumPy.org](https://numpy.org) and look at the documentation for more information and examples on Indexing and Slicing.

Additional Examples

```
In [39]: print(x2)
[[1 2 3]
 [4 5 6]
 [7 8 9]]

In [40]: slice1 = x2[:, :2]

In [41]: print(slice1)
[[1 2]
 [4 5]
 [7 8]]

In [42]: slice2 = x2[::2]

In [43]: print(slice2)
[[1 2 3]
 [7 8 9]]
```

```
In [44]: slice3 = x2[0, :2]

In [45]: print(slice3)
[[1 2]
 [4 5]
 [7 8]]

In [46]: slice4 = x2[1:2]

In [47]: print(slice4)
[[4 5 6]]

In [48]: slice5 = x2[::2, ::2]

In [49]: print(slice5)
[[1 3]
 [7 9]]
```

Let's look at these five examples to better understand the explanation from the previous slides.

These examples use the same array we saw earlier, x2.

So, slice 1, which is open bracket colon comma colon 2 will return the first two columns because its asking for all rows, and columns up through column 2, which is column zero and one. Slice 2 is an interesting example.

Slice 2, which is open bracket colon colon 2, will return row zero and row 2. Why does it return rows zero and row 2? Since we did not use a comma, this only operated at the row level. Since we didn't put in a value before the first colon, we are not specifying a starting point, so the default is zero. Since we didn't put in a value between the colons, we didn't specify a stopping point, so we will get the full length of the array, but we did put a 2 after the second colon, which tells the slicing operation the interval step is 2, which means, return every other row.

Now, let's look at slice 3, which includes a comma, so we're giving

instructions for rows and columns. Slice 3 is open bracket zero colon comma colon two. This is basically the same as slice 1, but we've specified all rows, and column up through column 2, so we get all rows of columns zero and one.

Next, slice 4 is open bracket one colon colon 2 close bracket, and it returns row 1 only, which is the second row. Here's another example that does not include a comma, so it's a row operation. We're saying start with row 1 and return every other row. Since there are three rows, we only get row 1, the second row.

Finally, slice 5 is a fun example that focuses on defining the step, which is the interval. You can see that it returns the four corners of the matrix x2. It does this because we're asking for every other row and every other column. So slice 5 is open bracket colon colon two comma colon colon two. So, this eliminates row 1 and column 1 (the second row and the second column), and returns 1, 3, 7, 9.

Self-Check #3

Collapse

Self-Check #3

Instructions

1. Return only the second and third column of matrix `A`.

2. Check your results. Your output should be:

```
array([[3, 4],  
       [9, 10]])
```

Well done! You've completed this exercise. Move on to the next course page by clicking the **Next** button in the lower-right corner of the course page.

Terminal

```
In [53]: A = np.array([[1,2,3,4], [7,8,9,10]])  
In [54]: print(A)  
[[ 1  2  3  4]  
 [ 7  8  9 10]]  
In [55]: selfcheck3 = A[0,:4]  
In [56]: print(selfcheck3)  
[[ 3  4]  
 [ 9 10]]  
In [57]: selfcheck3b = A[:,2:]  
In [58]: print(selfcheck3b)  
[[ 3  4]  
 [ 9 10]]  
In [59]: selfcheck3c = A[:,~2:]  
In [60]: print(selfcheck3c)  
[[ 3  4]  
 [ 9 10]]  
In [61]:
```

Next is self-check 3. Make sure you work through this and the other exercises on your own, and do additional examples until you understand the functions. This slide shows three different ways to solve this exercise.

Terminal

```
In [53]: A = np.array([[1,2,3,4], [7,8,9,10]])

In [54]: print(A)
[[ 1  2  3  4]
 [ 7  8  9 10]]

In [55]: selfcheck3 = A[0:,2:4]

In [56]: print(selfcheck3)
[[ 3  4]
 [ 9 10]]
```

(same as previous slide)

```
In [57]: selfcheck3b = A[:,2:]

In [58]: print(selfcheck3b)
[[ 3  4]
 [ 9 10]]

In [59]: selfcheck3c = A[:, -2:]

In [60]: print(selfcheck3c)
[[ 3  4]
 [ 9 10]]
```

This is the same as the previous slide - enlarged so we can see it better.

Additional Examples

```
Terminal

In [70]: print(A)
[[ 1  2  3  4]
 [ 7  8  9 10]]

In [71]: slice1 = A[:,3]

In [72]: print(slice1)
[[1 2 3 4]]

In [73]: slice2 = A[0,:3]

In [74]: print(slice2)
[[1 2 3]
 [7 8 9]]
```

```
In [75]: slice3 = A[0:,2:3]

In [76]: print(slice3)
[[3]
 [9]]

In [77]: slice4 = A[1:,2:4]

In [78]: print(slice4)
[[ 9 10]]

In [79]: slice5 = A[0:,2:4]

In [80]: print(slice5)
[[ 3  4]
 [ 9 10]]
```

Here are some additional examples using matrix A that we created for self-check 3. I encourage you to try these examples and maybe experiment on your own as well to better understanding slicing and indexing.

Slice 1 is a row operation. The start and end is not specified, but the interval is three. Since we only have rows 0 and 1, we only get back row zero.

Slice 2 contains a comma, so we're giving instructions for rows and columns. We're asking for all rows, and columns up through column 3.

Slice 3 asks for all rows, and columns starting with column two, up through and not including column 3, so this returns all rows of column 2 only.

Slice 4 is interesting. Here we're saying starting with row 1, return column 2 up through column 4, so we get two values back, 9 and 10, which is row 1, columns 2 and 3.

Finally slice 5 is similar to slice 3 but returns one more column

because the end is set at column 4 rather than 3.

Additional
Examples

```
In [82]: x2 = np.array([[1,2,3], [4,5,6], [7,8,9]])  
  
In [83]: print(x2)  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]  
  
In [84]: slice1 = x2[1:3,1:3]  
  
In [85]: print(slice1)  
[[5 6]  
 [8 9]]  
  
In [86]: slice2 = x2[0:2,1:3]  
  
In [87]: print(slice2)  
[[2 3]  
 [5 6]]
```

Here are more examples with matrix x2. Here slice 1 says start with row 1 up through but not including row 3, and column 1 up through but not including column 3. Slice 2 says start with row zero up to but not including row two, and column 1 up to but not including column 3.

Additional Examples

```
In [89]: x3 = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])  
  
In [90]: print(x3)  
[[ 1  2  3  4]  
 [ 5  6  7  8]  
 [ 9 10 11 12]  
[13 14 15 16]]  
  
In [91]: slice1 = x3[0:4:2,1:3]  
  
In [92]: print(slice1)  
[[ 2  3]  
[10 11]]  
  
In [93]: slice2 = x3[1:4:2,0:4:2]  
  
In [94]: print(slice2)  
[[ 5  7]  
[13 15]]
```

Here's one last slide of examples, this time with matrix x3 which has four rows and four columns, so zero through three and zero through three. Slice 1 asks for row zero up to but not including row 4, with a step of 2, so every other row, and asks for column 1 up to but not including column 3, so we get back column 1 and column 2 of row 0 and row 2. Finally, slice 2 says start at row 1, go up to but not including row 4, and get every other row AND start at column zero, go up to but not including column 4, and get every other column. So we get back four elements. The 5, which is row 1 column zero, the 7 which is row 1 column 2, the 13 which is row 3 column 0 and 15 which is row 3 column 2.

```
In [64]: x1
Out[64]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [65]: x1[-1]
Out[65]: array([ 7,  8,  9, 10])

In [66]: x1[::-1]
Out[66]:
array([[ 7,  8,  9, 10],
       [ 4,  5,  6,  7],
       [ 1,  2,  3,  4]])
```

What would we get if we did this?:

x1[2:0:-1,3:1:-2]

```
In [64]: x1
Out[64]:
array([[ 1,  2,  3,  4],
       [ 4,  5,  6,  7],
       [ 7,  8,  9, 10]])

In [65]: x1[-1]
Out[65]: array([ 7,  8,  9, 10])

In [66]: x1[::-1]
Out[66]:
array([[ 7,  8,  9, 10],
       [ 4,  5,  6,  7],
       [ 1,  2,  3,  4]])
```

What would we get if we did this?:

`x1[2:0:-1,3:1:-2]`

```
In [73]: x1[2:0:-1,3:1:-2]
Out[73]:
array([[10],
       [ 7]])
```

Can you explain why?

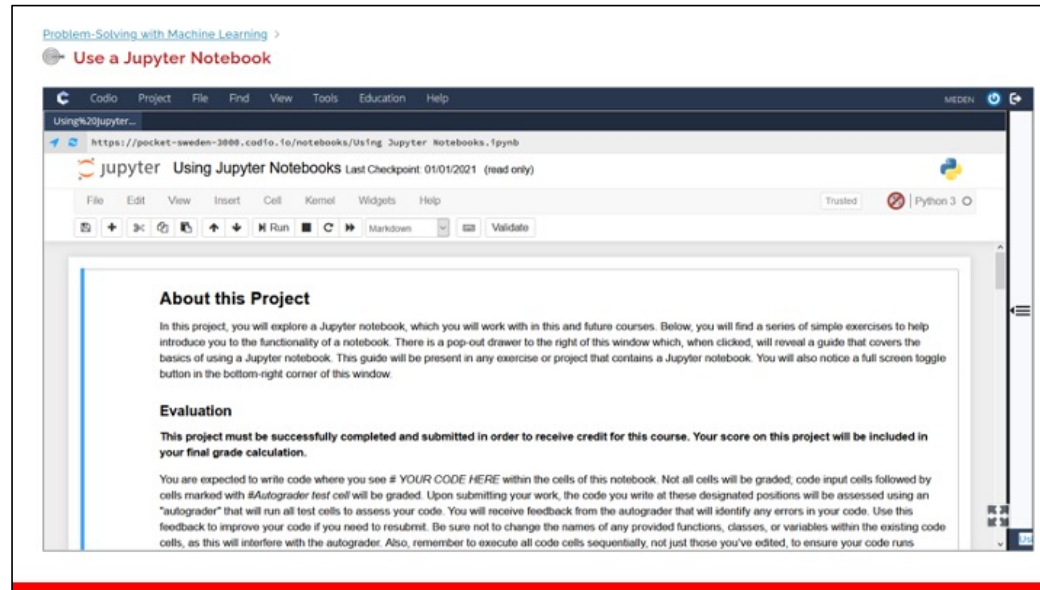
More NumPy Goodness

```
np.mean()  
np.median()  
np.square()  
np.sqrt()  
np.log()  
np.exp()  
np.repeat()
```

Nesting functions can be useful:

```
np.square(np.sqrt(x)) = x
```

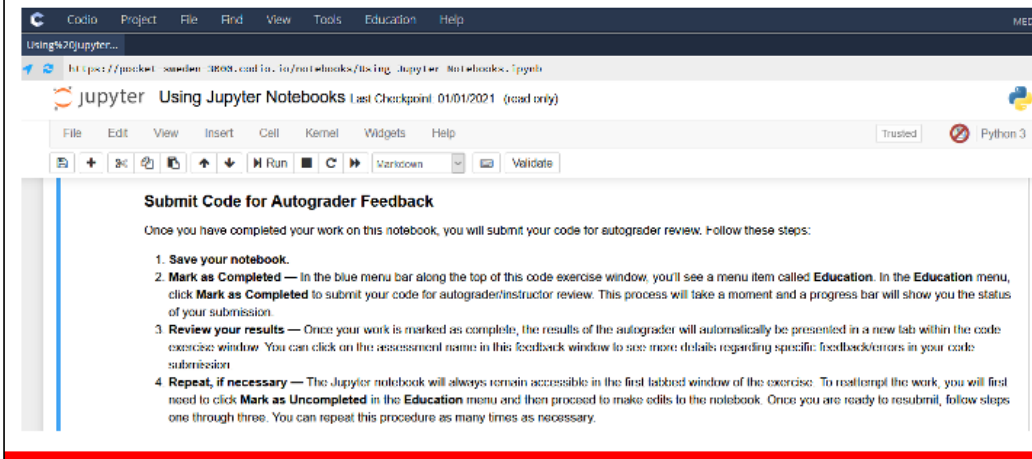
```
np.log(np.exp(x)) = x
```



Also built into the course in the codio environment are Jupyter notebooks. So you won't need to download anything to do the coding assignments in this course. Jupyter Notebook is an open-source application that can be to test, document, run, and share code.

Let's look at this assignment, Use a Jupyter Notebook. There are detailed instructions that will step you through using a Jupyter notebook, so just read through and do what is asked of you for each of the steps. This is a very straightforward assignment, and should not take very long.

Submit Assignments - Follow Instructions



The screenshot shows a Jupyter Notebook interface. At the top, there's a title bar with 'C' and 'Using Jupyter...'. Below it, a browser address bar shows a URL. The Jupyter interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, running, and other actions. The main content area displays the following instructions:

Submit Code for Autograder Feedback

Once you have completed your work on this notebook, you will submit your code for autograder review. Follow these steps:

1. **Save your notebook.**
2. **Mark as Completed** — In the blue menu bar along the top of this code exercise window, you'll see a menu item called **Education**. In the **Education** menu, click **Mark as Completed** to submit your code for autograder/instructor review. This process will take a moment and a progress bar will show you the status of your submission.
3. **Review your results** — Once your work is marked as complete, the results of the autograder will automatically be presented in a new tab within the code exercise window. You can click on the assessment name in this feedback window to see more details regarding specific feedback/errors in your code submission.
4. **Repeat, if necessary** — The Jupyter notebook will always remain accessible in the first tabbed window of the exercise. To resubmit the work, you will first need to click **Mark as Uncompleted** in the **Education** menu and then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one through three. You can repeat this procedure as many times as necessary.

Once you've completed the assignment, follow the instructions for submitting the assignment. Notice that it tells you to save the notebook and then mark it as complete. As the instructions state, you'll click on the Education tab on the menu above, and one of the options is Mark as Complete. Then the Autograder will run and you will soon see your results. You can make changes if needed, and resubmit the assignment.

Challenging Lab Survival Tips and Tricks

Print statements: at each step, print the current output

```
print("The value at step 3 = {}".format(variablename))
```

"b" - Insert cells and use them to perform experiments

Helpful hint: label a cell "Begin testing" and another one "End testing" at each section and keep your work.

Test Cells and Grader Functions

Replicate the tests, and stages of tests. What are they trying to do?

Look at the output of the grader function. How might that be produced?

<https://jasondeden.medium.com/debugging-code-issues-using-print-statements-and-simple-test-data-84d9487d1254>

New to Notebooks?

Lots of great tutorials on YouTube that cover the basics. If you're brand new, check them out.

Specifics for our environment:

<https://drive.google.com/file/d/1BM1EAFX3OSdlbu-DVtAVuqJnkZHu6p-l/view?usp=sharing>

(Low production quality, but helpful content...)

Module 4 Assignments

Euclidean Distance Function Without Loops

Build a Facial Recognition System

There are two assignments for module 4 of this course, and they are due by the last day of this course. You can submit the assignments early, and you can re-submit them if needed, as long as you're submitting before the deadline. The autograder portions of the assignment will give you feedback as soon as you run those portions.

For these two assignments, you will need to know and understand linear algebra concepts like dot products and matrix operations, and how to do matrix operations with NumPy as well as transformations such as reshape. So it is important to have a good understanding of the material in the Linear Algebra courses, as well as a good understanding of the material in modules 1, 2 and 3 of this course.

Questions?

Thank You

End of Live Session 1

Appendix

Optional content that can be covered if

Python Functions

```
def samplefunction(inputarray, scalarvalue):  
    import numpy as np #just in case hasn't been  
done  
    timesscalar = inputarray * scalarvalue  
    finalarray = np.square(timesscalar)  
    return finalarray
```

```
In [113]: def samplefunction(inputarray, scalarvalue):  
...:     import numpy as np #just in case hasn't been done  
...:     timesscalar = inputarray * scalarvalue  
...:     finalarray = np.square(timesscalar)  
...:     return finalarray
```

Python Functions

Function Name

Function Input Values
(must be provided when function is called)

```
def samplefunction(inputarray, scalarvalue):  
    import numpy as np #just in case hasn't been  
done  
    timesscalar = inputarray * scalarvalue  
    finalarray = np.square(timesscalar)  
    return finalarray
```

End of function,
values to return
follow

One or more values
(multiple can be separated by a comma)
to return after running the function

Code to be run based on the input

Note: Indentation matters!

Call a function

```
functionname(input1, input2,...)
```

```
In [116]: x1
```

```
Out[116]:
```

```
array([[ 1,  2,  3,  4],  
       [ 4,  5,  6,  7],  
       [ 7,  8,  9, 10]])
```

```
In [117]: samplefunction(x1,3)
```

```
Out[117]:
```

```
array([[ 9,  36,  81, 144],  
       [144, 225, 324, 441],  
       [441, 576, 729, 900]])
```